Inferring Termination Conditions for Logic Programs using Backwards Analysis

Samir Genaim and Michael Codish
The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
{genaim,mcodish}@cs.bgu.ac.il

Abstract

This paper focuses on the inference of modes for which a logic program is guaranteed to terminate. This generalizes traditional termination analysis where an analyzer tries to verify termination for a specified mode. The contribution is a methodology which combines traditional termination analysis and backwards analysis to obtain termination inference. We demonstrate the application of this methodology to enhance a termination analyzer to perform also termination inference.

1 Introduction

This paper focuses on the inference of modes for which a logic program is guaranteed to terminate. This generalizes traditional termination analysis where an analyzer tries to verify termination for a specified mode. For example, for the classic *append*/3 relation, a standard analyzer will determine that a query of the form $append(x, y, z)$ with x bound to a closed list terminates and likewise for the query in which z is bound to a closed list. In contrast, termination inference provides the result $append(x, y, z) \leftarrow x \vee z$ with the interpretation that the query $append(x, y, z)$ terminates if x or z are bound to closed lists. We refer to the first type of analysis as performing *termination checking* and to the second as *termination inference*. We consider universal termination using Prolog's leftmost selection rule and we assume that unifications do not violate the occurs check.

Termination inference is considered previously by Mesnard and coauthors in [15, 16, 17]. Their work can be used at `http://www.complang.tuwien.ac.at/cti`. Here, we make the observation that the missing link between termination checking and termination inference is *backwards analysis*. Backwards analysis is concerned with the following type of question: Given a program and an assertion at a given program point, what are the weakest requirements on the inputs to the program which guarantee that the assertion will hold whenever execution reaches that point.

In a recent paper, King and Lu [12] describe a framework for backwards analysis for logic programs set in the context of abstract interpretation. In their approach, the underlying abstract domain is required to be condensing or equivalently, a complete Heyting algebra. Basically, this property ensures that we can work backwards from an assertion in the program to find the weakest requirement on calls to the program which guarantee that an assertion will hold.

To demonstrate this link between termination checking and termination inference, we apply the framework for backwards analysis described by King and Lu [12] to enhance the termination (checking) analyzer described in [5] to perform also termination inference. We use the domain Pos for groundness dependencies which satisfies the required properties. The use of backwards analysis provides a formal justification for termination inference and leads to a simple and efficient implementation similar in power to that described in [16]. It too can be used online at `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

In rest of the paper, Section 2 provides some background and a motivating example. Section 3 reviews the idea of backwards analysis. Section 4 illustrates how to combine termination analysis with backwards analysis in order to obtain termination inference. Section 5 presents an experimentation with the resulting analyzer for termination inference. Finally, Section 6 reviews related work and Section 7 concludes.

2 Preliminaries and Motivating Example

We assume a familiarity with the standard definitions for logic programs [13, 1] as well as with the basics of abstract interpretation [6, 7]. This section describes the standard program analyses upon which we build in the rest of the paper. For notation, in brief: Variables in logic programs are denoted as in Prolog (using the upper case) while in relations, Boolean formula, and other mathematical context we use the lower case. We let $\bar{x}$ denote a tuple of distinct variables $x_1, \ldots, x_n$. To highlight a specific point in a program we use labels of the form ⓐ.

Size relations and instantiation dependencies rest at the heart of termination analysis: size information to infer that some measure on program states decreases as computation progresses; and instantiation information, to infer that the underlying domain is well founded. Consider the recursive clause of the *append*/3 relation: $\mathit{append}([X|Xs], Ys, [X|Zs]) \leftarrow \mathit{append}(Xs, Ys, Zs)$. It does not suffice to observe that the size of the first and third arguments decrease in the recursive call. To guarantee termination one must also ensure that one of these arguments is sufficiently instantiated in order to argue that this recursion can be activated only a finite number of times.

Instantiation information is obtained through abstract interpretation over the domain Pos which consists of the positive Boolean functions augmented with a bottom element (representing the formula $false$). The elements of the domain are ordered by implication and represent equivalence classes of propositional formula. This domain is usually associated with its use to infer groundness dependencies where a formula of the form $x \wedge (y \rightarrow z)$ is interpreted to describe a program state in which x is definitely bound to a ground term and there exists an instantiation dependency such that whenever y becomes bound to a ground term then so does z. Similar analyses can be applied to infer dependencies with respect to other notions of instantiation. We denote the approximation of the success set in Pos for groundness of a program P by $[\![P]\!]^{suc}_{gr}$. For details on Pos see [14].

