cTI: Bottom-Up Termination Inference for Logic Programs

S. Burckel[1], S. Hoarau[1], F. Mesnard[1], and U. Neumerkel[2]

[1] IREMIA - Université de La Réunion, FRANCE
{burckel,seb,fred}@univ-reunion.fr
[2] Institut für Computersprachen - Technische Universität Wien, AUSTRIA
ulrich@mips.complang.tuwien.ac.at

Abstract. We present cTI, a system for bottom-up termination inference. Termination inference is a generalization of termination analysis/checking. Traditionally, a termination analyzer tries to prove that a given class of queries terminates. This class must be provided to the system, requiring user annotations. With termination inference such annotations are not necessary. Instead, all provably terminating classes to all related predicates are inferred at once. The architecture of cTI is discussed, highlighting several new aspects to termination analysis. The notion of termination neutral arguments is introduced, which helps to narrow down the actual arguments responsible for termination in a norm independent manner. We show how our approach can be adopted to realize an incremental system able to reuse previously inferred results, thereby allowing to use the system within a programming environment. Further we show how termination inference serves to tackle generalizations of the usual notion of termination.

1 Introduction

Termination is a crucial aspect of program verification. For logic programs, the problem is of particular importance because there is *a priori* no syntactic restriction on queries. Termination has been the subject of many works in the last fifteen years in the logic programming community [21, 2, 19]. The research efforts can be divided in two groups (a survey is given in[11]): characterizing termination (e.g. [1]) and weakening such undecidable criteria to get decidable sufficient conditions (e.g. [22]) that lead to actual implementations. Our approach belongs to the latter stream. Our main innovation compared to other works of termination analysis (e.g. [22, 15, 7, 3]) is to provide a *generic* framework where we can *infer* sufficient universal termination conditions from the text of any Prolog program.

- Genericity stems from the fact that we can instantiate our system to various classes of termination. In this paper, we deal with left-termination and σ-termination [12].

– Inference means that we adopt a bottom-up approach to termination. There is no need to define a class of queries of interest. If required, such classes can be easily simulated within our framework. Moreover, the bottom-up nature can be exploited to realize an incremental system.

Currently the only requirement we impose on programs is that they must not create infinite rational terms. Hence we only consider NSTO programs [10, 9] that can be safely executed with any standard complying system or an execution with occurs check. Our system, called cTI, has been realized in SICStus Prolog and can be used via the WWW at http://www.complang.tuwien.ac.at/cti.
Contents. We organize the paper as follows. The next section gives an informal overview of the basic inference process of cTI. We then discuss three particular improvements: termination neutral arguments, norms, incrementality to support interactive environments, and an improved scheme for level mappings. Thereafter we presents a generalization for σ-termination.

2 cTI: a constraint-based left termination inference tool

Syntactic informations are often too weak to reason about non-trivial programs. Some semantics informations is required. For this reason our analyzer uses three main constraint structures [14]: Herbrand terms (CLP($\mathcal{H}$)) for the initial program P, non-negative integers (CLP($\mathbb{N}$)) and booleans (CLP($\mathcal{B}$)) for approximating P. The correspondence between these structures relies on *approximations* [16], which are a simple form [12] of abstract interpretation [5, 6]. We present our method to infer termination conditions by using the predicates app/3, app3/4, and nrev/3. A formal justification can be found in [16].

```
app([], Xs, Xs).
app([X|Xs], Ys, [X|Zs]) ←
    app(Xs, Ys, Zs).
```

```
nrev([], []).
nrev([X|Xs], Ys) ←
    nrev(Xs, Zs),
    app(Zs, [X], Ys).
```

```
app3(Xs, Ys, Zs, Us) ←
    app(Xs, Ys, Vs),
    app(Vs, Zs, Us).
```

1. The initial Prolog program P is mapped to $P^{\mathbb{N}}$, a program in CLP($\mathbb{N}$) using an approximation based on a symbolic norm. In our example, we use the term-size norm defined in section 3.2. All non-monotonic elements of the program are approximated by monotone constructs. E.g., Prolog's unsound negation \+X is approximated by X;true. It is maintained that if a goal in $P^{\mathbb{N}}$ is terminating, then also the corresponding goals in P terminate.

```
app_N(0, Xs, Xs).
app_N(1+X+Xs,Ys,1+X+Zs) ←
    app_N(Xs, Ys, Zs).
```

```
nrev_N(0, 0).
nrev_N(1+X+Xs,Ys) ←
    nrev_N(Xs, Zs),
    app_N(Zs, 1+X, Ys).
```

```
app3_N/4
same as
app3/4
```

2. In $\mathbb{N}$ we compute a model of all predicates. The model describes with a finite conjunction of linear equalities and inequalities the relations between the arguments of a goal (inter-argument relations $\mathbb{R}$) that hold for every solution. The actual computation is performed with CLP($\mathbb{Q}$). In our example we are able to determine the least model. In general, however, only a less precise

model is determined. For recursive predicates, the actual source of potential non-termination, we compute linear strictly decreasing level mappings called μ's. For instance, the meaning of $\mu^{\mathbb{N}}_{\text{app}}$ is: for each recursive call to app/3, the first and the third argument decrease.

