Applying static analysis techniques for inferring termination conditions of logic programs (preliminary version)

F. Mesnard[1] and U. Neumerkel[2]

[1] Iremia - Université de La Réunion, France
fred@univ-reunion.fr
[2] Institut für Computersprachen - Technische Universität Wien, Austria
ulrich@mips.complang.tuwien.ac.at

Abstract. We present the implementation of cTI, a system for universal left-termination *inference* of logic programs, which heavily relies on static analysis techniques.
Termination inference generalizes 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 described and some optimizations are discussed. Running times for classical examples from the termination literature in LP and for some middle-sized logic programs are given.

1 Introduction

Termination is a crucial aspect of program verification. For logic programs [27, 3], 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.

A first observation (see [42]) was to recognize that there were two notions of termination for logic programs which we explain now. Assume that we use a standard Prolog engine. *Existential* termination means that either the computation finitely fails or it produces one solution in finite time (then it may loop if we ask for another solution). On the other hand, *universal* termination means that the computation produces all solutions (if we repeatedly ask for another solution) in finite time then terminates. Although existential termination plays an important rôle in the termination of normal logic programs, this notion has severe drawbacks: it is not instantiation-closed (a goal may existentially terminate, hence it is not and-compositional (two goals may existentially terminate, but not their conjunction), but some of its instances may not terminate), and it depends on the textual order of clauses. Universal termination has none of these problematic features.

So existential termination has been the subject of a few papers (see e.g. [25, 28]). The research efforts were mainly on universal termination and can be divided in two groups (a survey is given in [20]): characterizing termination [4, 1, 34] and weakening such undecidable criteria to get decidable sufficient conditions (e.g. [41, 33, 43]) that lead to actual implementations. Our research belongs to both streams. A companion paper describes our approach in the theoretical setting of acceptability for constraint logic programming [31]. The present paper focuses on the implementation[1] of our ideas, which heavily relies on static analysis techniques.

Our main innovation compared to other recent works related to automated termination analysis [26, 16, 40, 10] is that we *infer* sufficient universal termination conditions from the text of any Prolog program. Inference implies that we adopt a bottom-up approach to termination. *There is no need to define a class of queries of interest.* We point out that giving a class of queries is imposed by all other works we are aware of (but if required, such classes can be easily simulated within our framework).

Our system, called cTI for *constraint-based Termination Inference*, can be used at the URL `http://www.complang.tuwien.ac.at/cti` and has been realized in SICStus Prolog. Currently the only requirement we impose on ISO-Prolog [18] programs to ensure correctness of the analysis is that they must not create infinite rational terms. Hence we only consider NTSO (*not subject to occur check*) programs [19] that can be safely executed with any standard complying system or an execution with occurs check. Our tool cTI is also available in the LP environment GUPU [32]. In what follows, we give an intuitive view of the analysis, some insights of its underlying, and running times for cTI.

2 An overview of cTI

Our aim is to compute classes of queries for which universal left termination is guaranteed.

Definition 1. *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 $\leftarrow q(\tilde{t})$ such that, for any goal $G \in TC_q$, each derivation of G using the left-to-right selection rule is finite.*

Syntactic informations are often too weak to reason about non-trivial programs. Some semantics information is required. For this reason our analyzer uses three main constraint structures [23, 24]: 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* [29], which are a simple form [21] of abstract interpretation [12, 13],

[1] A preliminary version of this paper was presented at the Workshop on Parallelism and Implementation Technology for Constraint Logic Programming Languages (ed. Ines de Castro Dutra), CL'2000, London.

also coined *abstract compilation*. We illustrate our method to infer termination conditions by using the predicates `app/3`, `app3/4`, and `nrev/2`.

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:

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

   E.g.$\|f(0,0)\|_{\mathrm{Term-Size}} = 1$. All non-monotonic elements of the program are approximated by monotone constructs. E.g., Prolog's unsound negation `\+G` is approximated by `((G,false);true)`. The main point here is that we maintain that if a goal in $P^{\mathbb{N}}$ is terminating, then also the corresponding goals in P terminate.

