cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog

FRED MESNARD
Iremia, Université de La Réunion, Saint Denis, France
(*e-mail:* `fred@univ-reunion.fr`)

ROBERTO BAGNARA
Department of Mathematics, University of Parma, Italy
(*e-mail:* `bagnara@cs.unipr.it`)

Abstract

We present cTI, the first system for universal left-termination inference of logic programs. Termination inference generalizes termination analysis and checking. Traditionally, a termination analyzer tries to prove that a given class of queries terminates. This class must be provided to the system, for instance by means of user annotations. Moreover, the analysis must be redone every time the class of queries of interest is updated. Termination inference, in contrast, requires neither user annotations nor recomputation. In this approach, terminating classes for all predicates are inferred at once. We describe the architecture of cTI and report an extensive experimental evaluation of the system covering many classical examples from the logic programming termination literature and several Prolog programs of respectable size and complexity.

KEYWORDS: Termination Inference; Termination Analysis; Logic Programming; Abstract Interpretation.

1 Introduction

Termination is a crucial aspect of program verification. It is of particular importance for logic programs (Lloyd 1987; Apt 1997), since there are no *a priori* syntactic restrictions to queries and, as a matter of fact, most predicates programmers tend to write do not terminate for their most general queries. In the last fifteen years, termination has been the subject of several research works in the field of logic programming (see, for instance, (Francez et al. 1985; Apt and Pedreschi 1990; Ruggieri 1999)). In contrast to what happens for other programming paradigms, there are two notions of termination for logic programs (Vasak and Potter 1986): *existential* and *universal* termination. To illustrate them, assume we are using a standard Prolog engine. Existential termination of a query means that either the computation finitely fails or it produces *one* solution in finite time. This does not exclude the possibility that the engine, when asked for further solutions, will loop. On the other hand, universal termination means that the computation yields all solutions and eventually fails in finite time (if we repeatedly ask for further solutions).

Although the concept of existential termination plays an important role in connection with *normal* logic programs, it has severe drawbacks that make it not appropriate in other contexts: existential termination is not *instantiation-closed* (i.e., a goal may existentially terminate, yet some of its instances may not terminate), hence it is not *and-compositional* (i.e., two goals may existentially terminate while their conjunction does not); finally, existential termination depends on the textual order of clauses in the program. Universal termination is a stronger and much more robust concept: it implies existential termination and it is both and-compositional and instantiation-closed.

Existential termination has been the subject of only a few works (Vasak and Potter 1986; Levi and Scozzari 1995; Marchiori 1996) whereas most research focused on universal termination. There are two main directions (see (De Schreye and Decorte 1994) for a survey): characterizing termination (Apt and Pedreschi 1990; Apt and Pedreschi 1993; Ruggieri 1999) and finding weaker but decidable sufficient conditions that lead to actual algorithms, e.g., (Ullman and Van Gelder 1988; Plümer 1990; Verschaetse 1992). Even though our research belongs to both streams, in this paper we focus on an intuitive presentation of the implementation of our approach. A companion paper presents a complete formalization of our work in the theoretical setting of acceptability for constraint logic programs (Mesnard and Ruggieri 2003), where we refine a necessary and sufficient condition for termination to the sufficient condition implemented in cTI.

Our main contribution compared to other automated termination analyzers (Lindenstrauss and Sagiv 1997; Decorte 1997; Speirs et al. 1997; Codish and Taboch 1999) is that our tool *infers* sufficient universal termination conditions from the text of any Prolog program, adopting a bottom-up approach to termination. An important feature of this approach first presented in (Mesnard 1996) is that there is no need to define in advance a class of queries of interest. (If required, these classes can be provided after the analysis has finished in order to specialize the obtained results.) Our system, called *cTI* from *constraint-based Termination Inference*, is written in SICStus Prolog. A preliminary account of the work described in this paper appeared in (Mesnard and Neumerkel 2001), where we showed that numeric computations took most of the execution times. Now cTI relies on the specialized Parma Polyhedra Library (Bagnara et al. 2002), a modern C++ library for the manipulation of convex polyhedra that significantly speeds up the analysis. Moreover, cTI has been extended so that it can analyze any ISO-Prolog program (ISO/IEC 1995; Deransart et al. 1996). The only correctness requirement we currently impose on programs is that they must not create infinite rational terms. Hence we assume execution with occurs-check or, equivalently, NSTO programs (i.e., programs that are *Not Subject to Occur-Check* (Deransart et al. 1991) and thus are safely executed with any standard conforming system). We point out that simple, sufficient syntactic methods for ensuring occurs-check freedom are presented in (Apt and Pellegrini 1994) while (Søndergaard 1986; Crnogorac et al. 1996) describe abstract-interpretation based solutions. Recently, *finite-tree analysis* (Bagnara et al. 2001a;

Bagnara et al. 2001b) has been proposed to confine infinite rational terms in programs that are not occurs-check free. Both the approach described in (Mesnard and Ruggieri 2003) and the cTI system can be extended, with the help of finite-tree analysis, to deal also with such programs.

Throughout the paper we assume a basic knowledge of logic programming (see, e.g., (Apt 1997)), constraint logic programming (see, e.g., (Marriott and Stuckey 1998)), abstract interpretation (see, e.g., (Cousot and Cousot 1992)), and propositional μ-calculus (see, e.g., (Clarke et al. 2000)). In Section 2 we present cTI informally with an example analysis. How to use cTI is described in Section 3. An experimental evaluation of the system is the subject of Section 4. Related work is discussed in Section 5 while Section 6 concludes.

2 An Overview of cTI

Our aim is to compute classes of queries for which universal left termination is guaranteed. We call such classes *termination conditions*. More precisely, let P be a Prolog program and q a predicate symbol of P. A termination condition for q is a set TC_q of goals of the form $\leftarrow c, q(\tilde{x})$ where c is a CLP($\mathcal{H}$) constraint such that, for any goal $G \in \mathrm{TC}_q$, each derivation of P and G using the left-to-right selection rule is finite.