	(least) models	level mappings
$\text{IR}^{\mathbb{N}}_{\text{app}}(x,y,z)$	$\equiv z = x + y$	$\mu^{\mathbb{N}}_{\text{app}}{}^{1}(x,y,z) \equiv x$
		$\mu^{\mathbb{N}}_{\text{app}}{}^{2}(x,y,z) \equiv z$
$\text{IR}^{\mathbb{N}}_{\text{nrev}}(x,y)$	$\equiv x = y$	$\mu^{\mathbb{N}}_{\text{nrev}}(x,y) \equiv x$
$\text{IR}^{\mathbb{N}}_{\text{app3}}(x,y,z,u)$	$\equiv u = x + y + z$	—

3. $P^{\mathbb{N}}$ is mapped to $P^{\mathcal{B}}$, a program in CLP($\mathcal{B}$). Here 1 means that an argument is bound w.r.t. the considered norm. Note that the obtained program no longer maintains the same termination property. Its sole purpose is to determine the actual dependencies of boundedness within the program. The simplified structure permits us to compute always the least model. In a single boolean term all previously computed linear level mappings are represented.

$\text{app}_{\mathcal{B}}$(1, Xs, Xs).
$\text{app}_{\mathcal{B}}$(1∧X∧Xs,Ys,1∧X∧Zs) ←
  $\text{app}_{\mathcal{B}}$(Xs, Ys, Zs).
	$\text{nrev}_{\mathcal{B}}$(1, 1).
	$\text{nrev}_{\mathcal{B}}$(1∧X∧Xs,Ys) ←
	  $\text{nrev}_{\mathcal{B}}$(Xs, Zs),
	  $\text{app}_{\mathcal{B}}$(Zs, 1∧X, Ys).
		$\text{app3}_{\mathcal{B}}/4$
		same as
		app3/4

	least models	level mappings
$\text{IR}^{\mathcal{B}}_{\text{app}}(x,y,z)$	$\equiv (x \wedge y) \Leftrightarrow z$	$\mu^{\mathcal{B}}_{\text{app}}(x,y,z) \equiv x \vee z$
$\text{IR}^{\mathcal{B}}_{\text{nrev}}(x,y)$	$\equiv x \Leftrightarrow y$	$\mu^{\mathcal{B}}_{\text{nrev}}(x,y) \equiv x$
$\text{IR}^{\mathcal{B}}_{\text{app3}}(x,y,z,u)$	$\equiv (x \wedge y \wedge z) \Leftrightarrow u$	—

The meaning of the boolean model is: when any proof of a call app(X,Y,Z) terminates, then Z is finite iff A and B are finite and, for nrev/2, when any proof of a call nrev(X,Y) terminates, X is finite iff Y is finite w.r.t. the used norm. Now, if we know (i.e. using level-mappings) that a call to app/3 terminates, then there is a finite number of successes and we may ensure the third argument is *bound* iff the two first arguments are *bound* w.r.t. the norm.

4. Finally, using all previously determined informations, $P^{\mathcal{B}}$ is translated into a system of boolean formulæ(see theorem 1) that ensures the propagation of the decreasing conditions of the level mappings through the call graph. The resolution of this system (computation of its greatest fixpoint by means of a boolean μ-solver [4]) gives, for each predicate symbol, a boolean term called a *termination condition*.

$$
\begin{array}{ll}
Pre_{\text{app}}(x,y,z) & \equiv x \vee z \\
Pre_{\text{nrev}}(x,y) & \equiv x \\
Pre_{\text{app3}}(x,y,z,u) & \equiv (x \wedge y) \vee (x \wedge u)
\end{array}
$$

So any call ← C, app(X,Y,Z), where C is a CLP($\mathcal{H}$) constraint, left-terminates if X or Z are *ground* in C etc.

The main result on which cTI is based on is the following Theorem 1, which ensures correctness of the left-termination condition.