Size relations express linear information about the sizes of terms (with respect to a given norm function) [3, 4, 8, 11]. For example, the relation $x \leq z \ \wedge \ y \leq z$ describes a program state in which the sizes of the terms associated with x and y are less or equal to the size of the term associated with z. Similarly, a relation of the form $z = x + y$ describes a state in which the sum of the sizes of the terms associated with x and y is equal to the size of the term associated with z. Several methods for inferring size relations are described in the literature [3, 4, 8, 9]. They differ primarily in their approach to obtaining a finite analysis as the abstract domain of size relations contains infinite chains.

Throughout this paper we will use the so-called term-size norm for size relations for which the corresponding notion of instantiation is groundness. We base our presentation on the termination analyzer described in [5] although we could use as well almost any of the alternatives described in the literature. This analyzer is based on a bottom-up T_P like semantics which makes loops observable in the form of binary clauses. This provides a convenient starting point for termination inference as derived in this paper. We denote the abstraction of this semantics over the domain of size relations as $[\![P]\!]^{bin}_{size}$. Each element of $[\![P]\!]^{bin}_{size}$ represents a loop and is of the form $p(\bar{x}) \leftarrow \pi, p(\bar{y})$ where π is a conjunction of linear constraints.

For the full picture, we note that the analyzer of [5] involves two phases. In the first phase, the user provides a program and the analyzer computes an approximation of its loops over two domains: size relations and instantiation dependencies. In the second phase, the user specifies the mode of an initial goal and the analyzer performs a termination check. For termination inference we make use of the descriptions of the loops with size information obtained in the first part of the first phase of the termination analysis. In addition, we apply a standard Pos analyzer to obtain instantiation dependencies which approximate the success set of the given program. We demonstrate our approach by example in four steps:

The first step: Consider the *append*/3 relation.

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

The termination checker [5] reports a single abstract binary clause:

```
append(A,B,C) :- [D<A, F<C, B=E], append(D,E,F).
```

indicating that subsequent calls $append(A, B, C)$ and $append(D, E, F)$ in a computation, involve a decrease in size for the first and third arguments ($D < A$ and $F < C$) and maintain the size of the second argument ($B = E$). To guarantee that this loop may be traversed only a finite number of times, it is sufficient to require that A or C be sufficiently instantiated. This can be expressed as a Boolean condition: $append(x, y, z) \leftarrow (x \vee z)$.

Backwards analysis is now applied to infer the weakest conditions on the program's predicates which guarantee this condition. For this example the inference is complete and we have derived the result: $\mathit{terminates}(\mathit{append}(x,y,z)) \leftarrow x \vee z$.

The second step: Consider the use of $\mathit{append}/3$ to define list membership. Adding the clause:

```
member(X,Xs) :- append(A,[X|B],Xs).
```

to the program introduces no additional loops. Backwards analysis should specify the weakest condition on $\mathit{member}(X,Xs)$ which guarantees the termination condition $A \vee Xs$ for $\mathit{append}(A,[X|B],Xs)$. This is obtained through projection which for backwards analysis is defined in terms of universal quantification as $\forall_A.(A \vee Xs)$. The resulting Boolean precondition for $\mathit{member}/2$ is: $\mathit{terminates}(\mathit{member}(x,y)) \leftarrow y$.

The third step: We now add to the program a definition for the subset/2 relation:

```
subset([X|Xs],Ys) :- member(X,Ys), subset(Xs,Ys).
subset([],Ys).
```

Termination checking reports an additional loop:

```
subset(A,B) :- [B=D,C<A], subset(C,D).
```

which will be traversed a finite number of times if A is sufficiently instantiated. For the first clause to terminate both loops must terminate: for $\mathit{append}/3$ in the call to $\mathit{member}(X,Ys)$ and for $\mathit{subset}/2$ in the call to $\mathit{subset}(Xs,Ys)$. So both Xs and Ys must be instantiated which implies that both arguments of $\mathit{subset}/2$ should be inputs. Namely, $\mathit{terminates}(\mathit{subset}(x,y)) \leftarrow x \wedge y$.

The fourth step: This step demonstrates that the precondition on a call in a clause body may be (partially) satisfied by answers to calls which precede it. Consider adding to the program a clause:

```
s(X,Y,Z) :- ⓐ append(X,Y,T), ⓑ subset(T,Z).
```

which defines a relation $s(x,y,z)$ such that the set z contains the union of sets x and y. The preconditions for termination derived in the previous steps specify the conditions $x \vee t$ and $t \wedge z$ at points ⓐ and ⓑ respectively. In addition, from a standard groundness analysis we know that on success $\mathit{append}(x,y,t)$ satisfies $(x \wedge y) \leftrightarrow t$. So, instead of imposing on the clause head both conditions from the calls in its body, as we did in the previous step, we may weaken the second condition in view of the results from the first call. Namely the termination condition inferred for $s(x,y,z)$ is

$$\forall_t.((x \vee t) \wedge (((x \wedge y) \leftrightarrow t) \rightarrow t \wedge z)) \equiv x \wedge y \wedge z.$$

In general, the steps illustrated above need to be applied in iteration. Though what we have shown works correctly for our example. In the next section we describe more formally the steps required for backwards analysis.