Our analyzer uses three main constraint structures: Herbrand terms for the initial program P (seen as a CLP($\mathcal{H}$) program), non-negative integers, and booleans (P is abstracted into both a CLP($\mathcal{N}$) and a CLP($\mathcal{B}$) program). We illustrate our method to infer termination conditions by means of an example. The method consists of six distinct steps, which will be illustrated on the following definition for the predicates `app/3`, `nrev/2` and `app3/4`.

```
app([], X, X).
app([E|X], Y, [E|Z]) :-
    app(X, Y, Z).
```

```
nrev([], []).
nrev([E|X], Y) :-
    nrev(X, Z),
    app(Z, [E], Y).
```

```
app3(X, Y, Z, U) :-
    app(X, Y, V),
    app(V, Z, U).
```

Step 1: From Prolog to CLP($\mathcal{N}$). From the Prolog program P, a CLP($\mathcal{N}$) program $P^{\mathcal{N}}$ is obtained by applying a symbolic norm. In our example, we use the *term-size* norm, which is the one cTI applies by default. All ISO-predefined predicates have been manually pre-analyzed for this norm. Notice that, as explained in (Mesnard and Ruggieri 2003), termination inference for pure Prolog programs can be based on any linear norm. The symbolic term-size norm is inductively defined as follows:

$$\|t\|_{\text{term-size}} \stackrel{\text{def}}{=} \begin{cases} 1 + \sum_{i=1}^{n} \|t_i\|_{\text{term-size}}, & \text{if } t = f(t_1, \dots, t_n) \text{ with } n > 0; \\ 0, & \text{if } t \text{ is a constant;} \\ t, & \text{if } t \text{ is a variable.} \end{cases}$$

For example, $\|f(0,0)\|_{\text{term-size}} = 1$. All non-monotonic elements of the program are approximated by monotone constructs. For instance, Prolog's unsound negation `\+ G` is approximated by `((G, false) ; true)`. More generally, extra-logical predicates are mapped to their first-order counterparts so that the termination property is preserved. For our running example, we obtain the following CLP($\mathcal{N}$) clauses:

$\mathrm{app}_{\mathcal{N}}(0, x, x).$	$\mathrm{nrev}_{\mathcal{N}}(0, 0).$	$\mathrm{app3}_{\mathcal{N}}(x, y, z, u) \leftarrow$
$\mathrm{app}_{\mathcal{N}}(1 + e + x, y, 1 + e + z) \leftarrow$	$\mathrm{nrev}_{\mathcal{N}}(1 + e + x, y) \leftarrow$	$\quad \mathrm{app}_{\mathcal{N}}(x, y, v),$
$\quad \mathrm{app}_{\mathcal{N}}(x, y, z).$	$\quad \mathrm{nrev}_{\mathcal{N}}(x, z),$	$\quad \mathrm{app}_{\mathcal{N}}(v, z, u).$
	$\quad \mathrm{app}_{\mathcal{N}}(z, 1 + e, y).$	

Step 2: Computing a numeric model. A model of the CLP($\mathcal{N}$) program is now computed. For each predicate p, the model describes, with a finite conjunction of linear equalities and inequalities denoted by $\mathrm{post}_p^{\mathcal{N}}$, the linear inter-argument relations that hold for every solution of p. In our example we obtain the following model:

$$\begin{aligned}
\mathrm{post}_{\mathrm{app}}^{\mathcal{N}}(x, y, z) &\iff x + y = z,\\
\mathrm{post}_{\mathrm{nrev}}^{\mathcal{N}}(x, y) &\iff x = y,\\
\mathrm{post}_{\mathrm{app3}}^{\mathcal{N}}(x, y, z, u) &\iff x + y + z = u.
\end{aligned}$$

The actual computation is performed on the set of nonnegative, infinite precision rational numbers, using a fixpoint calculator based on PPL, the Parma Polyhedra Library (Bagnara et al. 2002), and the standard widening (Cousot and Halbwachs 1978; Halbwachs 1979). In our example the least model is found. In general, however, only a less precise model can be determined.

Step 3: Computing a numeric level mapping. The information provided by the numerical model is crucial to compute a *level mapping* $|\cdot|^{\mathcal{N}}$. Let p be an n-ary predicate symbol in the CLP($\mathcal{N}$) program. The level mapping associates to p a function $f_p \colon \mathbb{N}^n \to \mathbb{N}$ that is guaranteed to decrease when going from the head of the clause to each recursive call(s), if any, for each clause defining p. For example, a level mapping $|\cdot|^{\mathcal{N}}$ such that $|\mathrm{nrev}_{\mathcal{N}}(x, y)|^{\mathcal{N}} = x$ intuitively means: for each ground instance[1] of each recursive clause defining $\mathrm{nrev}_{\mathcal{N}}$, the first argument decreases when going from the head of the clause to the recursive call (since $1 + e + x > x$ for each $e, x \in \mathbb{N}$). Since no clause defining $\mathrm{app3}_{\mathcal{N}}$ is recursive, the level mapping can be defined so that $|\mathrm{app3}(x, y, z, t)|^{\mathcal{N}} = 0$. The level mapping computed for our example is defined by:

$$\begin{aligned}
|\mathrm{app}(x, y, z)|^{\mathcal{N}} &= \min(x, z),\\
|\mathrm{nrev}(x, y)|^{\mathcal{N}} &= x,\\
|\mathrm{app3}(x, y, z, u)|^{\mathcal{N}} &= 0.
\end{aligned}$$

This is obtained by means of an improvement of the technique by K. Sohn and A. Van Gelder for the automatic generation of linear level mappings. Their algorithm, which is based on linear programming, is complete in the sense that it will

[1] That is, where natural numbers have replaced variable symbols.

always provide a linear level mapping if one exists (Sohn and Van Gelder 1991). Our extension, which is described in (Mesnard and Neumerkel 2001), consists in first computing a constraint over the coefficients of a generic linear level mapping (step 3a). Then we generate a concrete level mapping (step 3b). Notice that for a multi-directional predicate (such as `app/3`) we may get multiple linear level mappings. These are combined, with the min operator, into one non-linear level mapping.