Theorem 1. *Let P be a program, p and q be two predicate symbols of P. Assume that p is defined by m_p rules r_k: $p(\tilde{x}) \leftarrow c_k,\ p_{k,1}(\tilde{x}_{k,1}), \ldots\ p_{k,n_k}(\tilde{x}_{k,n_k})$ and for each $q \notin \bar{p}$ and appearing in the rules defining $\bar{p}$, a left-termination condition Pre_q has been computed. If the set of boolean terms $\{Pre_p\}_{p\in\bar{p}}$ verifies:*

$$\forall p \in \bar{p} \begin{cases} Pre_p(\tilde{x}) \rightarrow_{\mathcal{B}} \mu_p^{\mathcal{B}}(\tilde{x}), \\ \big[\forall 1 \le k \le m_p,\ \forall 1 \le j \le n_k, \\ \qquad \Big(Pre_p(\tilde{x}) \wedge c_k^{\mathcal{B}} \wedge \bigwedge_{i=1}^{j-1} Post_{p_{k,i}}(\tilde{x}_{k,i})\Big) \rightarrow_{\mathcal{B}} Pre_{p_{k,j}}(\tilde{x}_{k,j})\big] \end{cases}$$

then $\{Pre_p\}_{p\in\bar{p}}$ is a left-termination condition for $\bar{p}$.

2.1 Running cTI

Table 5 in the appendix presents timings of cTI using some standard benchmarks[1]. The columns indicate the time for TNA analysis, the model $\mathrm{IR}_{\mathbb{N}}$, the determination of the level mappings $\mu^{\mathbb{N}}$, the left termination condition TC_{l} and additionally the σ termination condition TC_{σ} (Sect. 4.2), which is not included in the total run time. The runtime for other phases is negligible. More information about our current implementation can be obtained by using cTI via the WWW-interface featuring also a large set of programs.

3 Recent progress

In the following sections we highlight some particular aspects of the current implementation. Prior to the actual analysis, we determine termination neutral arguments rapidly. Then we consider norms, incrementality and an extended scheme for level mappings.

3.1 Inferring termination neutral arguments

Predicates frequently contain arguments that have no influence on universal left termination. Differences are a frequent pattern with this property. In most uses of differences (e.g. list differences), one argument of the difference is termination neutral.

Definition 1 (Termination neutral argument (TNA)). *Let G be a goal $\leftarrow p(\ldots, x_i, \ldots)$, where the i-th argument is a variable x_i that occurs only at this position and let θ be a substitution s.t. DOM(θ) = x_i. The argument i is a termination neutral argument (TNA), if*

$$\forall \theta : G\theta \text{ terminates} \Leftrightarrow G \text{ terminates}$$

[1] collected by Naomi Lindenstrauss, www.cs.huji.ac.il/~naomil and also available at www.complang.tuwien.ac.at/cti/bench.

Example 1. The second argument of app/3, the third of app3/4, and the second of nrev/2 are termination neutral.

Since the TNA property is independent of a particular norm, the analysis can be performed prior to any other analysis. The information is currently used to accelerate the search for level mappings.

To determine some termination neutral arguments we represent each argument of a predicate with a boolean 0-1 variable where 1 means termination neutral and 0 means that the argument may influence termination. Constraints are imposed that tell which arguments may influence termination. The actual result is computed by labeling for a maximal satisfying assignment.

We consider a clause (with nontrivial head unifications explicit) to consist of the following possibly empty sequences of goals $S_1 \circ S_2 \circ S_3 \circ S_4$. S_1 contains any goals, S_2 is a possibly nonterminating goal, S_3 are always terminating goals, S_4 is an always failing goal followed by any other goals. S_3 and S_4 are ignored in the analysis. Head variables that occur in S_1 may influence termination. All these variables are set to 0. A head variable H that first occurs in S_2 may influence termination only if the argument B where it occurs in is not termination neutral. The constraint $\neg B \Rightarrow \neg H$ ensures that if B is false (may influence termination), then H is false, too. The following predicate shows all constraints imposed for our three predicates. A labeling process finds the desired answer.