3 Backward Analysis

This section presents an abstract interpretation for backwards analysis using the domain Pos distilled from the general presentation given in [12]. Clauses are assumed to be normalized and contain assertions so that they are of the form $h(\bar{x}) \leftarrow \mu \diamond b_1, \ldots, b_n$ where μ is a Pos formula, interpreted as an instantiation condition that must hold at the beginning of the clause, and b_i is either an atom, or a unification operation.

The analysis associates preconditions, specified in Pos, with the predicates of the program. Initialized to *true* (the top element in *Pos*) these preconditions become more restrictive (move down in Pos) through iteration until they stabilize. At each iteration, clauses are processed from right to left using the current aproximations for preconditions on the calls together with the results of a standard groundness analysis to infer new approximations for these preconditions.

For the basic step, consider a clause of the form: $p \leftarrow \ldots \textcircled{a}, q, \textcircled{b} \ldots$ and assume that the current approximation for the precondition for q is φ_q, the success of q is approximated by ψ_q, and that processing the clause from right to left has already propagated a condition φ_b at the point $\textcircled{b}$. Then, to insure that φ_b will hold after the success of q, it suffices to require at $\textcircled{a}$ the conjunction of φ_q with the weakest condition σ such that $(\sigma \wedge \psi_q) \rightarrow \varphi_b$. This σ is precisely the pseudo-complement [10] of ψ_q with respect to φ_b, obtained as $\psi_q \rightarrow \varphi_b$. So propagating one step to the left gives the condition $\varphi_a = \varphi_q \wedge (\psi_q \rightarrow \varphi_b)$.

Now consider a clause $h(\bar{x}) \leftarrow \mu \diamond b_1, \ldots, b_n$ with an assertion $\mu \in$ Pos at the left of the body. Assume that the current approximation for the precondition of $h(\bar{x})$ is φ and let ψ_i and φ_i denote respectively the approximation of the success set of b_i (obtained through standard groundness analysis) and the current precondition for b_i $(1 \leq i \leq n)$. Backwards analysis infers a new approximation φ' on $h(\bar{x})$ by consecutive application of the basic step described above. We start with $e_{n+1} = true$ and through n steps (with i going from n to 1) compute a condition $e_i = \varphi_i \wedge (\psi_i \rightarrow e_{i+1})$ which should hold just before the call to b_i. After computing e_1 we take $e_0 = \mu \wedge e_1$ and project e_0 on the variables $\bar{x}$ of the head by means of universal quantification. The new condition is finally obtained through conjunction with the previous condition φ.

There is one subtlety in that Pos is not closed under universal quantification. To be precise, projection of x from φ is defined as the largest element in Pos which implies $\forall_x.\varphi$. When $\forall_x.\varphi$ is not positive then the projection gives $false$ which is the bottom element in Pos.

Example 1 *Consider the clause*

`subset(A,B) :-` $\textcircled{e_0}$ `A` $\diamond$ $\textcircled{e_1}$ `A=[X|Xs],` $\textcircled{e_2}$ `B=Ys,`
$\textcircled{e_3}$ `member(X,Ys),` $\textcircled{e_4}$ `subset(Xs,Ys)` $\textcircled{e_5}$.

(the assertion A states that the first argument must be ground) where the success patterns (derived by a standard groundness analysis) and the current approximation of the preconditions are (respectively):

$$\llbracket P \rrbracket^{suc}_{gr} = \left\{ \begin{array}{l} member(x,y) \leftarrow (y \rightarrow x) \\ subset(x,y) \leftarrow (y \rightarrow x) \end{array} \right\} I = \left\{ \begin{array}{l} member(x,y) \leftarrow y \\ subset(x,y) \leftarrow x \end{array} \right\}.$$

Starting from $e_5 = true$, the conditions $e_4, \ldots, e_0$ are obtained as follows:

$$\begin{array}{lll} e_4 = & Xs & \wedge \quad ((Ys \rightarrow Xs) \rightarrow e_5) \\ e_3 = & Ys & \wedge \quad ((Ys \rightarrow X) \rightarrow e_4) \\ e_2 = & true & \wedge \quad ((B \leftrightarrow Ys) \rightarrow e_3) \\ e_1 = & true & \wedge \quad ((A \leftrightarrow (X \wedge Xs) \rightarrow e_2) \\ e_0 = & A \wedge e_1 & \end{array}$$

Projecting e_0 to the variables in the head gives $\forall_{Xs,Ys,X}.(e_0) = A \wedge B$. Which leads to the new precondition $subset(x,y) \leftarrow x \wedge y$.