In contrast with the well-known standard framework of acceptability, the decrease of the level mapping has to be shown only for predicates belonging to the same strongly connected component (SCC) of the call graph. Step 5 below will ensure that the other calls to predicates from lower SCC's do left terminate. The advantage of this approach is twofold: first, the computation of a level mapping, being SCC-based, is modular. Secondly, the expressive power of *linear* level mappings with respect to termination is much higher than in the acceptability case.

Step 4: From CLP($\mathcal{N}$) to CLP($\mathcal{B}$). From the CLP($\mathcal{N}$) program $P^{\mathcal{N}}$ a CLP($\mathcal{B}$) program, $P^{\mathcal{B}}$, is obtained by mapping each natural number to 1 (true), each variable symbol to itself, and addition to logical conjunction.

$$\begin{array}{l|l|l}
\mathrm{app}_{\mathcal{B}}(1, x, x). & \mathrm{nrev}_{\mathcal{B}}(1, 1). & \mathrm{app3}_{\mathcal{B}}(x, y, z, u) \leftarrow \\
\mathrm{app}_{\mathcal{B}}(1 \wedge e \wedge x, y, 1 \wedge e \wedge z) \leftarrow & \mathrm{nrev}_{\mathcal{B}}(1 \wedge e \wedge x, y) \leftarrow & \quad \mathrm{app}_{\mathcal{B}}(x, y, v), \\
\quad \mathrm{app}_{\mathcal{B}}(x, y, z). & \quad \mathrm{nrev}_{\mathcal{B}}(x, z), & \quad \mathrm{app}_{\mathcal{B}}(v, z, u). \\
 & \quad \mathrm{app}_{\mathcal{B}}(z, 1 \wedge e, y). &
\end{array}$$

The purpose of $P^{\mathcal{B}}$ is the one of capturing boundedness dependencies within $P^{\mathcal{N}}$ or, equivalently, rigidity dependencies within the original program.[2] A model for $P^{\mathcal{B}}$ is then computed and a boolean level mapping $|\cdot|^{\mathcal{B}}$ is obtained from the numerical level mapping computed in Step 3. In order to do that, the translation scheme outlined above is augmented with the association of the logical disjunction $x \vee y$ to $\min(x, y)$: this means that $\min(x, y)$ is a bounded quantity if x or y or both are bounded. Here is what we obtain for the example program:

$$\begin{array}{rcll}
\mathrm{post}^{\mathcal{B}}_{\mathrm{app}}(x, y, z) & \Longleftrightarrow & (x \wedge y) \leftrightarrow z, & |\mathrm{app}(x, y, z)|^{\mathcal{B}} = x \vee z, \\
\mathrm{post}^{\mathcal{B}}_{\mathrm{nrev}}(x, y) & \Longleftrightarrow & x \leftrightarrow y, & |\mathrm{nrev}(x, y)|^{\mathcal{B}} = x, \\
\mathrm{post}^{\mathcal{B}}_{\mathrm{app3}}(x, y, z, u) & \Longleftrightarrow & (x \wedge y \wedge z) \leftrightarrow u, & |\mathrm{app3}(x, y, z, u)|^{\mathcal{B}} = 1.
\end{array}$$

For instance, as we use the term-size norm, this model tells us that for any computed answer θ to a call $\mathtt{nrev}(x, y)$, $x\theta$ is ground if and only if $y\theta$ is ground.

Step 5: Computing boolean termination conditions. The information obtained from $P^{\mathcal{B}}$ for each program point is combined with the level mapping by means of the following boolean μ-calculus formulæ, whose solution gives the desired boolean termination conditions.

[2] A term t is *rigid* with respect to a symbolic norm $\|\cdot\|$ if and only if its measure is invariant by instantiation, i.e., $\|t\| = \|t\theta\|$ for any substitution θ.

$$
\begin{aligned}
\mathrm{pre}_{\mathrm{app}} &= \nu T \,.\, \lambda(x,y,z) \,. \\
&\begin{cases}
\big|\mathrm{app}(x,y,z)\big|^{\mathcal{B}} \\
\forall e, x', z' : \Big(\big(x \leftrightarrow (1 \wedge e \wedge x')\big) \wedge \big(z \leftrightarrow (1 \wedge e \wedge z')\big)\Big) \rightarrow T(x',y,z')
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{pre}_{\mathrm{nrev}} &= \nu T \,.\, \lambda(x,y) \,. \\
&\begin{cases}
|\mathrm{nrev}(x,y)|^{\mathcal{B}} & (1)\\
\forall e, x', z : \Big(\big(x \leftrightarrow (1 \wedge e \wedge x')\big)\Big) \rightarrow T(x',z) & (2)\\
\forall e, x', z : \Big(\big(x \leftrightarrow (1 \wedge e \wedge x')\big) \wedge \mathrm{post}^{\mathcal{B}}_{\mathrm{nrev}}(x',z)\Big) \rightarrow \mathrm{pre}_{\mathrm{app}}(z, 1 \wedge e, y) & (3)
\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{pre}_{\mathrm{app3}} &= \nu T \,.\, \lambda(x,y,z,u) \,. \\
&\begin{cases}
\big|\mathrm{app3}(x,y,z,u)\big|^{\mathcal{B}} \\
\forall v : 1 \rightarrow \mathrm{pre}_{\mathrm{app}}(x,y,v) \\
\forall v : \mathrm{post}^{\mathcal{B}}_{\mathrm{app}}(x,y,v) \rightarrow \mathrm{pre}_{\mathrm{app}}(v,z,u)
\end{cases}
\end{aligned}
$$