tnaargs([app(A_1, A_2, A_3), app3(B_1, B_2, B_3, B_4), nrev(C_1, C_2)]) $\leftarrow$
% app/3 clause 1: $[] \circ [] \circ [A_1 = [], A_2 = A_3] \circ []$
% app/3 clause 2:
% $[A_1 = [X|Xs], A_3 = [X|Zs]] \circ [\text{app}(Xs, A_2, Zs)] \circ [] \circ []$
$A_1 = 0, \neg A_2 \Rightarrow \neg A_2, A_3 = 0,$
% app3/4:
% $[\text{app}(B_1, B_2, Vs)] \circ [\text{app}(Vs, B_3, B_4)] \circ [] \circ []$
$B_1 = 0, B_2 = 0, \neg A_2 \Rightarrow \neg B_3, \neg A_3 \Rightarrow \neg A_4,$
% nrev/2 clause 1: $[] \circ [] \circ [C_1 = [], C_2 = []] \circ []$
% nrev/2 clause 2:
% $[C_1 = [X|Xs], \text{nrev}(Xs, Zs)] \circ [\text{app}(Zs, [X], C_2)] \circ [] \circ []$ % not optimal!
% $[C_1 = [X|Xs]] \circ [\text{nrev}(Xs, Zs)] \circ [\text{app}(Zs, [X], C_2)] \circ []$ % ideal
$C_1 = 0, \neg A_3 \Rightarrow \neg C_2$

For app/3 and app3/4 the results are optimal. However, in nrev/2 we are not able to infer that the second argument is termination neutral. While the second goal app/3 is in fact always terminating —cTI is also able to prove this— we do not have this information right at hand at this point in time prior to the actual termination inference. In fact, the primary purpose of TNA analysis within cTI is to speed up the subsequent inference process, sacrificing precision for speed. The analysis performed in this manner is typically faster than parsing the corresponding program text with read/1 using SICStus' tokenizer written in C. It has never been observed[2] to take longer than parsing the program twice.

[2] cTI's WWW interface provides an option "TNA only" to perform this comparison

3.2 Using better norms

By default cTI uses the following symbolic norm term-size. Note that the predicate's arity is not taken into account to keep the values as small as possible. E.g.$||f(0,0)||_{\text{Term}-\text{Size}} = 1$.

$$||t||_{\text{Term}-\text{Size}} = \begin{cases} 1 + \sum_{i=1}^{n} ||t_i||_{\text{Term}-\text{Size}} & \text{if } t = f(t_1, \ldots, t_n), n > 0 \\ 0 & \text{if } t \text{ is a constant} \\ t & \text{if } t \text{ is a variable} \end{cases}$$

The term-size norm is one of the strongest norm we can imagine. A bound term in CLP($\mathbb{N}$) corresponds to a completely instantiated term in CLP($\mathcal{H}$). Therefore all arguments of compound terms have to be known to verify termination conditions. Many programs, however, terminate also for less instantiated terms. For example the predicate nrev/2 terminates also for a goal like nrev([A,B],Z). Here, the first argument is not bound w.r.t. term-size and thus no left-termination condition can be inferred. Since the list elements in the first argument are termination neutral, it is desirable to ignore them. In this case termination can be proved, provided that the list elements are ignored within the norm. We consider to add new norms both automatically and manually.

Type informations may help to hide some irrelevant parts of a term when switching to CLP($\mathbb{N}$). Trying to use types to infer norms [8] is an appealing idea but introduces several difficulties. A solution based on typed norms is presented in [7]. We use another approach presented in [12] which is currently being implemented. For example, with the code of app/3 predicate, our system can infer that the first argument is a list of unknown type elements and then we can use the list-length norm.

Alternatively norms can be added manually, by specifying a term mapping. Here, some terms are rewritten to some other terms prior to the inference. With the directive ← cti:norm_tmap([[_|L] = l(L)]). we specify a norm similar to the classical list-length. All occurrences of the list constructor are replaced by the structure l/1. Notice that termination in the new program implies termination in the original one, since all non-monotone constructs and built-ins have been ignored. The above term mapping corresponds effectively to the following norm:

$$||t||_{\text{List}-\text{Length}} = \begin{cases} 1 + ||t_2||_{\text{List}-\text{Length}} & \text{if } t = [t_1|t_2] \\ 1 + \sum_{i=1}^{n} ||t_i||_{\text{List}-\text{Length}} & \text{if } t = f(t_1, \ldots, t_n), n > 0 \\ 0 & \text{if } t \text{ is a constant} \\ t & \text{if } t \text{ is a variable} \end{cases}$$

This norm slightly differs from the usual definition of list-length. Usually all known terms different from the list constructor are mapped to 0. With this norm we obtain the desired results for many programs that use both lists and other structures. In a similar manner also most weighted norms can be realized, as they are required for ASSOCIATIVE, shown on cTI's WWW page.

