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. This leads to a better understanding of termination inference, simplifies its formal justification, and facilitates implementation. We evaluate the application of this approach to enhance an existing 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* and we make the observation that the (missing) link between the two is a technique called *backwards analysis*.

Backwards analysis addresses 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. This ensures that when working backwards from an assertion in the program, at each step we can find a best approximation (weakest requirement) to eventually provide a condition on the inputs which guarantees that the assertion will hold.

To demonstrate this link between termination checking and termination inference, we apply backwards analysis as described by King and Lu [12] to enhance the termination (checking) analyzer described in [4] so that it will perform also termination inference. Our focus is on 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 [15–17] (with a tool accessible at `http://www.complang.tuwien.ac.at/cti`). Our observation is that the link with backwards analysis provides a straightforward justification and also leads to a better implementation design.

2 Preliminaries and Motivating Example

We assume a familiarity with the standard logic program terminology [13, 1] as with the basics of abstract interpretation [5, 6]. This section reviews the program analyses upon which we build in the rest of the paper. For notation, in brief: Variables in logic programs are denoted using the upper case as in Prolog; 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: $append([X|Xs], Ys, [X|Zs]) \leftarrow append(Xs, Ys, Zs)$. 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. For details on Pos see [14].

Size relations express linear information about the sizes of terms (with respect to a given norm function) [2, 3, 7, 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, $z = x + y$ 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 [2, 3, 7, 8]. 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 [4] although we could use as well almost any of the alternatives described in the literature. This analyzer is based on a bottom-up 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. Each element in the abstraction of this semantics 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 [4] involves two phases: First, the user provides a program and the analyzer approximates its loops over two domains: size relations and instantiation dependencies. Then, the user specifies the input modes of an initial goal and the analyzer performs a termination check. For termination inference, it is the first part of the first phase of the termination analysis which is useful. From the descriptions of the loops with size information we extract the intitial Boolean assertions from which backwards analysis then proceeds. 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 [4] reports a single loop (abstract binary clause):

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

indicating that subsequent calls to *append* 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 is 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: $terminates(append(x, y, z)) \leftarrow x \vee z$.

The second step: Consider the use of *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 $member(X, Xs)$ which guarantees the termination condition $A \vee Xs$ for $append(A, [X|B], Xs)$. 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 *member*/2 is: $terminates(member(x, y)) \leftarrow y$.

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 the first argument (A) is sufficiently instantiated. However, for the program to terminate both of its loops must terminate — also the one for *append*/3 in the call to $member(X, Ys)$. So, drawing on the result of the previous step, also the second argument for *subset*/2 must be instantiated. To sum up, we have $terminates(subset(x, y)) \leftarrow x \wedge y$.

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 $append(x, y, t)$ satisfies $(x \wedge y) \leftrightarrow t$ (intuitively: indicating the flow of data either from x and y to t or vice versa). So, instead of imposing on the clause head the conditions from both 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 which must be satisfied when the clause is called, and b_i is either an atom, or a unification operation.

The analysis associates pre- and post- instantiation conditions, specified in Pos, with the predicates of the program. The postcondition ψ_p for predicate p is the result of the standard instantiation dependency analysis. It reflects the instantiations generated by p. The precondition φ_p for p is a condition which guarantees the termination of calls to p. Preconditions are initialized to *true* (the top element in Pos) and 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 postconditions to infer new approximations for the preconditions.

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

Now consider a clause $h(\bar{x}) \leftarrow \mu \diamond b_1, \ldots, b_n$ with an assertion $\mu \in$ Pos. Denote the current approximation of the precondition for b_i by φ_i and its postcondition by ψ_i $(1 \leq i \leq n)$. Assume that the current approximation of the precondition for $h(\bar{x})$ is φ. Backwards analysis infers a new approximation φ' on $h(\bar{x})$ by consecutive application of the basic step described above. We start with $e_{n+1} = \mathit{true}$ 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.

To be precise, as Pos is not closed under universal quantification, projection of x from φ is defined as the largest element in Pos which implies $\forall_x.\varphi$. This is well-defined as the bottom element in Pos (*false*) is always a candidate.