Here is the intuition behind such boolean μ-calculus formulæ. Consider the `nrev/2` predicate. Its unit clause is taken into account in the computation of the numeric and the boolean model. For computing the boolean termination condition $\mathrm{pre}_{\mathrm{nrev}}$, we consider the clause

$$\mathrm{nrev}_{\mathcal{B}}(x,y) \leftarrow [x \leftrightarrow (1 \wedge e \wedge x')], \mathrm{nrev}_{\mathcal{B}}(x',z), \mathrm{app}_{\mathcal{B}}(z, 1 \wedge e, y).$$

We are looking for a boolean relation $T(x,y)$ satisfying the following conditions:

- for each (x,y) in T, the level mapping has to be bounded, which leads to condition (1) above;
- the recursive call to `nrev/2` has to terminate, hence condition (2);
- for any state resulting from the evaluation of the first call, the subsequent call to `app/3` has to terminate, giving condition (3);
- finally, we are interested in the weakest solution for T, hence the boolean termination condition is defined as a greatest fixpoint:

$$\mathrm{pre}_{\mathrm{nrev}} = \nu T \,.\, \lambda(x,y) \,.\, \big\{(1) \wedge (2) \wedge (3)\big\}.$$

Solving the equations for our example gives:

$$
\begin{aligned}
\mathrm{pre}_{\mathrm{app}}(x,y,z) &= x \vee z, \\
\mathrm{pre}_{\mathrm{nrev}}(x,y) &= x, \\
\mathrm{pre}_{\mathrm{app3}}(x,y,z,u) &= (x \wedge y) \vee (x \wedge u).
\end{aligned}
$$

The greatest fixpoint is evaluated with the boolean μ-solver described in (Colin et al. 1997), which computes on the domain Pos of positive boolean formulæ (Armstrong et al. 1998) and is based on the boolean solver of SICStus Prolog.

Step 6: Back to Prolog. In the final step of the analysis, the boolean termination conditions are lifted to termination conditions with the following interpretation, where the c's are CLP($\mathcal{H}$) constraints:

- each goal '`?-` c, `app(X,Y,Z).`' left-terminates if `X` or `Z` are ground in c;
- each goal '`?-` c, `nrev(X,Y).`' left-terminates if `X` is ground in c;
- each goal '`?-` c, `app3(X,Y,Z,U).`' left-terminates if `X` and `Y` are ground in c or `X` and `U` are ground in c.

3 Using cTI

Once compiled and installed, cTI is invoked with the command '`cti source`', where the program in '`source`' is assumed to be an ISO-Prolog program. The user may then control the behavior of cTI with some options. We describe the main ones.

'`-p file`' By default, undefined predicates are assumed to fail. The user may enrich or redefine the set of built-ins recognized by the system, by specifying '`-p file`' on the command line. This has the effect of importing the predicates whose numerical model, boolean model, and termination condition are given in '`file`'. As predicates imported that way cannot be redefined in the analyzed program, this scheme provides a way to overcome potential weaknesses of the analysis.

'`-t timeout_in_ms`' The analysis steps 2, 3a, 3b, 4, and 5 described in Section 2 all include potentially expensive computations. Because of this, for each such step, the computation concerning each SCC is subject to a timeout, whose default value is 2 seconds. The '`-t`' option allows the user to modify this value.

'`-n N`' For the computation of the numeric model (step 2), a widening is used after n iterations of the approximate fixpoint iteration. The default value for n is 1.

The user may also modify a program to give specific information for selected program points. We illustrate this facility by means of examples; the precise syntax is given in the cTI's documentation. One may specify that particular program variables will only be bound to non-negative integers and that the analyzer should take into account some constraints involving them. For instance, cTI does not detect that the following program terminates:

```
p1(N) :- N > 0, M is N-1, p1(M).
p1(N) :- N > 1, A is N>>1, Z is N-A, p1(A), p1(Z).
```

where the predefined arithmetic functor '`>>/2`' is the bitwise arithmetic right shift. On the other hand, cTI is able to show that `p2(N)` terminates:

```
p2(N) :- cti:{N > 0, M = N-1}, p2(M).
p2(N) :- cti:{N > 1, 2*A =< N, N =< 2*A+1, Z = N-A}, p2(A), p2(Z).
```

Finally, at any program point, the user can add linear inter-argument relations or groundness relations that the analyzer will take for granted. The system can thus prove the termination of the goal '`?- top.`' where the predicate `top/0` is defined by the program given in Section 2 augmented with the following clause, where the term-size of `L1` is declared to be less than 10 and `L2` is declared to be ground:

```
top :- cti:{n(L1) < 10},app(L1,Zs,L2),cti:{b(L2)},app(Xs,Ys,L2).
```

While such programs are no longer ISO-Prolog programs, the annotations can be automatically removed so as to obtain the original programs back. The assertion language currently used in cTI is only experimental, and future versions of the system may be based on the language defined in (Hermenegildo et al. 2000).

4 Experimental Evaluation

Unless otherwise specified, the experiments we present here were all conducted with the option (see Section 3) `-p predef_for_compatibility.pl`, which ensures that non-ISO built-ins used in the benchmarks (several of which are written in a non-ISO dialect of Prolog) are predefined. This experimental evaluation was done on a GNU/Linux system with an Intel i686 CPU clocked at 2.4 GHz, 512 Mb of RAM, running the Linux kernel version 2.4, SICStus Prolog 3.10.0 (28.3 MLips), PPL version 0.5, and cTI version 1.0.