3.3 Incrementality

Most current systems for termination analysis require considerable computing resources. For instance, cTI requires two fixpoint computations in CLP($\mathcal{B}$) and one in CLP($\mathbb{Q}$). However, we believe that it is very helpful to provide termination information during program development, not only "on demand", when some delay to infer the conditions is acceptable, but also implicitly. Our actual goal is to adapt cTI such that all termination conditions can be maintained after each modification of a program to support the programming environment GUPU[17] used for introductory Prolog courses at our universities. To this purpose we have split the original design that comprised monolithic phases. For each predicate group (containing all predicates within a mutual recursive cycle) we maintain separately a slightly canonized version of $P^{\mathbb{N}}$, the actual results described earlier and the dependencies between them. When a modification takes place, we first check if also the version in $P^{\mathbb{N}}$ has changed. If it has, termination inference is restarted for this predicate group. If the results match those previously obtained, no further treatment is necessary. Otherwise only those predicates that depend on the current predicate have to be reanalyzed. Since (at least in programming courses) one typically writes new predicates reusing previously written ones, and since each predicate group typically contains only a single predicate, the recalculation time is kept relatively small. Further, the inference process can be aborted at any time if it still takes too long, losing only the information of the currently analyzed predicate group. All previously computed results are still valid. The split in cTI's design might also be helpful to parallelize the inference process where we could split the tasks on a predicate group basis leading to a rather coarse grained parallelization that could exploit standard hardware in a network.

3.4 Extended level mappings

The determination of decreasing level mappings is based on an original extension of a linear programming technique described in [20]. While our current level mappings are strong enough to infer most desired termination conditions, there are some cases where they turned out to be too weak, e.g. the ACK program encoding the Ackermann function, where the desired termination conditions were not inferred.

We now define an extension of valuations that enables to prove termination for a strictly larger class of programs that contains for example the ACK program. We are going to construct the proof of termination of the following CLP($\mathbb{N}$) program that contains enough ideas for possible generalizations [that we indicate in brackets].

$$p(Y+1,0,Y). \qquad (R_1)$$

$$p(Z,X+1,0) \leftarrow \quad p(Z,X,1). \qquad (R_2)$$

$$p(Z,X+1,Y+1) \leftarrow \quad p(T,X+1,Y), p(Z,X,T). \qquad (R_3)$$

$$q(X) \leftarrow \quad 2*X+1 \leq Y, p(Y,X,X). \qquad (R_4)$$

For any $(z,x,y) \in \mathbb{N}^3$, $p(z,x,y)$ holds if and only if $z = Ackermann(x,y)$. For any $x \in \mathbb{N}$, $q(x)$ holds if and only if $Ackermann(x,x) \geq 2x+1$ holds. Our aim is to construct two matrices [one per predicate symbol] P and Q in $\mathcal{M}_{n,n}(\mathbb{N})$ with $n = 3$ [n is the maximal arity of predicates] and two vectors P', Q' in $\mathbb{N}^3$ such that, for any rule (with a possible constraint C), any vector of positive integers (satisfying C) has an image that decreases by applying the linear mapping of the corresponding predicate. More precisely, our aim is to find P, P', Q, Q' and a well ordering $\gg$ on $\mathbb{N}^3$ such that the following sufficient conditions for the termination hold for any $(x,y,z,t) \in \mathbb{N}^4$:

$$P.(z,x+1,0) + P' \gg P.(z,x,1) + P' \qquad (C_{2,1})$$
$$P.(z,x+1,y+1) + P' \gg P.(t,x+1,y) + P' \qquad (C_{3,1})$$
$$P.(z,x+1,y+1) + P' \gg P.(z,x,t) + P' \qquad (C_{3,2})$$
$$Q.(x,0,0) + Q' \gg P.(y,x,x) + P' \text{ when } 2x+1 \leq y \qquad (C_{4,1})$$

Observe that the rule (R_1) gives no condition, (R_3) gives two conditions, and we normalize the dimensions in $(C_{4,1})$ [it could be interesting to consider other embeddings]. Let $\gg$ be the lexicographical ordering :

$$(X_1,X_2,X_3) \gg (Y_1,Y_2,Y_3) \iff \begin{cases} X_1 > Y_1 & \text{or} \\ X_1 = Y_1, X_2 > Y_2 & \text{or} \\ X_1 = Y_1, X_2 = Y_2, X_3 > X_3 \end{cases}$$