In [12], the authors formalize backwards analysis as the greatest fixed point of an operator over Pos. We have implemented this operator as a meta interpreter which manipulates Boolean formula represented using binary decision diagrams. See Appendix A.

4 Termination Inference

Termination inference proceeds as follows: **(a)** apply the initial phase of the termination analysis described in [5] to obtain a set of abstract binary clauses which approximate the loops in the program; **(b)** extract Boolean conditions on the instantiation of variables to guarantee that each of the identified loops can be executed a finite number of times; and **(c)** apply backwards analysis as defined in [12] to infer the weakest modes on initial goals which guarantee that these Boolean conditions will hold.

The following definition specifies how to extract from the results of the initial phase of the termination analysis those assertions from which backwards analysis starts.

Definition 1 *The termination assertion $\mu(p(\bar{x}))$ for the predicate p/n in the program P is determined as follows:*

1. *The condition for a single binary clause is*

$$\mu(p(\bar{x}) \leftarrow \pi, p(\bar{y})) = \bigvee \left\{ \bigwedge_{i \in I} x_i \;\middle|\; I \subseteq \{1, \ldots, n\},\ \pi \models (\sum_{i \in I} x_i > \sum_{i \in I} y_i\ ) \right\}$$

2. *and the assertion for $p(\bar{x})$ is:*

$$\mu(p(\bar{x})) = \bigwedge \left\{ \mu(\ell) \;\middle|\; \ell = p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in \llbracket P \rrbracket^{bin}_{size} \right\}$$

In theory we could obtain a stronger assertion by considering arbitrary linear combinations of the arguments of $p(\bar{x})$ instead of restricting coefficients to 0 and 1 as we do in the definition by taking subsets I of the argument positions. In practice, in the implementation, we impose a weaker assertion which does not consider all subsets $I \subseteq \{1, \ldots, n\}$, but rather only the singletons (to detect argument positions which decrease in size) and the set of arguments which do not decrease in size. This simplistic approach works well in practice. A more elaborate approach is described in [19].

Example 2 *Consider as P the split/3 relation (from merge sort):*

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

The binary clauses obtained by the analyzer of [5] are:

$$\ell_1 = \quad split(x_1,x_2,x_3) \leftarrow [y_1 < x_1, y_3 < x_2, x_3 = y_2], split(y_1,y_2,y_3).$$

$$\ell_2 = \quad split(x_1,x_2,x_3) \leftarrow [y_1 < x_1, y_2 < x_2, y_3 < x_3], split(y_1,y_2,y_3).$$

$$\ell_3 = \quad split(x_1,x_2,x_3) \leftarrow [y_1 < x_1, y_3 < x_2, y_2 < x_3], split(y_1,y_2,y_3).$$

We have $\mu(\ell_1) = \mu(\ell_3) = x_1 \vee (x_2 \wedge x_3)$, because $y_1 < x_1$ and $y_2 + y_3 < x_2 + x_3$; and $\mu(\ell_2) = x_1 \vee x_2 \vee x_3$ because $y_1 < x_1, y_2 < x_2, y_3 < x_3$. The assertion for split/3 is $\mu(split(x_1,x_2,x_3)) = (x_1 \vee (x_2 \wedge x_3)) \wedge (x_1 \vee x_2 \vee x_3) = x_1 \vee (x_2 \wedge x_3)$.

Definition 2 *A mode is a tuple of the form $p(m_1,\ldots,m_n)$ where m_i $(1 \leq i \leq n)$ is either **b** ('bound') or **f** ('free'). We say that $p(m_1,\ldots,m_n)$ is safe for $p(x_1,\ldots,x_n)$ if the conjunction $\wedge\{x_i \mid m_i = b\}$ implies the termination condition inferred by backwards analysis for $p(\bar{x})$.*

Example 3 *In the previous example we inferred $\mu(split(x_1,x_2,x_3)) = \varphi$ with $\varphi = x_1 \vee (x_2 \wedge x_3)$. Hence $p(b,f,f)$ and $p(f,b,b)$ are safe modes for split/3 because $x_1 \rightarrow \varphi$ and $(x_2 \wedge x_3) \rightarrow \varphi$.*

The correctness of the method follows from the results of [5] and [12].