Standard programs from the termination literature. Table 1 presents timings and results of cTI on some standard LP termination benchmarks. The columns are labeled as follows:

program: the name of the analyzed program (the asterisk near a name means that we had to use one of the options that allow to tune the behavior of cTI);
top-level predicate: the predicate of interest;
checked: the class of queries *checked* by the analyzers of (Decorte et al. 1999; Lindenstrauss and Sagiv 1997; Speirs et al. 1997);
result: the best result among those reported in (Decorte et al. 1999; Lindenstrauss and Sagiv 1997; Speirs et al. 1997) (where, of course, '*yes, the program terminates*' is better than '*no, don't know*');
inferred: the termination condition *inferred* by cTI (1 means that any call to the predicate terminates, 0 means that cTI could not find a terminating mode for that predicate);
time: the running time, in seconds, for cTI to infer the termination conditions.

For all the examples presented in Table 1, our analyzer is able to infer a class of terminating queries at least as large than the one checked by the analyzers of (Decorte et al. 1999; Lindenstrauss and Sagiv 1997; Speirs et al. 1997) (although we manually tuned cTI three times). We point out that TermiLog (Lindenstrauss and Sagiv 1997) and TerminWeb (Codish and Taboch 1999) are sometimes able to prove termination whereas cTI is not and *vice versa*.

Standard programs from the abstract interpretation literature. Table 2 presents timings of cTI using some standard benchmarks[3] from the LP program analysis community. We have chosen eleven middle-sized, well-known logic programs. All the programs are taken from (Bueno et al. 1994) except `credit` and `plan`. The first column of Table 2 gives the name of the analyzed program and the second one

[3] These have been collected by N. Lindenstrauss, see `www.cs.huji.ac.il/~naomil`.

Table 1. *De Schreye's, Apt's, and Plümer's programs.*

		Others		cTI	
program	top-level predicate	checked	result	inferred	time (s)
`permute`	permute(x, y)	x	yes	x	0.03
`duplicate`	duplicate(x, y)	x	yes	$x \vee y$	0.02
`sum`	sum(x, y, z)	$x \wedge y$	yes	$x \vee y \vee z$	0.03
`merge`	merge(x, y, z)	$x \wedge y$	yes	$(x \wedge y) \vee z$	0.03
`dis-con`	dis(x)	x	yes	x	0.03
`reverse`	reverse(x, y, z)	$x \wedge z$	yes	x	0.02
`append`	append(x, y, z)	$x \wedge y$	yes	$x \vee z$	0.02
`list`	list(x)	x	yes	x	0.01
`fold`	fold(x, y, z)	$x \wedge y$	yes	y	0.02
`lte`	goal	1	yes	1	0.02
`map`	map(x, y)	x	yes	$x \vee y$	0.02
`member`	member(x, y)	y	yes	y	0.01
`mergesort`	mergesort(x, y)	x	no	0	0.06
`mergesort*`	mergesort(x, y)	x	no	x	0.07
`mergesort_ap`	mergesort_ap(x, y, z)	x	yes	z	0.11
`mergesort_ap*`	mergesort_ap(x, y, z)	x	yes	$x \vee z$	0.11
`naive_rev`	naive_rev(x, y)	x	yes	x	0.03
`ordered`	ordered(x)	x	yes	x	0.01
`overlap`	overlap(x, y)	$x \wedge y$	yes	$x \wedge y$	0.01
`permutation`	permutation(x, y)	x	yes	x	0.03
`quicksort`	quicksort(x, y)	x	yes	x	0.06
`select`	select(x, y, z)	y	yes	$y \vee z$	0.01
`subset`	subset(x, y)	$x \wedge y$	yes	$x \wedge y$	0.02
`sum`	sum(x, y, z)	z	yes	$y \vee z$	0.02
`pl2.3.1`	p(x, y)	x	no	0	0.01
`pl3.5.6`	p(x)	1	no	x	0.01
`pl3.5.6a`	p(x)	1	yes	x	0.01
`pl4.0.1`	append3(x,y,z,v)	$x \wedge y \wedge z$	yes	$(x \wedge y) \vee (x \wedge v)$	0.02
`pl4.5.2`	s(x, y)	x	no	0	0.03
`pl4.5.3a`	p(x)	x	no	0	0.01
`pl5.2.2`	turing(x, y, z, t)	$x \wedge y \wedge z$	no	0	0.11
`pl7.2.9`	mult(x, y, z)	$x \wedge y$	yes	$x \wedge y$	0.02
`pl7.6.2a`	reach(x, y, z)	$x \wedge y \wedge z$	no	0	0.02
`pl7.6.2b`	reach(x, y, z, t)	$x \wedge y \wedge z \wedge t$	no	0	0.03
`pl7.6.2c`	reach(x, y, z, t)	$x \wedge y \wedge z \wedge t$	yes	$z \wedge t$	0.04
`pl8.3.1`	minsort(x, y)	x	no	$x \wedge y$	0.04
`pl8.3.1a`	minsort(x, y)	x	yes	x	0.04
`pl8.4.1`	even(x)	x	yes	x	0.02
`pl8.4.2`	e(x, y)	x	yes	x	0.07

gives the number of its clauses (before any program transformation takes place). The following six columns indicate the running times (minimum execution times over ten runs), in seconds, for computing:

$M_P^{\mathcal{N}}$: a numeric model (step 2);
C_μ: the constraint over the coefficients of a generic linear level mapping (step 3a);
μ: the concrete level mapping (step 3b);
$M_P^{\mathcal{B}}$: a boolean model (step 4);

Table 2. *Running times for middle-sized programs.*

		analysis times (s)						
program	clauses	$M_P^{\mathcal{N}}$	C_μ	μ	$M_P^{\mathcal{B}}$	TC	total	Q%
`ann`	177	0.17	0.48	0.08	0.17	0.06	1.00	49%
`bid`	50	0.03	0.04	0.02	0.02	0.02	0.14	100%
`boyer`	136	0.07	0.06	0.02	0.08	0.02	0.30	85%
`browse`	30	0.05	0.12	0.03	0.04	0.01	0.26	60%
`credit`	57	0.02	0.03	0.02	0.02	0.01	0.11	100%
`peephole`	134	0.18	0.56	0.03	0.20	0.06	1.08	94%
`plan`	29	0.02	0.03	0.01	0.02	0.02	0.11	100%
`qplan`	148	0.20	0.52	0.12	0.18	0.07	1.13	68%
`rdtok`	55	0.13	0.39	0.03	0.07	0.02	0.65	44%
`read`	88	0.26	1.00	0.04	0.31	0.08	1.72	52%
`warplan`	101	0.10	0.25	0.01	0.08	0.02	0.49	33%
		18%	50%	6%	17%	6%	100%	
		average % of time for each analysis phase						

TC**:** the boolean termination conditions (step 5).