$\mathrm{app}_{\mathbb{N}}$(0, Xs, Xs).
$\mathrm{app}_{\mathbb{N}}$(1+X+Xs,Ys,1+X+Zs) ←
  $\mathrm{app}_{\mathbb{N}}$(Xs, Ys, Zs).
	$\mathrm{nrev}_{\mathbb{N}}$(0, 0).
	$\mathrm{nrev}_{\mathbb{N}}$(1+X+Xs,Ys) ←
	  $\mathrm{nrev}_{\mathbb{N}}$(Xs, Zs),
	  $\mathrm{app}_{\mathbb{N}}$(Zs, 1+X, Ys).
		$\mathrm{app3}_{\mathbb{N}}$/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 *post*) that hold for every solution. The actual computation is performed with CLP($\mathbb{Q}$), using a generic fixpoint calculator with a standard widening (see section 4). In our example we are able to determine the least model. In general, however, only a less precise model is determined. For each recursive predicate p (the only source of potential non-termination), we compute a linear level mapping (see section 5) called μ_p. For instance, the meaning of $\mu^{\mathbb{N}}_{\mathrm{app}}$ is: for any ground recursive clause defining $app_{\mathbb{N}}$, the first and the third argument decrease. The meaning of $\mu^{\mathbb{N}}_{\mathrm{rev}}$ is: for any ground recursive clause defining $rev_{\mathbb{N}}$, the first argument decreases (we do not have to compare $nrev_{\mathbb{N}}$ and $app_{\mathbb{N}}$ at this step). The need of the numeric model of P is only justified by the need to compute a level mapping on a more semantical basis than by a purely syntactic approach.

(least) models	level mappings
$post^{\mathbb{N}}_{\mathrm{app}}(x,y,z) \equiv z = x + y$	$\mu^{\mathbb{N}}_{\mathrm{app}}(x,y,z) \equiv min(x,z)$
$post^{\mathbb{N}}_{\mathrm{nrev}}(x,y) \equiv x = y$	$\mu^{\mathbb{N}}_{\mathrm{nrev}}(x,y) \equiv x$
$post^{\mathbb{N}}_{\mathrm{app3}}(x,y,z,u) \equiv u = x + y + z$	—

3. $P^{\mathbb{N}}$ is mapped to $P^{\mathcal{B}}$, a program in CLP($\mathcal{B}$). Here 1 means that an argument is bounded 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 allows us to always compute the least model. For each predicate, its previously computed linear level mapping is represented by a single boolean term.

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

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

4. Using all previously determined informations, $P^{\mathcal{B}}$ is translated into the following system of boolean fixpoint formulæ that ensures the propagation of the finiteness of the level mappings through the call graph and the Prolog dataflow.

$$pre_{\mathrm{app}} = \nu A.\lambda(b,c,d). \begin{cases} b \vee d \\ \wedge \\ \forall f,g,h.[(f \wedge g \leftrightarrow b) \wedge (f \wedge h \leftrightarrow d)] \rightarrow A(g,c,h) \end{cases}$$

$$pre_{\mathrm{nrev}} = \nu A.\lambda(b,c). \begin{cases} b \\ \wedge \\ \forall d,e,f.[d \wedge e \leftrightarrow b] \rightarrow A(e,f) \\ \wedge \\ \forall d,e,f.[(f \leftrightarrow e) \wedge (d \wedge e \leftrightarrow b)] \rightarrow pre_{\mathrm{app}}(f,d,c) \end{cases}$$

$$pre_{\mathrm{app3}} = \nu A.\lambda(b,c,d,e). \begin{cases} \forall f.1 \rightarrow pre_{\mathrm{app}}(b,c,f) \\ \wedge \\ \forall f.[f \leftrightarrow c \wedge b] \rightarrow pre_{\mathrm{app}}(f,d,e) \end{cases}$$

The resolution of this system (computation of the greatest fixpoint by means of a boolean μ-solver [11]) gives, for each predicate symbol, its boolean termination condition:

$$\begin{aligned} pre_{\mathrm{app}}(x,y,z) &\equiv x \vee z \\ pre_{\mathrm{nrev}}(x,y) &\equiv x \\ pre_{\mathrm{app3}}(x,y,z,u) &\equiv (x \wedge y) \vee (x \wedge u) \end{aligned}$$

These boolean termination conditions lift to termination conditions (definition 1) with the following interpretation:

- any goal ← c, `app(X,Y,Z)`, where c is a CLP($\mathcal{H}$) constraint, left-terminates if `X` or `Z` are ground in c.
- any goal ← c, `nrev(X,Y)` left-terminates if `X` is ground in c.
- any goal ← c, `app3(X,Y,Z,U)` left-terminates if either `X` and `Y` are ground in c or `X` and `U` are ground in c.