The key idea is that $\gg$ is decidable and can be defined with linear programming assumptions: $(X_1,X_2,X_3) \gg (Y_1,Y_2,Y_3)$ (under some possible linear condition $\mathcal{C}$) is equivalent to the following:

$$\begin{cases} \{\mathcal{C}, X_1 < Y_1\} & \text{is not feasible in } \mathbb{N} \text{ and} \\ \{\mathcal{C}, X_1 = Y_1, X_2 < Y_2\} & \text{is not feasible in } \mathbb{N} \text{ and} \\ \{\mathcal{C}, X_1 = Y_1, X_2 = Y_2, X_3 \leq Y_3\} & \text{is not feasible in } \mathbb{N} \end{cases}$$

In the example, $(C_{4,1})$ is equivalent to that the following three linear systems are not feasible in $\mathbb{N}$.

$$S^1_{4,1} = \begin{cases} 2x+1 \leq y \\ Q_{1,1}x + Q'_1 < P_{1,1}y + P_{1,2}x + P_{1,3}x + P'_1 \end{cases}$$

$$S^2_{4,1} = \begin{cases} 2x+1 \leq y \\ Q_{1,1}x + Q'_1 = P_{1,1}y + P_{1,2}x + P_{1,3}x + P'_1 \\ Q_{2,1}x + Q'_2 < P_{2,1}y + P_{2,2}x + P_{2,3}x + P'_2 \end{cases}$$

$$S^3_{4,1} = \begin{cases} 2x+1 \leq y \\ Q_{1,1}x + Q'_1 = P_{1,1}y + P_{1,2}x + P_{1,3}x + P'_1 \\ Q_{2,1}x + Q'_2 = P_{2,1}y + P_{2,2}x + P_{2,3}x + P'_2 \\ Q_{3,1}x + Q'_3 \leq P_{3,1}y + P_{3,2}x + P_{3,3}x + P'_3 \end{cases}$$

So, given P, P', Q, Q', we test the non feasibility of all the systems: $S^1_{2,1}, S^2_{2,1}, S^3_{2,1}$, $S^1_{3,1}, S^2_{3,1}, S^3_{3,1}, S^1_{3,2}, S^2_{3,2}, S^3_{3,2}, S^1_{4,1}, S^2_{4,1}, S^3_{4,1}$. In practice, we can first constraint

the coefficients to be in a finite set (for example $\{0,1\}$). This method gives a solution (in less than one second using MuPAD on a personal computer):

$$P = \begin{pmatrix} 0\,1\,0 \\ 0\,0\,1 \\ 0\,0\,0 \end{pmatrix}; \quad P' = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \quad Q = \begin{pmatrix} 1\,0\,0 \\ 0\,0\,0 \\ 0\,0\,0 \end{pmatrix}; \quad Q' = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

That proves the termination of the Ackermann function program. In general, one can consider, as it is currently done in cTI, the dual aspects of the linear systems and compute directly the coefficients of the matrices that satisfy all the conditions (if they exist).

4 Beyond left-termination inference

4.1 Explaining non-termination

In [18] we have proposed a slicing approach to explain non-terminating queries which is used within GUPU. The principal idea is to insert false goals into a program (failure-slice) to narrow down the actual reason for non-termination. With the help of cTI and its termination conditions, we can improve the process to rule out terminating (and therefore uninteresting) slices.

4.2 σ-termination

An important extension from strict left-termination analysis presented in section 2 concerns the introduction of permutations inside the body of the clauses (see [12, 13] for more details). The main idea consists in performing termination analysis for the class of all the permuted programs within a *single* analysis. The motivation came from this fact that we may have to deal with several modes of a predicate where each mode needs a specific version of the initial program. The versions often differ from one another by a permutation of some literals inside the body of some clauses. We call this notion of termination the (universal) σ-termination.

Definition 2. *A selection rule s is called local if s selects one of the last literals introduced in the current resolvent during the resolution process.*

Definition 3. *A program P with query Q σ-terminate if there exists a local selection rule s such that any derivation from P and Q via s is finite.*

And, of course we expect to infer the corresponding termination condition. To encode the permutations inside the body of the clauses, we introduce boolean variables.