Theorem 1 *Let P be a logic program and $p(\bar{m})$ a safe mode for $p(\bar{x})$. Then P terminates for $p(\bar{m})$.*

Proof (sketch). Let $p(\bar{m})$ be a safe mode for p/n, let G be an initial query of this mode and let Q be a call to a predicate q/k which loops in an SLD derivation for G. Let μ be the Boolean assertion imposed on q/k by termination inference and φ be the termination condition for p/n. From the correctness of backwards analysis [12] it follows that μ must hold for Q because $\wedge\{x_i|m_i = b\} \Rightarrow \varphi$ and φ guarantee μ. From the construction of μ which considers the binary clauses for q and the correctness of the termination analysis [5] it follows that the loop on q/k must terminate.

5 Experimental Results

We have implemented an analyzer for termination inference based on the ideas presented in the previous sections. The implementation combines the first phase of the termination analysis described in [5] (binary clauses with size information) and the backwards analysis algorithm described in [12]. The implementation of backwards analysis is described in the appendix. It is based on two meta-interpreters: one for the approximation of the success set (computes a lfp); and the other for the backwards analysis itself (computes a gfp). Both manipulate binary decision diagrams using a package written by Armstrong and Schachte (used in [2] and described in [18]).

We use the same benchmarks as used in [16]. Our analyzer runs SICStus 3.7.1 on a Pentium III 500MHZ machine with 128MB RAM under Linux RedHat 7.1 (kernel 2.4.2-2). Timings for cTI are reported for a faster machine (Athlon 750MHz, 256Mb, SICStus 3.8.4). Table 1 indicates analysis times (in seconds). The columns indicate the cost for: **Pre**: preprocessing (reading, abstraction, computing sccs, printing results); **Size**: size analysis (to approximate binary clauses); **Pos**: groundness dependencies analysis (to approximate answers); **Ass**: computing instantiation assertions from the abstract binary clauses; **BA**: backwards analysis; **cTI**: the analysis using cTI (as reported in [16]). To conform with [16] all programs are analyzed using the term-size norm with widening applied every third iteration, except for the programs marked by a $\star$ for which the list-length norm is applied and widening is performed every fourth iteration.

The two blocks of programs in Table 1 correspond respectively to those from Tables 2 and 5 in [16]. For the first block we infer exactly the same termination conditions as cTI. For the second block (of larger programs), we infer a positive termination condition for the same percentage of predicates as does cTI, except for the last three programs where a “$\oplus$” indicates that we infer a positive termination condition for more predicates than does cTI and a “$\ominus$” vice-versa.

In comparison with standard termination checking, the first three columns (**Pre**, **Size**, and **Pos**) correspond to phases which need to be performed during termination checking. The next two columns (**Ass** and **BA**) indicate the extra cost for termination inference. It is interesting to note that the times in these columns are very small and in general less expensive than the additional cost to perform checking (for a single mode) in the second phase of the standard termination analysis.

6 Related Work

This paper draws on results from two areas: termination analysis and backwards analysis. It combines these results to obtain an analyzer which infers sufficient conditions for termination of logic programs. For termination analysis we build on the implementation described in [5]. For backwards analysis we have implemented the algorithm described in [12]. The analyzer we obtain for termination inference is similar to the one described in [16].