The correctness of the analysis is based on is the following result [29, 31]:

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: $p(\tilde{x}) \leftarrow c_k,\ p_{k,1}(\tilde{x}_{k,1}), \dots\ p_{k,n_k}(\tilde{x}_{k,n_k})$ and for each $q \notin \bar{p}$ and appearing in the rules defining $\bar{p}$, a boolean termination condition pre_q has been computed. If the set of boolean terms $\{pre_p\}_{p\in\bar{p}}$ verifies:*

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

then $\{pre_p\}_{p\in\bar{p}}$ is a correct boolean termination condition for $\bar{p}$.

3 Running cTI

3.1 Standard programs from the termination literature in LP

Tables 1 and 2 presents timings and results of cTI using some standard LP termination benchmarks, where the following abbreviations mean:

- *cTI time*: the running time for cTI to infer termination conditions;
- *top-level predicate*: the predicate of interest;
- *Others: checked*: the class of queries checked by the analyzers of [17, 26, 40];
- *result*: the best result (y > n > ?) among [17, 26, 40];
- *cTI: inferred*: the termination condition inferred by cTI (1 means that any call to the predicate terminates, 0 means that cTI can not find a terminating mode for that predicate);

For the MERGESORT (and similarly for MERGESORT_AP), the problem lies in the *split*/3 predicate, aiming at splitting a list (the first argument) in two sublists (the second and third argument) which length are *almost* equal:

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

Using the *term_size* norm, depending on the precision of the numeric abstract interpreter (see section 4), we have:

$$prec \leq 1 \Rightarrow post_{\text{split}}^{\mathbb{N}}(x, y, z) \equiv true$$
$$prec \geq 2 \Rightarrow post_{\text{split}}^{\mathbb{N}}(x, y, z) \equiv x = y + z$$

Switching to the *list_size* norm defined as follows:

$$\|t\|_{\text{list-size}} = \begin{cases} 1 + \|u\|_{\text{list-size}} & \text{si } t = [s|u] \\ t & \text{si } t \text{ est une variable} \\ 0 & \text{sinon} \end{cases}$$

Table 1. Programs from [20, 2], cTI 0.29, Athlon 750 MHz, 256Mo, SICStus 3.8.4

times in [s] program	cTI time	top-level predicate	Others: checked	result	cTI: inferred
PERMUTE	0.15	$permute(x, y)$	x	yes	x
DUPLICATE	0.05	$duplicate(x, y)$	x	yes	$x \vee y$
SUM	0.18	$sum(x, y, z)$	$x \wedge y$	yes	$x \vee y \vee z$
MERGE	0.26	$merge(x, y, z)$	$x \wedge y$	yes	$(x \wedge y) \vee z$
DIS-CON	0.24	$dis(x)$	x	yes	x
REVERSE	0.08	$reverse(x, y, z)$	$x \wedge z$	yes	x
APPEND	0.09	$append(x, y, z)$	$x \wedge y$	yes	$x \vee z$
LIST	0.01	$list(x)$	x	yes	x
FOLD	0.10	$fold(x, y, z)$	$x \wedge y$	yes	y
LTE	0.13	$goal$	1	yes	1
MAP	0.09	$map(x, y)$	x	yes	$x \vee y$
MEMBER	0.03	$member(x, y)$	y	yes	y
MERGESORT	0.43	$mergesort(x, y)$	x	no	0
MERGESORT	0.57	$mergesort(x, y)$	x	no	x
MERGESORT_AP	0.79	$mergesort\_ap(x, y, z)$	x	yes	z
MERGESORT_AP	0.92	$mergesort\_ap(x, y, z)$	x	yes	$x \vee z$
NAIVE_REV	0.12	$naive\_rev(x, y)$	x	yes	x
ORDERED	0.04	$ordered(x)$	x	yes	x
OVERLAP	0.05	$overlap(x, y)$	$x \wedge y$	yes	$x \wedge y$
PERMUTATION	0.15	$permutation(x, y)$	x	yes	x
QUICKSORT	0.39	$quicksort(x, y)$	x	yes	x
SELECT	0.08	$select(x, y, z)$	y	yes	$y \vee z$
SUBSET	0.09	$subset(x, y)$	$x \wedge y$	yes	$x \wedge y$
SUBSET	0.09	$subset(x, y)$	y	no	$x \wedge y$
SUM	0.12	$sum(x, y, z)$	z	yes	$y \vee z$

we get:

$$prec \leq 1 \Rightarrow post^{\mathbb{N}}_{\text{split}}(x, y, z) \equiv true$$
$$prec = 2 \Rightarrow post^{\mathbb{N}}_{\text{split}}(x, y, z) \equiv x = y + z$$
$$prec \geq 3 \Rightarrow post^{\mathbb{N}}_{\text{split}}(x, y, z) \equiv x = y + z \wedge 0 \leq y - z \leq 1$$

This last model is sufficiently precise for proving termination of the considered programs.