Example 1. Consider the clause

```
subset(A,B) :- e0 A ◇ e1 A=[X|Xs], e2 B=Ys,
    e3 member(X,Ys), e4 subset(Xs,Ys) e5.
```

(the assertion A states that the first argument must be ground) and assume that the postconditions and current approximation of the preconditions are (respectively):

$$post = \left\{ \begin{array}{l} member(x,y) \leftarrow (y \rightarrow x) \\ subset(x,y) \leftarrow (y \rightarrow x) \end{array} \right\} \qquad pre = \left\{ \begin{array}{l} member(x,y) \leftarrow y \\ subset(x,y) \leftarrow x \end{array} \right\}.$$

The conditions $e_5, \ldots, e_0$ are obtained as follows:

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

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 $subset(x,y) \leftarrow x \wedge y$. Note that the preconditions for unifications in the clause body are $true$ and their postconditions, the usual groundness dependences.

Backwards analysis is formalized in [12] as the greatest fixed point of an operator over Pos. Our implementation is described in [9].

4 Termination Inference

Termination inference proceeds as follows: **(a)** apply the initial phase of the termination analysis described in [4] to obtain a set of abstract binary clauses which approximate the loops in the program (with size relations); **(b)** extract Boolean conditions on the instantiation of variables to guarantee that each of the identified loops can be executed only a finite number of times; and **(c)** apply backwards analysis as defined in [12] to infer modes for 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. The idea is that: (1) in each loop, at least one set of arguments I, the sum of the sizes of which decreases, should all be instantiated enough (so that is a disjunction); and (2) all of the loops must satisfy the previous point (so that is a conjunction).

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*
$\mu(p(\bar{x}) \leftarrow \pi, p(\bar{y})) = \bigvee \left\{ \bigwedge_{i \in I} x_i \;\middle|\; I \subseteq \{1, \ldots, n\},\ \pi \models (\sum\limits_{i \in I} x_i > \sum\limits_{i \in I} y_i\ ) \right\}$
2. *and the assertion for $p(\bar{x})$ is:*
$\mu(p(\bar{x})) = \bigwedge \left\{\ \mu(\ell) \;\middle|\; \ell = p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in [\![P]\!]^{bin}_{size}\ \right\}$

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 (in case their sum does). This simplistic approach works well in practice. A more elaborate approach is described in [18].

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 [4] are:

$$\begin{aligned}
\ell_1 = &\quad split(x_1, x_2, x_3) \leftarrow [y_1 < x_1, y_3 < x_2, x_3 = y_2], split(y_1, y_2, y_3). \\
\ell_2 = &\quad split(x_1, x_2, x_3) \leftarrow [y_1 < x_1, y_2 < x_2, y_3 < x_3], split(y_1, y_2, y_3). \\
\ell_3 = &\quad split(x_1, x_2, x_3) \leftarrow [y_1 < x_1, y_3 < x_2, y_2 < x_3], split(y_1, y_2, y_3).
\end{aligned}$$

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* $\boldsymbol{b}$ *('bound') or* $\boldsymbol{f}$ *('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 [4] 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 [4] it follows that the loop on q/k must terminate.

5 Experimental Results

We have implemented an analyzer for termination inference and it is accessible at `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`. Basically, we combine the first phase of the termination analysis described in [4] (which describes the programs loops using size information) with an implementation of the backwards analysis algorithm described in [12] (for the Pos domain).

To evaluate our analyzer, we use the same benchmarks as used in [16]. The experimentation is set up to use the same parameters (choice of norm and widenning steps) as reported 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 costs for: **Size**: approximating loops with size information; **Pos**: approximating answers with instantiation dependencies; **Ass**: computing initial instantiation assertions (from the size information); **BA**: backwards analysis; **Total**: total analysis time (including preprocessing - not itemized on its own). **cTI**: total analysis time using cTI (as reported in [16]).

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), Mesnard and Neumerkel report precision in terms of the percentage of the programs predicates for which some (non false) termination condition is inferred. We obtain the same percentages, except for the last three programs where a "$\oplus$" indicates that we infer more terminating predicates than does cTI and a "$\ominus$" vice-versa.

The results indicate that the precision of the two analyzers is quite similar and that (given that cTI is running on a faster machine) ours is in most cases more efficient. The interesting aspects of the experimental results come from a closer comparison with previous work: First, in comparison with termination checking as reported in [4], we observe that in most cases termination inference is not only more general but also faster than termination checking. Note that the two columns labelled by **Size**, and **Pos** correspond to tasks which are performed anyway during (the first phase of) termination checking. The next two columns (labelled **Ass** and **BA**) concern tasks specific to termination inference which are very fast and in general less expensive than the cost of the second phase (checking a given mode) of the analysis described in [4].