Definition 4. *Let P be a program. For each rule of P: $p(\tilde{x}) \leftarrow c, \tilde{B}$ where the body $\tilde{B}$ contains the literals: $p_1, \ldots, p_n$ (but we do not know the order of them), we can associate a sequence of $n(n-1)/2$ boolean variables $(b_{i,j})_{1 \leq i < j \leq n}$, called*

order variables, with the following constraints that express transitivity of the order relation:

$$\forall 1 \leq i < j < k \leq n, \begin{cases} b_{i,j} \wedge b_{j,k} \rightarrow_{\mathcal{B}} b_{i,k}, & b_{j,k} \wedge \neg b_{i,k} \rightarrow_{\mathcal{B}} \neg b_{i,j}, \\ b_{i,k} \wedge \neg b_{j,k} \rightarrow_{\mathcal{B}} b_{i,j}, & \neg b_{i,k} \wedge b_{i,j} \rightarrow_{\mathcal{B}} \neg b_{j,k}, \\ \neg b_{i,j} \wedge b_{i,k} \rightarrow_{\mathcal{B}} b_{j,k}, & \neg b_{j,k} \wedge \neg b_{i,j} \rightarrow_{\mathcal{B}} \neg b_{i,k} \end{cases}$$

The semantics of these variables is $b_{i,j} = 1$ *if* p_i *appears before* p_j*, and equals* 0 *if* p_j *appears before* p_i *(and this for all* $1 \leq i < j \leq j$*).*

The boolean formula we construct and we have to solve is quite similar to the one seen in section 2. The difference is the introduction of the order variables:

Theorem 2. *Assume that for each* $q \notin \bar{p}$ *appearing in the rules defining* $\bar{p}$*, a* σ*-termination condition* Pre_q *has been computed. If the set of boolean terms* $\{Pre_p\}_{p \in \bar{p}}$ *verifies:*

$$\forall p \in \bar{p} \begin{cases} Pre_p(\tilde{x}) \rightarrow_{\mathcal{B}} \gamma_p(\tilde{x}), \\ \forall 1 \leq k \leq m_p \ \exists (b^k_{e,h})_{1 \leq e < h \leq j_k} \ \forall 1 \leq j \leq j_k \\ \Big(Pre_p(\tilde{x}) \wedge c^{\mathcal{B}}_{k,0} \wedge \bigwedge_{i=1}^{j-1} (\neg b^k_{i,j} \vee Post_{p_{k,i}}(\tilde{x}_{k,i})) \wedge \\ \qquad \bigwedge_{i=j+1}^{j_k} (b^k_{j,i} \vee Post_{p_{k,i}}(\tilde{x}_{k,i})) \Big) \rightarrow_{\mathcal{B}} Pre_{p_{k,j}}(\tilde{x}_{k,j}) \end{cases}$$

then $\{Pre_p\}_{p \in \bar{p}}$ *is a* σ*-termination condition for* $\bar{p}$*.*

Example 2. The boolean formula to solve for the predicate nrev/2 is:

$$\exists b, \big(Pre_{\text{nrev}}(x \wedge x_s, y_s) \wedge (b \vee Post_{\text{app}}(z_s, x, y_s)) \rightarrow_{\mathcal{B}} Pre_{\text{nrev}}(x_s, z_s)\big) \bigwedge \\ \big(Pre_{\text{nrev}}(X \wedge x_s, y_s) \wedge (\neg b \vee Post_{\text{nrev}}(x_s, z_s)) \rightarrow_{\mathcal{B}} Pre_{\text{app}}(z_s, x, y_s)\big)$$

assuming that $Pre_{\text{app}}(x, y, z)$ has already been computed and is equal to $x \vee z$, we find $Pre_{\text{nrev}}(x, y) = x \vee y$.

Performing σ-termination analysis increases the analysis time by a small fraction Evidently, it is significantly cheaper to integrate σ-termination analysis, than to analyze each possible permutation of the initial program separately.

5 Conclusion

We have illustrated the idea of bottom-up termination inference for logic programs, by a quick review of cTI, our implementation for universal left-termination, and two generalizations: non-termination and σ-termination. We believe that the concept of termination inference combined the current constraint technology may lead to smart semantics software tools for helping logic programmers.