3.2 Middle-sized programs

Table 4 presents timings of cTI using some standard benchmarks[2] from the LP program analysis community. We have chosen twelve middle-sized well-known

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

Table 2. programs from [33], cTI 0.29, Athlon 750 MHz, 256Mo, SICStus 3.8.4

times in [s] program	cTI time	top-level predicate	Others: checked	result	cTI: inferred
PL1.1	0.08	append(x,y,z)	$x \wedge y$	yes	$x \vee z$
PL1.1	0.08	append(x,y,z)	z	yes	$x \vee z$
PL1.2	0.16	perm(x,y)	x	yes	x
PL2.3.1	0.01	p(x,y)	x	no	0
PL3.1.1	0.09	a	?	?	0
PL3.5.6	0.05	p(x)	1	no	x
PL3.5.6A	0.06	p(x)	1	yes	x
PL3.5.6A	0.06	p(x)	1	yes	1
PL4.0.1	0.10	append3(x,y,z,t)	$x \wedge y \wedge z$	yes	$(x \wedge y) \vee (x \wedge t)$
PL4.4.3	0.26	merge(x,y,z)	$x \wedge y$	yes	$(x \wedge y) \vee z$
PL4.4.6A	0.12	perm(x,y)	x	yes	x
PL4.5.2	0.17	s(x,y)	x	no	0
PL4.5.3A	0.01	p(x)	x	no	0
PL4.5.3B	0.02	goal	?	?	0
PL4.5.3C	0.01	goal	?	?	0
PL5.2.2	3.41	turing(x,y,z,t)	$x \wedge y \wedge z$	no	0
PL6.1.1	0.39	qsort(x,y)	x	yes	x
PL7.2.9	0.21	mult(x,y,z)	$x \wedge y$	yes	$x \wedge y$
PL7.6.2A	0.14	reach(x,y,z)	$x \wedge y \wedge z$	no	0
PL7.6.2B	0.22	reach(x,y,z,t)	$x \wedge y \wedge z \wedge t$	no	0
PL7.6.2C	0.29	reach(x,y,z,t)	$x \wedge y \wedge z \wedge t$	yes	$z \wedge t$
PL8.2.1	0.43	mergesort(x,y)	x	no	0
PL8.2.1	0.58	mergesort(x,y)	x	no	x
PL8.2.1A	0.47	mergesort(x,y)	x	yes	x
MERGESORT_T	0.94	mergesort(x,y)	x	yes	x
PL8.3.1	0.26	minsort(x,y)	x	no	$x \wedge y$
PL8.3.1A	0.24	minsort(x,y)	x	yes	x
PL8.4.1	0.13	even(x)	x	yes	x
PL8.4.2	0.52	e(x,y)	x	yes	x

logic programs. Almost all the programs are taken from [7] except CREDIT, PLAN and MINISSAEXP. Table 3 describes them, where the following abbreviations mean:

- *lines* is the number of lines of the Prolog program in pure form (e.g. no disjunction), with one predicate symbol per line and no blank line;
- *facts* and *rules* denote, respectively, the numbers of facts (unit clauses) and rules (non-unit clauses) in the program;
- *sccs* gives the number of strongly connected components (sccs, i.e. cycles of mutually recursive predicate symbols) in the call graph;
- *length* denotes the number of predicate symbols in the longest cycle in the call graph;

– *vars* denotes the sum of the arities of the predicate symbols of the longest cycle in the call graph.

Table 3. Informations about analyzed programs.

Program	lines	facts	rules	sccs	length	vars
ANN	571	101	99	44	2	7
BID	108	24	26	20	1	4
BOYER	275	63	78	25	2	5
BROWSE	107	4	29	15	1	6
CREDIT	108	33	24	24	1	4
MINISSAEXP	833	37	223	100	5	17
PEEPHOLE	322	72	80	11	2	5
PLAN	64	12	17	16	1	4
QPLAN	403	63	87	38	3	11
RDTOK	285	7	57	12	4	12
READ	299	15	75	17	7	33
WARPLAN	304	43	68	33	3	14

The first five columns of Table 4 indicate the time for computing:

– a model $Post_{\mathbb{N}}$ (section 4);
– the constraint defining the level mapping μ (section 5);
– the concrete level mapping;
– the least model $Post_{\mathcal{B}}$;
– the boolean termination conditions.