Backwards reasoning for imperative programs dates back to the early days of

program	Pre	Size	Pos	Ass	BA	Total	cTI
permute	0.01	0.12	0.00	0.01	0.00	0.14	0.15
duplicate	0.00	0.03	0.00	0.00	0.00	0.03	0.05
sum1	0.00	0.05	0.00	0.01	0.00	0.06	0.18
merge	0.00	0.19	0.00	0.02	0.00	0.21	0.26
dis-con	0.00	0.09	0.00	0.00	0.01	0.10	0.24
reverse	0.02	0.05	0.00	0.01	0.00	0.08	0.08
append	0.00	0.05	0.01	0.00	0.00	0.06	0.09
list	0.01	0.01	0.01	0.00	0.00	0.03	0.01
fold	0.00	0.05	0.00	0.01	0.00	0.06	0.10
lte	0.00	0.06	0.01	0.00	0.00	0.07	0.13
map	0.00	0.05	0.00	0.00	0.00	0.05	0.09
member	0.00	0.04	0.01	0.00	0.00	0.05	0.03
mergesort	0.00	0.44	0.00	0.02	0.00	0.46	0.43
mergesort⋆	0.01	0.98	0.01	0.01	0.01	1.02	0.57
mergesort_ap	0.00	0.63	0.00	0.04	0.00	0.67	0.79
mergesort_ap⋆	0.00	1.29	0.03	0.03	0.00	1.35	0.92
naive_rev	0.01	0.08	0.01	0.00	0.00	0.10	0.12
ordered	0.00	0.03	0.00	0.00	0.00	0.03	0.04
overlap	0.00	0.05	0.01	0.00	0.00	0.06	0.05
permutation	0.02	0.10	0.00	0.01	0.00	0.13	0.15
quicksort	0.00	0.38	0.01	0.04	0.00	0.43	0.39
sum2	0.00	0.08	0.00	0.01	0.00	0.09	0.08
select	0.00	0.09	0.01	0.00	0.00	0.10	0.09
subset	0.01	0.09	0.01	0.00	0.00	0.11	0.12
ann	0.16	4.46	0.07	0.30	0.03	5.02	5.01
bid	0.04	0.62	0.02	0.05	0.01	0.74	0.79
boyer	0.10	2.55	0.05	0.04	0.01	2.75	3.53
browse	0.04	0.96	0.01	0.15	0.00	1.16	1.81
credit	0.04	0.43	0.02	0.04	0.01	0.54	0.61
peephole	0.09	4.46	0.04	0.07	0.02	4.68	12.08
plan	0.03	1.03	0.02	0.03	0.01	1.12	0.71
qplan	0.13	10.86	0.05	0.51	0.03	11.58	7.30
rdtok ⊕	0.05	2.86	0.02	0.16	0.01	3.10	2.92
read ⊖	0.12	4.43	0.03	0.04	0.03	4.65	6.87
warplan ⊕	0.08	2.54	0.04	0.14	0.03	2.83	3.18

Table 1: Experimental Results

static analysis and has been applied extensively in functional programming. Applications of backwards analysis in the context of logic programming are few. For details concerning other applications of backwards analysis, see [12]. The application described in [12] is similar to ours. There, the authors infer modes for a logic program which guarantee that Prolog builtins do not report instantiation errors. The authors note that when trying to figure out how to run programs written by a third party they typically start from builtins and work backwards to infer the intended modes of use for the program.

Note that our work can also be applied to the same task as it is natural to assume that the intended mode result in terminating computations. So for example where King and Lu infer the mode $x \vee y$ for the predicate $sort(x, y)$ in permutation sort, termination can be guaranteed for the more restrictive mode x; and where they infer for $partition(x_1, x_2, x_3, x_4)$ in the quicksort program the mode $x_2 \wedge (x_1 \vee (x_3 \wedge x_4))$, to guarantee termination we infer the mode x_1. Together, these give the mode $x_1 \wedge x_2$ which is in fact the intended mode for this program. It is interesting to note that both of these analyses can be performed together as a single backwards analysis. We simply take the conjunction of the initial assertions from both analyses.

The only other work on termination inference that we are aware of is that of Mesnard and coauthors. The implementation of Mesnard's analyzer is described in [16] and its formal justification is given in [17]. Their cTI analyzer is very similar to ours, though designed differently.

For the first step our work relies on a traditional termination (checking) analyzer. Hence we avoid getting involved with the specification, implementation and justification of the internal details of this phase. Instead, we consider it a black box which supplies an approximation of the loops which could arise when executing a program. The corresponding component in cTI derives size information just as we do but detects level mappings (the measure on the program state which decreases in a loop) using a more sophisticated, albeit more costly, technique adapted from [19]. In most cases this does not make a practical difference on precision but it is more powerful. In any case, it appears that both analyzers, given the same level mappings will derive the same results in the second phase.

Our second phase is packaged as an instance of backwards analysis. Though similar in essence to the corresponding phase in cTI, ours can be viewed as a black box and simplifies the implementation of the analyzer and its formal justification. There is also a difference: cTI specifies recursive equations for each point in a clause and computes a greatest fixed point using a μ-calculus solver [16]. Backwards analysis propagates Boolean constraints from right to left and avoids some degree of re-computation. Our implementation computes the greatest fixed point using a meta-interpreter written in Prolog. The two techniques appear to be equivalent, but our backwards analysis phase is 5-6 times faster (on a slower machine) than the corresponding phase in cTI (based on the comparison of our Table 1 with Table 5 from [16]).

7 Conclusion

We have demonstrated that backwards analysis provides a useful link between termination checking and termination inference. This leads to a better understanding of termination inference and simplifies the formal justification and the implementation of termination inference. We demonstrate how putting the components together enables us to enhance the termination analyzer described in [5] to perform also termination inference.