The next column reports the total runtime in seconds while the last column, labeled 'Q%', expresses the quality of the analysis, computed as the ratio of the number of user-defined predicates that have a non-empty termination condition over the total number of user-defined predicates (the result of an analysis presents all the user-defined predicates together with their corresponding termination conditions).

We note that cTI can prove that `bid`, `credit`, and `plan` are *left-terminating*: every ground atom left-terminates. For any such program P, T_P has only one fixpoint (Apt 1997, Theorem 8.13), which may help proving its partial correctness. Moreover, as the ground semantics of such a program is decidable, Prolog is its own decision procedure, which does help testing and validating the program.

On the other hand, when the quality of the analysis is less than 100%, it means that there exists at least one SCC where the inferred termination condition is 0. Let us call such SCC's *failed SCC's*. They are clearly identified, which may help the programmer. Here are some reasons why cTI may fail: potential non-termination, poor numeric model, non-existence of a linear level mapping for a predicate with respect to the model, inadequate norm. Also, the analysis of the SCC's which depend on a failed SCC is likely to fail, but this does not prevent cTI from analyzing other parts of the call graph.

Some larger programs. Finally, we have tested cTI on the following programs:

- `chat` is a parser written by F.C.N. Pereira and D.H.D. Warren;
- `lptp` is an interactive theorem prover for Prolog written by R. Stärk (Stärk 1998);
- `pl2wam` is the compiler from Prolog to WAM of GNU-Prolog 1.1.2 developed by D. Diaz (Diaz and Codognet 2000);

Table 3. *Running times for larger programs.*

program	clauses	analysis times (s) $M_P^{\mathcal{N}}$	C_μ	μ	$M_P^{\mathcal{B}}$	TC	total	timeouts	Q%
`chat`	515	3.89	2.78	2.84	2.85	0.35	12.80	1/1/1/0/0	71%
`lptp`	1298	3.99	14.10	1.84	2.88	1.65	25.10	0/1/0/0/0	67%
`pl2wam`	1190	2.12	2.37	0.99	2.00	1.29	9.22	0/0/0/0/0	64%
`slice`	952	2.20	10.46	0.13	2.08	0.93	16.20	0/3/0/0/0	55%
`symbolic1`	923	1.49	0.67	0.04	0.61	0.29	3.47	0/0/0/0/0	58%

- `slice` is a multi-language interpreter developed by R. Bagnara and A. Riaudo;
- `symbolic1` seems to be a simulator for a Prolog machine. We do not know the origin of this file.

The results of the analysis are given in Table 3. As explained in the previous section, we set up a timeout of 2 seconds per SCC for computing a CLP($\mathcal{N}$) model, the constraints defining level mappings, a CLP($\mathcal{B}$) model, and the termination condition. So we have a limit of 10 seconds of CPU time per SCC. The last but one column in Table 3 summarizes the number of timeouts for steps 2/3a/3b/4/5, respectively.

5 Related Work

The compiler of the Mercury programming language (Somogyi et al. 1996) includes a termination checker, described in (Speirs et al. 1997). The speed of the analyzer is quite impressive. We see two reasons for this. First, the termination checker is written in Mercury itself. Second, and most importantly, the analyzer takes high profit of the mode informations that are part of the text of the program being checked. On the other hand, while the running times of cTI are bigger, termination inference is a more general problem than termination checking: in the worst case, an exponential number of termination checks are needed to simulate termination inference.

TALP (Arts and Zantema 1996) is an automatic tool that transforms a well-moded logic program (see, e.g., (Apt 1997)) into a term rewriting system such that termination of the latter implies termination of the former. The generated term rewriting system is then proved terminating by the CiME tool (`http://cime.lri.fr/`). The system seems quite powerful for this class of logic programs.

(Genaim and Codish 2001) made recently a link between backward analysis (King and Lu 2002) and termination analysis, which leads to termination inference. Although they used a completely different scheme for computing level mappings, the results of the analysis on the programs described in Tables 1 and 2 were rather similar, both in time and quality, to previous versions of cTI that rely on the rational linear solver of SICStus Prolog. Thanks to the PPL, cTI is now significantly faster (speed-ups from a factor of two to more than an order of magnitude have been observed). The latest version of TerminWeb emphasizes termination analysis of typed logic programs.

Termination of logic programming where numerical computations are taken into account are studied in (Serebrenik and De Schreye 2001; Serebrenik and De Schreye 2002). The authors present some advanced techniques for explicitly dealing with integers and floating point numbers computations.

The *size-change termination principle* has been proposed in (Lee et al. 2001) for deciding termination of first-order functional programs. The resulting analysis is close to the TermiLog approach (Lindenstrauss and Sagiv 1997) and the authors establish its intrinsic complexity.

Finally, we point out that the system Ciao-Prolog (Bueno et al. 1997) adopts another approach for termination, based on complexity analysis (Debray et al. 1994).

6 Conclusion

We have presented cTI, the first bottom-up left-termination inference tool for ISO-Prolog, and its experimental evaluation over standard termination benchmarks as well as middle-sized and larger logic programs. Running cTI on large programs shows that the approach scales up satisfactorily. We believe that, thanks to the Parma Polyhedra Library, cTI is today the fastest and most robust termination inference tool for logic programs.