The timings are minimum execution times over ten iterations. Next we give:

– the total runtime (including various syntactic transformations);
– the speed of the analysis (the average number of analyzed lines of code in one second);
– the quality of the analysis, computed as the ratio of the number of relations which have a a non-empty termination condition over the total number of relations.

Let us comment on the results of Table 4.

The speed of the analysis is surprisingly slower for PEEPHOLE than for the other programs. A more careful look on its code shows that its call graph contains 5 cycles of length 2, which slow down the computation of the constraints defining the level mapping.

We note that cTI was able to prove that BID, CREDIT, and PLAN are *left-terminating* (see [3], every ground atom left-terminates). For any such program

P, T_P has only one fixpoint ([3], Theorem 8.13), which helps for proving partial correctness. Moreover, the ground semantics of such a program is decidable (Prolog is the decision procedure !), which helps for testing and validating the program.

Table 4. middle-sized programs, cTI 0.29, Athlon 750 MHz, 256Mo, SICStus 3.8.4

times in [s] program	$Post_{\mathbb{N}}$	C_μ	μ	$Post_{\mathcal{B}}$	TC	total time	lines/sec	q %
ANN	1.07	2.62	0.17	0.46	0.13	5.43	105	28
BID	0.17	0.33	0.03	0.09	0.04	0.81	133	70
BOYER	2.55	0.36	0.03	0.25	0.05	3.91	70	75
BROWSE	0.37	1.01	0.08	0.12	0.03	1.81	59	30
CREDIT	0.12	0.18	0.04	0.07	0.03	0.61	177	87
MINISSAEXP	2.98	4.77	0.49	0.73	0.38	11.03	75	65
PEEPHOLE	1.24	8.94	0.14	0.47	0.12	11.69	28	49
PLAN	0.13	0.32	0.03	0.09	0.03	0.71	90	58
QPLAN	1.52	4.32	0.23	0.62	0.16	7.56	53	50
RDTOK	1.22	0.95	0.07	0.26	0.05	2.92	98	23
READ	1.05	5.25	0.03	0.34	0.16	7.29	41	39
WARPLAN	0.82	1.67	0.03	0.27	0.03	3.18	96	23
mean	23%	54%	2%	7%	2%	100%	85	50%

4 Fixpoint Computations

As explained in section 2, we have to compute some models of two versions of the initial program: $P^{\mathbb{N}}$, the CLP($\mathbb{N}$) version, and $P^{\mathcal{B}}$, the CLP($\mathcal{B}$) version. To this aim, we have developed an abstract immediate consequence operator U_P. This operator is quite similar to the well-known T_P. This section borrows numerous results found in [12, 15, 13, 14].

4.1 The algorithm

The key of our abstract computation is the notion of *rational interpretation* for a predicate symbol p:

Definition 2. *Let P be a program and p be a predicate symbol of P. We call a rational interpretation of p an equivalence of the form: $p(\tilde{x}) \leftrightarrow c$ where c, a disjunction of conjunctions of atomic constraints, is a formula s.t. $vars(c) \subseteq \tilde{x}$. We extend this notion to P: a rational interpretation of P is a set I containing exactly one interpretation for each predicate symbol p of P.*

We write $\mathcal{I}$ the set of all rational interpretations. We want to compute a rational interpretation which is a model of P. To this end, we define below an operator U_P, where we impose $\bigvee(\tilde{x}) \supseteq \cup\tilde{x}$.