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[3] F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Sixth International Workshop on Logic Program Synthesis and Transformation (LOPSTR'96)*, pages 204–223, 1996.

[4] A. Brodsky and Y. Sagiv. Inference of monotonicity constraints in Datalog programs. In *Proceedings of the Eighth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 190–199, 1989.

[5] M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages*, pages 238–252, New York, 1977. ACM Press.

[7] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.

[8] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.

[9] D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(02):117–154, 1995.

[10] R. Giacobazzi and F. Scozzari. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[11] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[12] A. King and L. Lu. A backward analysis for constraint logic programs. Technical report, Kent University, 2001.

[13] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, second edition, 1987.

[14] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[15] F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. *Proc. of JICSLP'96*, pages 7–21, 1996.

[16] F. Mesnard and U. Neumerkel. Applying statsic analysis techniques for inferring termination conditions of logi programs. In *Static Analysis Symposium*, 2001.

[17] F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. Technical report, Universite de La Reunion, 2001.

[18] P. Schachte. *Precise and Efficient Static Analysis of Logic Programs.* PhD thesis, The University of Melbourne, Australia, 1999.

[19] K. Sohn and A. van Gelder. Termination dedection in logic programs using argument sizes. In *Intl. Symp. on Principles of Database Systems*, pages 216–226. ACM Press, 1991.

A Implementing Backwards Analysis

We implement backwards analysis for Pos as a meta-interpreter written in Prolog just as we do for usual groundness analysis. The main differences are: that clause bodies are processed from right to left, standard groundness dependencies are factored through application of the pseudo-complement, the fact that projection is implemented in terms of universal quantification (instead of existential), and since the analysis is moving down from the top element, so between iterations instead of a join (disjunction) we apply the meet (conjunction).

The implementation consists in two meta-interpreters: one for the approximation of the success set (computes a lfp); and the other for the backwards analysis itself (computes a gfp). Both manipulate binary decision diagrams using a BDD package written by Armstrong and Schachte and described in [18] and reported in [2]. In our actual implementation, the interpreters are semi-naive and optimized to consider the strongly connected components in the programs call graph. Here, we illustrate a simplified (but working) version of backwards analysis based on naive fixed point evaluation. Both versions can be obtained from `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

Figure 1 depicts the two interpreters. On the left the interpreter for standard groundness dependencies analysis and on the right the interpreter for backwards analysis. Prior to analysis, clauses are normalized and represented as facts of the form $my\_clause(h(\bar{x}), [b_1, \ldots, b_n])$. Unifications are abstracted and of the form $X = Vars$ where X is a variable and $Vars$ is a conjunction of variables represented as list.

The "control" component (identical for both interpreters) is given in lines 1-5. The call to `iterate` triggers the iteration of an `operator` which continues until a fixed point is reached. Whenever, new information is obtained, a `flag` is raised. Iteration stops when `retract(flag)` fails in the second clause for `iterate`, that is, when no changes where made in the previous iteration.

Groundness Analysis

The `operator` on the left (lines 7-11) performs a standard groundness dependencies analysis. The current Pos approximation for each predicate p/n is maintained as a fact of the form $fact(gr, p(x_1, \ldots, x_n), \varphi)$ where $\varphi \in$ Pos is a represented as a BDD. If there is no such fact for p/n then its current approximation is $false$ (corresponding to the bottom element of the domain). The `operator` provides the inner loop of the algorithm which for each $my\_clause(p(\bar{x}), Body)$ solves the $Body$ using the current Pos approximations (line 9), projects the result on the variables $\bar{x}$ (line 10) and then updates the current approximation for p/n with the result (line 11).

The relation $solve(Body, \varphi_{in}, \varphi_{out})$ defined in lines 16-24 specifies that the solution of the $Body$ with the approximation φ_{in} is φ_{out}. If the $Body$ is empty (line 16), then $\varphi_{in} = \varphi_{out}$; Otherwise the first call in the body is either a unification (line 17) of the form $X = Vars$ in which case we take $\varphi_1 = X \leftrightarrow Vars$ (line 18), or else it is an $Atom$ (line 21) in which case we take as φ_1 the current approximation for $Atom$ (line 22). In both cases we continue solving the rest of the $Body$ with $\varphi_2 = \varphi_{in} \wedge \varphi_1$.