Second, in comparison with the termination inference described in [16], note that while the total analysis times are similar, 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]). This is due to the application of a carefully designed backwards analysis algorithm which propagates information (backwards) through the clause bodies. In contrast, the corresponding phase in cTI sets up the entire collection of recursive equations specifying the constraints that hold at each program point and then solves these simultaneously.

program	Size	Pos	Ass	BA	Total	cTI
permute	0.12	.00	.01	.00	0.14	0.15
duplicate	0.03	.00	.00	.00	0.03	0.05
sum1	0.05	.00	.01	.00	0.06	0.18
merge	0.19	.00	.02	.00	0.21	0.26
dis-con	0.09	.00	.00	.01	0.10	0.24
reverse	0.05	.00	.01	.00	0.08	0.08
append	0.05	.01	.00	.00	0.06	0.09
list	0.01	.01	.00	.00	0.03	0.01
fold	0.05	.00	.01	.00	0.06	0.10
lte	0.06	.01	.00	.00	0.07	0.13
map	0.05	.00	.00	.00	0.05	0.09
member	0.04	.01	.00	.00	0.05	0.03
msort	0.44	.00	.02	.00	0.46	0.43
msort$\star$	0.98	.01	.01	.01	1.02	0.57
msort_ap	0.63	.00	.04	.00	0.67	0.79
msort_ap$\star$	1.29	.03	.03	.00	1.35	0.92
naive_rev	0.08	.01	.00	.00	0.10	0.12
ordered	0.03	.00	.00	.00	0.03	0.04
overlap	0.05	.01	.00	.00	0.06	0.05
permute	0.10	.00	.01	.00	0.13	0.15
quicksort	0.38	.01	.04	.00	0.43	0.39
sum2	0.08	.00	.01	.00	0.09	0.08
select	0.09	.01	.00	.00	0.10	0.09
subset	0.09	.01	.00	.00	0.11	0.12
ann	4.46	.07	.30	.03	5.02	5.01
bid	0.62	.02	.05	.01	0.74	0.79
boyer	2.55	.05	.04	.01	2.75	3.53
browse	0.96	.01	.15	.00	1.16	1.81
credit	0.43	.02	.04	.01	0.54	0.61
peep	4.46	.04	.07	.02	4.68	12.08
plan	1.03	.02	.03	.01	1.12	0.71
qplan	10.86	.05	.51	.03	11.58	7.30
rdtok $\oplus$	2.86	.02	.16	.01	3.10	2.92
read $\ominus$	4.43	.03	.04	.03	4.65	6.87
warplan $\oplus$	2.54	.04	.14	.03	2.83	3.18

Table 1. Experimental Results

6 Related Work

Backwards reasoning for imperative programs dates back to the early days of 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 (for instance when collecting and testing that benchmark programs for program analysis actually work) they typically start from builtins and work backwards to infer the intended modes of use for the program.

In fact, our work can also be applied to the same task as it is natural to assume that the intended mode of use results in terminating computations. So for example where King and Lu infer the mode $x \vee y$ for the predicate $permutation\_sort(x, y)$, termination inference gives 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. The conjunction of the two (as the intended mode of use should be both terminating and not lead to instantiation errors) gives the mode $x_1 \wedge x_2$ (which is the intended mode of use). It is interesting to note that the two backwards analyses can be performed together. We simply start from the conjunction of the initial assertions from the two applications.

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. The main difference is in the design. Our approach is "black box", combining existing components from a standard termination checker and a generic backwards analysis technique. There are two main technical differences between the analysers: (1) when inferring the initial Boolean assertions from the results of the size analysis, cTI uses a more sophisticated, albeit more costly, technique (to detect level mappings) adapted from [18]. In practice, for the benchmark collection this makes almost no difference for precision but it is in general more powerful; and (2) In the phase where we apply backwards analysis implemented as a simple Prolog meta-interpreter, cTI invokes a μ-calculus solver to compute the greatest fixed point of a system of equations which seems more complex (though equivalent) to what backwards analysis is solving.

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 [4] to perform also termination inference.

Acknowledgement: We acknowledge the many discussions, as well as the exchange of code and benchmarks, with Andy King, Fred Mesnard and Cohavit Taboch.

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. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In *Sixth International Workshop on Logic Program Synthesis and Transformation (LOP-STR'96)*, pages 204–223, 1996.
3. 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.
4. 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.
5. 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.
6. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.
7. 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.
8. D. De Schreye and K. Verschaetse. Deriving linear size relations for logic programs by abstract interpretation. *New Generation Computing*, 13(02):117–154, 1995.
9. S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. Technical report, Ben-Gurion University of the Negev, 2001. `http://www.cs.bgu.ac.il/~mcodish/Papers`.
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. 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.