Definition 3. *U_P is a function on $\mathcal{I}$ defined for any rational interpretation I of a program P by:*

$$\begin{aligned} U_P(I) = \{\mathtt{p}(\tilde{x}) \leftrightarrow c \mid\ & c \equiv \textstyle\bigvee_{cl\in P}\left(\exists_{-\tilde{x}}(c_0 \wedge \bigwedge_{1\le i\le n} c_i)\right), \\ & cl \equiv \mathtt{p}(\tilde{x}) \leftarrow c_0 \diamond \mathtt{p}_1(\tilde{x}_1), \ldots, \mathtt{p}_n(\tilde{x}_n) \text{ and} \\ & \forall i \in [\![1;n]\!], \mathtt{p}_i(\tilde{x}_i) \leftrightarrow c_i \in I\} \end{aligned}$$

We define the successive powers of U_P as usual. It turns out that U_P is monotone and continuous. Now let us establish a link between the meaning of a program P and the U_P operator. First, we give a ground semantics of a rational interpretation:

Definition 4. *Let I be a rational interpretation, we define the semantics of I by: $[I] = \{p(\tilde{d}) \mid p(\tilde{x}) \leftrightarrow c \in I, \tilde{d} \in \tilde{D}_\chi, \models_\chi c(\tilde{d})\}$ where D_χ is the domain of computation.*

For any interpretation I, we have: $T_P([I]) \subseteq [U_P(I)]$. Now, as a fixpoint I of U_P verifies $T_P([I]) \subseteq [U_P(I)] = [I]$, we get: any fixpoint of U_P is a model of P.

For CLP($\mathcal{B}$), $\bigvee$ is the union of two constraints. Hence we have $T_P([I]) = [U_P(I)]$. It justifies the use of U_P for computing the least boolean model (= lfp(U_P)) of P. Figure 1 presents an algorithm for the U_P operator.

function $\mathtt{U_P}(I) : J$
Require: I is a rational interpretation of P.
Ensure: J is a rational interpretation of P.

$J \leftarrow \emptyset$;
for all clause$(p(\tilde{x}) \leftarrow c\ \diamond\ p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n)) \in P$ **do**
  for $i \leftarrow 1$ **to** n **do**
    let $p_i(\tilde{x}_i) \leftrightarrow c_i \in I$;
  end for
  let $p(\tilde{x}) \leftrightarrow c' \in J$;
  $c \leftarrow \bigvee(c', \Pi_{\tilde{x}}(c_1 \wedge \ldots \wedge c_n))$;
  $J \leftarrow \mathit{update}(J, p(\tilde{x}) \leftrightarrow c))$;
end for
return J ;

Fig. 1. U_P, a T_P-like operator.

4.2 Widenings

For CLP($\mathbb{Q}$), lfp(U_P) is not, in general, reachable in finite time. That is the reason why a *widening* operator ($\triangledown$) [12] is used. The widening operator is used to force convergence of the U_P operator. This operator has a major impact on the precision and the speed of the computation. In cTI, we adopt such an approach for the numeric model computation only since the least boolean model is finitely reached. However, we have coded a *generic* fixpoint calculator for both CLP($\mathbb{Q}$) and CLP($\mathcal{B}$) [8, 22, 30]. A simple widening ($\triangledown_1$) on system of linear inequalities can be found in [15]. This is an equivalent definition [35]:

Definition 5. *Let S_1 and S_2 be two sets of linear inequalities defining two polyhedra in $\mathbb{Q}^n$. Then:*

$$S_1 \ \triangledown_1 \ S_2 = \{\beta \in S_1 \mid S_2 \Rightarrow \beta\}$$

$\triangledown_2$ (see [14]) is an improved version of $\triangledown_1$, which is simplified for efficiency in [36]:

Definition 6. *Let S_1 and S_2 be two sets of linear inequalities defining two polyhedra in $\mathbb{Q}^n$. Then:*

$$\begin{aligned} S_1 \ \triangledown_2 \ S_2 = & \{\beta \in S_1 \mid S_2 \Rightarrow \beta\} \bigcup \\ & \{\gamma \in S_2 \mid S_1 \Rightarrow \gamma \bigwedge \exists \beta \in S_1 ((S_1 - \{\beta\}) \cup \{\gamma\}) \Rightarrow \beta\} \end{aligned}$$

Tests to determinate the impact of using $\triangledown_1$ or $\triangledown_2$ on the accuracy of cTI are under construction. Figure 2 presents an algorithm for successive iterations of U_P until it reaches a fixpoint. In the current implantation of cTI, for CLP($\mathbb{Q}$), *prec* is set to 2.

It remains to show that $I_n = \texttt{ite\_U}_\texttt{P}(prec)$ is a model of P. First, note that, by induction on k, $I_k \subseteq I_{k+1}$. So we have in fact an equality when we reach line 11 for the last time: $I_n = I_{n-1}$. Then the last assignment for I_n is either line 7 if $n \leq prec$. In this case, we have $I_n = U_P(I_n)$ hence $T_P([I_n]) \subseteq [I_n]$. Or the last assignment for I_n is line 9: $I_n \leftarrow I_n \triangledown U_P(I_n) \supseteq U_P(I_n)$ by definition of any widening operator $\triangledown$. But we know that $T_P([I]) \subseteq [U_P(I)]$ for all I. Again, $T_P([I_n]) \subseteq [I_n]$.