The relation $bdd\_lub(Head, \varphi)$ (lines 28-40) updates the current approximation φ_1 for the $Head$ replacing it by $\varphi \vee \varphi_1$. There are three cases: If φ entails φ_1 (line 30), then there is no update to perform. Otherwise, the current fact for the approximation is removed, if it exists, and replaced by the disjunction (line 39). If there is no current fact then the current approximation is false, so φ is the new approximation (line 37). If the *lub* changes the current approximation, then the flag is raised (at line 40) unless it is already up.

The $get\_fact(Type, Atom, \varphi)$ relation (lines 42-49 and used in both interpreters) picks up the current Pos approximation for an *Atom* and performs a renaming.

Backwards Analysis

The current approximation for backwards analysis of each predicate p/n is maintained as a fact of the form $fact(ba, p(x_1, \ldots, x_n), \varphi)$ where $\varphi \in \mathsf{Pos}$. If there is no such fact for p/n then its current approximation is *true* (corresponding to the top element of the domain). In addition, there is an initial assertion (for termination inference) associated with each predicate and maintained as a fact of the form $assertion(p(\bar{x}), \varphi)$.

The interpreter on the right is quite similar to the one on the left. In the *operator* relation (lines 7-14): we consider for each clause its initial assertion (line 9) and process the clause from right to left (line 10). Projection is performed by universal quantification (line 13) and instead of a lub we apply a glb (at line 14).

In the *solve* relation (lines 16-26) implication is applied as pseudo complement on the result of a unification (line 19) or on the result of the groundness analysis (line 24). The definition of the $bdd\_glb/2$ relation (lines 28-40) is dual to the $bdd\_lub/2$ relation to its left.

```
iterate ← operator, fail.
iterate ←
    retract(flag),
    iterate.
iterate.

operator ←
    my_clause(Head,Body),
    solve(Body,true,φ1),
    bdd_existential_q(φ1,Head,φ2),
    bdd_lub(Head,φ2).




solve([],φ,φ).
solve([X=Vars | Xs],φin,φout) ←
    bdd_iff(X,Vars,φ1),
    bdd_and(φin,φ1,φ2),
    solve(Xs,φ2,φout).
solve([Atom | Xs],φin,φout) ←
    get_fact(gr,Atom,φ1),
    bdd_and(φ,φin,φ2),
    solve(Xs,φ2,φout).



bdd_lub(Head,φ) ←
    fact(gr,Head,φ1),
    bdd_entailed(φ,φ1),
    !.
bdd_lub(Head,φ) ←
    (
     retract(fact(gr,Head,φ1)) →
         bdd_or(φ1,φ,φ2),
    ;
         φ2 = φ
    ),
    assert(fact(gr,Head,φ2)),
    (flag → true ; assert(flag)).

get_fact(Type,Atom,Φ) ←
    functor(Atom,Name,A),
    functor(F_Atom,Name,A),
    Atom =.. [_ | A_Vs],
    F_Atom =.. [_ | F_Vs],
    fact(Type,F_Atom,φ),
    bdd_rename(φ,F_Vs,A_Vs,Φ),
    !.
```

```
iterate ← operator, fail.
iterate ←
    retractall(flag),
    iterate.
iterate.

operator ←
    my_clause(Head,Body),
    get_assertion(Head,φ0)
    reverse(Body,RBody),
    solve(RBody,true,φ1),
    bdd_and(φ0,φ1,φ2),
    bdd_universal_q(φ2,Head,φ3),
    bdd_glb(Head,φ3).

solve([],φ,φ).
solve([X=Vars | Xs],φin,φout) ←
    bdd_iff(X,Vars,φ1),
    bdd_implies(φ1,φin,φ2),
    solve(Xs,φ2,φout).
solve([Atom | Xs],φin,φout) ←
    get_fact(gr,Atom,φ1),
    get_fact(ba,Atom,φ2),
    bdd_implies(φ1,φin,φ3),
    bdd_and(φ2,φ3,φ4),
    solve(Xs,φ4,φout).

bdd_glb(Head,φ) ←
    fact(ba,Head,φ1),
    bdd_entailed(φ1,φ),
    !.
bdd_glb(Head,φ) ←
    (
     retract(fact(ba,Head,φ1)) →
         bdd_and(φ1,φ,φ2),
    ;
         φ2 = φ
    ),
    assert(fact(ba,Head,φ2)),
    (flag → true ; assert(flag)).

get_fact(Type,Atom,Φ) ←
    functor(Atom,Name,A),
    functor(F_Atom,Name,A),
    Atom =.. [_ | A_Vs],
    F_Atom =.. [_ | F_Vs],
    fact(Type,F_Atom,φ),
    bdd_rename(φ,F_Vs,A_Vs,Φ),
    !.
```

Figure 1: Pos Interpreters for groundness (left); and backwards (right) analyses