When a SCC is too large, computations relying on projection may become too expensive. So we have added for each computation which may be too costly a timeout and if necessary we are able to return a value which does not destroy the correctness of the analysis, although the quality of the inference is obviously weaker. It allows cTI to keep on analyzing the program. As a side effect, the running time of cTI is *linear* with respect to the number of SCC's in the call graph.

Finally, one can observe that the termination conditions computed in Section 2 are actually *optimal* with respect to the language used for describing classes of queries. Can one prove such properties automatically? (Mesnard et al. 2002) presents a first step in this direction.

Acknowledgments. We would like to thank Ulrich Neumerkel for numerous discussions we had on termination inference and for the help he provided while debugging cTI. Thanks also to the readers of a previous version of this paper for their comments.

Availability. cTI is distributed under the GNU General Public License. The analyzer, together with the programs analyzed for benchmarking, are available from cTI's web site: `http://www.cs.unipr.it/cTI`.

References

Apt, K. and Pellegrini, A. 1994. On the occur-check-free Prolog programs. *ACM Transactions on Programming Languages and Systems 16,* 3, 687–726.

Apt, K. R. 1997. *From Logic Programming to Prolog.* International Series in Computer Science. Prentice Hall.

Apt, K. R. and Pedreschi, D. 1990. Studies in pure Prolog: Termination. In *Computational Logic: Symposium Proceedings*, J. W. Lloyd, Ed. ESPRIT Basic Research Series. Springer-Verlag, Berlin, Brussels, Belgium, 150–176.

Apt, K. R. and Pedreschi, D. 1993. Reasoning about termination of pure Prolog programs. *Information and Computation 106,* 1, 109–157.

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

Arts, T. and Zantema, H. 1996. Termination of logic programs using semantic unification. In *Logic Program Synthesis and Transformation: Proceedings of the 5th International Workshop*, M. Proietti, Ed. Lecture Notes in Computer Science, vol. 1048. Springer-Verlag, Berlin, Utrecht, The Netherlands, 219–233.

Bagnara, R., Gori, R., Hill, P. M., and Zaffanella, E. 2001a. Finite-tree analysis for constraint logic-based languages. See Cousot (2001), 165–184.

Bagnara, R., Ricci, E., Zaffanella, E., and Hill, P. M. 2002. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis: Proceedings of the 9th International Symposium*, M. V. Hermenegildo and G. Puebla, Eds. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, Madrid, Spain, 213–229.

Bagnara, R., Zaffanella, E., Gori, R., and Hill, P. M. 2001b. Boolean functions for finite-tree dependencies. See Nieuwenhuis and Voronkov (2001), 579–594.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and Puebla, G. 1997. The Ciao Prolog system. Reference manual. Tech. Rep. CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM). Available from http://www.clip.dia.fi.upm.es/.

Bueno, F., de la Banda, M. G., and Hermenegildo, M. V. 1994. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Logic Programming: Proceedings of the 1994 International Symposium*, M. Bruynooghe, Ed. MIT Press Series in Logic Programming. The MIT Press, Ithaca, NY, USA, 253–268.

Clarke, E. M., Grumberg, O., and Peled, D. A. 2000. *Model Checking.* The MIT Press.

Codish, M. and Taboch, C. 1999. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming 41,* 1, 103–123.

Colin, S., Mesnard, F., and Rauzy, A. 1997. Constraint logic programming and mu-calculus. In *Proceedings of the ERCIM/COMPULOG Workshop on Constraints.* Schloss Hagenberg, Austria.

Cousot, P., Ed. 2001. *Static Analysis: 8th International Symposium, SAS 2001.* Lecture Notes in Computer Science, vol. 2126. Springer-Verlag, Berlin, Paris, France.

Cousot, P. and Cousot, R. 1992. Abstract interpretation and applications to logic programs. *Journal of Logic Programming 13,* 2&3, 103–179.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.* ACM Press, Tucson, Arizona, 84–96.

Crnogorac, L., Kelly, A. D., and Søndergaard, H. 1996. A comparison of three occur-check analysers. In *Static Analysis: Proceedings of the 3rd International Symposium*, R. Cousot and D. A. Schmidt, Eds. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Berlin, Aachen, Germany, 159–173.

De Schreye, D. and Decorte, S. 1994. Termination of logic programs: The never-ending story. *Journal of Logic Programming 19 & 20*, 199–260.

Debray, S. K., López-García, P., Hermenegildo, M. V., and Lin, N.-W. 1994. Estimating the computational cost of logic programs. In *Static Analysis: Proceedings* *of the 1st International Symposium*, B. Le Charlier, Ed. Lecture Notes in Computer Science, vol. 864. Springer-Verlag, Berlin, Namur, Belgium, 255–265.

Decorte, S. 1997. Enhancing the power of termination analysis of logic programs through types and constraints. Ph.D. thesis, Department of Computer Science, K. U. Leuven, Leuven, Belgium.

Decorte, S., De Schreye, D., and Vandecasteele, H. 1999. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems 21,* 6, 1137–1195.

Deransart, P., Ed-Dbali, A., and Cervoni, L. 1996. *Prolog: The Standard*. Springer-Verlag, Berlin.

Deransart, P., Ferrand, G., and Téguia:, M. 1991. NSTO programs (Not Subject to Occur-Check). In *Logic Programming: Proceedings of the 1991 International Symposium*, V. A. Saraswat and K. Ueda, Eds. MIT Press Series in Logic Programming. The MIT Press, San Diego, USA, 533–547.

Diaz, D. and Codognet, P. 2000. The GNU Prolog system and its implementation. In *Proceedings of the 2000 ACM Symposium on Applied Computing*, J. Carrol, E. Damiani, H. Haddad, and D. Oppenheim, Eds. Vol. 2. Como, Italy.