4.3 Optimization

Since the fixpoint computation engine is used twice, making it as efficient as possible is quite important. *The current optimization takes all unit clauses defining the predicate symbols of the analyzed scc into account in a single pass and then processes only the non-unit clauses of the scc.* Table 5 shows the timings between the non-optimized version of U_P and the optimized version. Note that we also replace the union operator $\bigvee$ of line 7 of the algorithm presented Fig. 1 by a convex hull (in both versions for CLP($\mathbb{Q}$), opt and nopt), which can be easily coded via projection in CLP($\mathbb{Q}$) using a trick which first appears in [5] (see also [6]).

function $\texttt{ite\_U}_\texttt{P}(prec) : I_n$

Require: $prec$ is a non-negative integer.
Ensure: I_n is a rational interpretation such that $\text{lfp}(\texttt{U}_\texttt{P}) \subseteq I_n$.

$n \leftarrow 0$;
$I_n \leftarrow \emptyset$;
repeat
    $I \leftarrow \texttt{U}_\texttt{P}(I_n)$;
    $n \leftarrow n + 1$;
    if $n \leq prec$ **then**
        $I_n \leftarrow I$;
    else
        $I_n \leftarrow I_{n-1} \nabla I$;
    end if
until $I_n \subseteq I_{n-1}$
return I_n ;

Fig. 2. An algorithm to finitely reach a super set of $\text{lfp}(\texttt{U}_\texttt{P})$.

5 Computing level-mappings

One key concept in many approaches for termination lies in the use of *level mappings*, i.e. mappings from ground atoms to natural numbers. We present an improvement of an already known technique for their automatic generation. Indeed, K. Sohn and A. Van Gelder have described in 1991 an algorithm (SVG in short, see [38]) based on linear programming which ensures the existence of linear level mappings. This method, despite its power, does not seem to be very well-known among researchers aiming at automating termination. Hence we recall it after some preliminaries. Then, the remaining subsections propose an extensions to SVG.

5.1 Preliminaries

We consider pure CLP($\mathbb{N}$) programs, with three predefined symbols for constraints: $=$, $\geq$, and $\leq$ and their standard meaning. Those programs are abstractions of (constraint) logic programs using (fixed or inferred) norms. We assume that clauses are written in flat form: $p_0(\tilde{x_0}) \leftarrow c_0, p_1(\tilde{x_1}), c_1, \ldots, c_{l-1}, p_l(\tilde{x_l}), c_l$, with $i \neq j \rightarrow \tilde{x_i} \cap \tilde{x_j} = \emptyset$ (where $\emptyset$ denotes the empty set). For sake of concision, *we disallow mutually recursive predicates* (this restriction does not apply to cTI). Note that we frequently switch to CLP($\mathbb{Q}^+$) as some computational problems in this structure are much cheaper (e.g. satisfiability). There is clearly a loss in the precision of the analysis: results are correct but not complete. From now on, we write CLP for CLP($\mathbb{N}$) or CLP($\mathbb{Q}^+$). Section 4 shows how we can compute a model M for a CLP program P, where each predicate $p(\tilde{x})$ is defined as a (finite) conjunction of CLP constraints. We use this model to simplify the program P.

Table 5. Impact of the optimization on the analysis times.

times in [s]	$Post_{\mathbb{N}}$			$Post_{\mathcal{B}}$		
Programs	opt	nopt	gain	opt	nopt	gain
ANN	1.07	1.33	20%	0.46	0.65	29%
BID	0.17	0.27	37%	0.09	0.14	35%
BOYER	2.55	3.48	27%	0.25	0.48	48%
BROWSE	0.37	0.35	-5%	0.12	0.15	20%
CREDIT	0.12	0.19	37%	0.07	0.13	46%
MINISSAEXP	2.98	2.74	-9%	0.73	1.17	38%
PEEPHOLE	1.24	1.35	8%	0.47	0.63	25%
PLAN	0.13	0.20	35%	0.09	0.13	31%
QPLAN	1.52	1.88	19%	0.62	0.86	28%
RDTOK	1.22	0.70	-74%	0.26	0.29	10%
READ	1.05	1.26	17%	0.34	0.50	32%
WARPLAN	0.82	0.96	15%	0.27	0.37	27%
average			11%			31%
min.			-74%			10%
max.			37%			48%