References

1. K.R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.
2. K.R. Apt and D. Pedreschi. Studies in pure prolog: Termination. In J.W. Lloyd, editor, *Proc. of the Symp. in Computational Logic*, pages 150–176. Springer, 1990.
3. M. Codish and C. Taboch. A semantics basis for termination analysis of logic programs. *Journal of Logic Programming*, 41:103–123, 1999.
4. S. Colin, F. Mesnard, and A. Rauzy. Un module prolog de mu-calcul booléen : une réalisation par bdd. *Actes des JFPLC'99*, 1999.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximations of fixpoints. *Proc. of the 4th ACM Symp. on POPL*, pages 238–252, 1977.
6. P. Cousot and R. Cousot. Abstract interpretation and applications to logic programs. *J. Logic Programming*, 13:103–180, 1992.
7. S. Decorte. *Enhancing the power of termination analysis of logic programs through types and constraints.* PhD thesis, Katholieke Universiteit Leuven, 1997.
8. S. Decorte, F. Stefaan, and D. De Schreye. Automatic inference of norms : a missing link in automatic termination analysis. In *Proc. of ISLP'93*, pages 420–436, 1993.
9. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer, 1996.
10. P. Deransart, G. Ferrand, and M. Téguia. NSTO programs (not subject to occur-check). *Proc. of the Int. Logic Programming Symp.*, pages 533–547, 1991.
11. D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *J. Logic Programming*, 1993.
12. S. Hoarau. *Inférer et compiler la terminaison des programmes logiques avec contraintes.* PhD thesis, Université de La Réunion, 1999.
13. S. Hoarau and F. Mesnard. Inferring and compiling termination for constraint logic programs. In P. Flener, editor, *Proc. of the 8th Int. Workshop LOPSTR'98, LNCS 1559*, pages 240–254. Springer, 1998.
14. J. Jaffar and M.J. Maher. Constraint logic programming: a survey. *J. Logic Programming*, pages 503–581, 1994.
15. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. *Proc. of the 14th International Conference on logic Programmming*, pages 63–77, 1997.
16. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proc. of JICSLP'96*, pages 7–21. MIT Press, 1996.
17. U. Neumerkel. Gupu: A prolog course environment and its programming methodology. In M. Maher, editor, *Proc. of JICSLP'96*, page 549. MIT Press, 1996. http://www.complang.tuwien.ac.at/ulrich/gupu/.
18. U. Neumerkel and F. Mesnard. Localizing and explaining reasons for non-terminating logic programs with failure-slices. In G. Nadathur, editor, *Int. Conf. PPDP'99, LNCS 1702*, pages 328–341. Springer, 1999.
19. S. Ruggieri. *Verification and Validation of Logic Programs.* PhD thesis, Universitá di Pisa, 1999.
20. K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. *Proc. of PODS'91*, pages 216–226, 1991.
21. T. Vasak and J. Potter. Characterization of terminating logic programs. In *IEEE Symp. on Logic Programming*, pages 140–147. IEEE Computer Society, 1986.
22. K. Verschaetse and D. De Schreye. Deriving termination proofs for logic programs, using abstract procedures. *Proc. of the 8th ICLP*, pages 301–315, 1991.

Table 1. cTI 0.22 on Pentium II, 300 MHz, SICStus 3.8.2, 3.1 Mlips.

times in [ms] program	TNA	IR_{IN}	μ	$IR_{\mathcal{B}}$	TC_{I}	Total	TC_{σ}
DDS1.1	2	122	150	20	10	336	10
DDS3.13	2	57	97	12	7	195	6
DDS3.15	2	120	490	60	22	796	38
DDS3.17	5	75	492	20	7	646	10
DDS3.8	5	117	117	30	2	298	8
PL3.5.6	2	47	37	12	7	129	4
PL4.0.1	5	235	155	40	37	536	58
PL4.4.3	0	130	477	60	22	786	34
PL4.5.2	5	182	82	40	22	447	36
PL7.2.9	7	277	287	25	22	680	38
PL8.3.1	7	752	602	100	47	1700	86
PL8.4.1	0	57	237	10	2	321	2
PERMUTATION	7	215	335	90	32	756	84
SUM	5	112	150	7	7	308	8
LIST	0	25	30	7	2	79	2
NAIVE_REV	2	190	220	37	15	528	24
ORDERED	7	27	60	10	5	138	10
SELECT	0	62	142	20	7	267	10
CURRY_AP	10	632	667	112	67	1613	74
MAP	2	67	115	10	10	233	8
DERIV	10	527	2157	240	72	3336	146
OCCUR	2	327	537	92	67	1241	130
GRAMMAR	5	302	17	70	12	652	30
QUERY	2	140	10	75	20	531	76
FIB_T	0	167	332	20	15	596	26
MMATRIX	10	652	1152	175	92	2407	150
ACK	5	427	420	17	12	935	16
GAME	0	5	7	2	5	53	3
YALE_S_P	5	80	875	25	15	1069	10
ARIT_EXP	7	105	717	22	15	922	13
PQL	0	170	395	22	10	648	13
ASSOCIATIVE	2	77	200	20	10	377	20
QUEENS	5	280	515	70	50	1122	186
arith. mean	4	205	372	48	23	748	41
mean % of total	1%	27%	50%	6%	3%	-	-
max. % of total	5%	46%	82%	14%	9%	-	-