Francez, N., Grumberg, O., Katz, S., and Pnueli, A. 1985. Proving termination of Prolog programs. In *Proceedings of Third Workshop on Logics of Programs*, R. Parikh, Ed. Lecture Notes in Computer Science, vol. 193. Springer-Verlag, Berlin, Brooklyn, New York, 89–105.

Genaim, S. and Codish, M. 2001. Inferring termination conditions for logic programs using backwards analysis. See Nieuwenhuis and Voronkov (2001), 685–694.

Halbwachs, N. 1979. Thèse de 3$^{\text{ème}}$ cycle d'informatique. Ph.D. thesis, Université scientifique et médicale de Grenoble, Grenoble, France.

Hermenegildo, M. V. and Puebla, G., Eds. 2002. *Static Analysis: 9th International Symposium, SAS 2002*. Lecture Notes in Computer Science, vol. 2477. Springer-Verlag, Berlin, Madrid, Spain.

Hermenegildo, M. V., Puebla, G., and Bueno, F. 2000. An assertion language for constraint logic programs. In *Analysis and Visualization Tools for Constraint Programming*, P. Deransart, M. V. Hermenegildo, and J. Małuszyński, Eds. Lecture Notes in Computer Science, vol. 1870. Springer-Verlag, Berlin, 23–61.

ISO/IEC. 1995. *ISO/IEC 13211-1: 1995 Information technology — Programming languages — Prolog — Part 1: General core*. International Standard Organization.

King, A. and Lu, L. 2002. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming 2,* 4–5, 517–547.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. 2001. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, C. Norris and J. J. B. Fenwick, Eds. ACM SIGPLAN Notices, vol. 36. Association for Computing Machinery, London, UK, 81–92.

Levi, G. and Scozzari, F. 1995. Contributions to a theory of txistential termination for definite logic programs. In *Proceedings of the "1995 Joint Conference on Declarative Programming (GULP-PRODE'95)"*, M. Alpuente and M. I. Sessa, Eds. Marina di Vietri, Italy, 631–642.

Lindenstrauss, N. and Sagiv, Y. 1997. Automatic termination analysis of logic programs. In *Logic Programming: Proceedings of the Fourteenth International Conference on Logic Programming*, L. Naish, Ed. MIT Press Series in Logic Programming. The MIT Press, Leuven, Belgium, 63–77.

Lloyd, L. W. 1987. *Foundations of Logic Programming*, second ed. Springer-Verlag, Berlin.

Marchiori, M. 1996. Proving existential termination of normal logic programs. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, M. Wirsing and M. Nivat, Eds. Lecture Notes in Computer Science, vol. 1101. Springer-Verlag, Berlin, Munich, Germany, 375–390.

Marriott, K. and Stuckey, P. J. 1998. *Programming with Constraints: An Introduction.* The MIT Press.

Mesnard, F. 1996. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, M. J. Maher, Ed. MIT Press Series in Logic Programming. The MIT Press, Bonn, Germany, 7–21.

Mesnard, F. and Neumerkel, U. 2001. Applying static analysis techniques for inferring termination conditions of logic programs. See Cousot (2001), 93–110.

Mesnard, F., Payet, E., and Neumerkel, U. 2002. Detecting optimal termination conditions of logic programs. See Hermenegildo and Puebla (2002), 509–525.

Mesnard, F. and Ruggieri, S. 2003. On proving left termination of constraint logic programs. *ACM Transactions on Computational Logic 4,* 2, 1–26.

Nieuwenhuis, R. and Voronkov, A., Eds. 2001. *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001).* Lecture Notes in Artificial Intelligence, vol. 2250. Springer-Verlag, Berlin, Havana, Cuba.

Plümer, L. 1990. *Terminations Proofs for Logic Programs.* Number 446 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

Ruggieri, S. 1999. Verification and validation of logic programs. Ph.D. thesis, Dipartimento di Informatica, Università di Pisa, Pisa, Italy. Printed as Report TD-6/99.

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in prolog. See Nieuwenhuis and Voronkov (2001), 654–668.

Serebrenik, A. and De Schreye, D. 2002. On termination of logic programs with floating point computations. See Hermenegildo and Puebla (2002), 151–164.

Sohn, K. and Van Gelder, A. 1991. Termination detection in logic programs using argument sizes. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems.* Association for Computing Machinery, Denver, Colorado, 216–226. Extended abstract.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The execution algorithm of Mercury: An efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1–3, 17–64.

Søndergaard, H. 1986. An application of abstract interpretation of logic programs: Occur check reduction. In *Proceedings of the 1986 European Symposium on Programming*, B. Robinet and R. Wilhelm, Eds. Lecture Notes in Computer Science, vol. 213. Springer-Verlag, Berlin, Saarbrücken, Federal Republic of Germany, 327–338.

Speirs, C., Somogyi, Z., and Søndergaard, H. 1997. Termination analysis for Mercury. In *Static Analysis: Proceedings of the 4th International Symposium*, P. Van Hentenryck, Ed. Lecture Notes in Computer Science, vol. 1302. Springer-Verlag, Berlin, Paris, France, 157–171.

Stärk, R. F. 1998. The theoretical foundations of LPTP (a Logic Program Theorem Prover). *Journal of Logic Programming 36,* 3, 241–269.

Ullman, J. D. and Van Gelder, A. 1988. Efficient tests for top-down termination of logical rules. *Journal of the ACM 35,* 2, 345–373.

Vasak, T. and Potter, J. 1986. Characterisation of terminating logic programs. In *Proceedings of the 1986 Symposium on Logic Programming*, R. M. Keller, Ed. IEEE Computer Society Press, Salt Lake City, Utah, 140–147.

Verschaetse, K. 1992. Static termination analysis for definite Horn clause programs. Ph.D. thesis, Department of Computer Science, K. U. Leuven, Leuven, Belgium.