Definition 7. *Let M_P be a model of the CLP program P. The definition of a predicate p is* simplified wrt M *when, for the clauses defining p/n, we add to the right of each predicate $q(\tilde{x})$ its meaning $c_q(\tilde{x})$ relative to M_P. Moreover, those predicates $q/m \neq p/n$ which appear in the bodies are replaced by true (e.g. the dummy constraint $0 = 0$). Hence we end with a finite set of CLP clauses of the form: $p(\tilde{x_0}) \leftarrow c_0, p(\tilde{x_1}), c_1, \ldots, c_{l-1}, p(\tilde{x_l}), c_l$. The simplified program is denoted P_M^{simpl}.*

We are interested in the automatic discovery of linear level mappings.

Definition 8. *Let p/n be a recursive predicate symbol of a CLP program P. A* linear level mapping *μ for $p(x_1, \ldots, x_n)$ is a linear relation $\sum_{i=1}^{n} \mu_i x_i$, where the coefficients μ_i are non-negative integers.*

Such linear level mappings should satisfy a property ensuring their usefulness for left-termination:

Definition 9. *A linear level mapping μ for p is* valid wrt $P_{M_P}^{simpl}$ *if for each clause recursive defining p in P_M^{simpl}, say $p(\tilde{x_0}) \leftarrow c_0, p(\tilde{x_1}), c_1, \ldots, c_{l-1}, p(\tilde{x_l}), c_l$, for $k = 0$ to $l - 1$, $\bigwedge_{i=0}^{k} c_i \rightarrow \mu^T \tilde{x_0} \geq 1 + \mu^T \tilde{x_k}$, where μ^T denotes the transpose of the vector μ.*

5.2 The algorithm SVG

Let us first quickly review the algorithm of Sohn and Van Gelder. It aims at checking the existence of one valid linear level mapping. SVG starts with a pure

CLP program P and a constrained goal. A top-down boundedness analysis (see [29, 30]) reveals the calling modes of each predicate. Arguments are detected as either bounded (denoted b) or unbounded (u). A CLP model M is computed and P is simplified to P_M^{simpl}. Then SVG examines each recursive procedure p/n in turn (the precise order does not matter). Let us symbolically define the level mapping for $p(x_1, \ldots, x_n)$ as $\mu^T\tilde{x} = \sum_{1\leq i\leq n} \mu_i^{u\ or\ b} x_i$ where $\mu_i^u = 0$ is x_i is labelled as unbounded wrt the calling mode of p/n and $\mu_i^b \geq 0$ if x_i is labelled as bounded. Each clause r_i is processed. For one such clause, l simplified rules (for $k = 0$ to $k = l-1$) are constructed: $p(\tilde{x_0}) \leftarrow \bigwedge_{0\leq j\leq k} c_j, p(\tilde{x_k})$. One can assume that the constraint $C_{ij} = \bigwedge_{0\leq j\leq k} c_j$ is satisfiable, already projected onto $\tilde{x_0} \cup \tilde{x_k}$, only contains inequalities of the form $\leq$, and implies $\tilde{x_0} \geq 0$ and $\tilde{x_k} \geq 0$. Such a simplified rule gives rise to the following (pseudo-)linear programming problem

$$minimize\ \theta = \mu^T(\tilde{x_0} - \tilde{x_k})\ subject\ to\ C_{ij} \tag{1}$$

A valid linear level mapping μ exists (at least for this recursive call of this clause) if $\theta^* \geq 1$ where θ^* denotes the minimum of the objective function. Unfortunately, because of the symbolic constants μ, (1) is *not* a linear programming problem.

The *clever* idea of the authors is to consider its dual form:

$$maximize\ \eta = \tilde{y}\beta\ subject\ to\ \tilde{y} \geq 0 \wedge \tilde{y}A \geq (\mu, -\mu) \tag{2}$$

By duality theory (see [37] for instance), we have $\theta^* = \eta^*$. Now, the authors observe that μ appears linearly in the dual problem (it is not true for (1)) because no μ_i appears in A. Hence (2) can be rewritten, by adding $\eta \geq 1$ and $\tilde{\mu^b} \geq 0 \wedge \tilde{\mu^u} = 0$, as S_{ij}, a set of linear inequations. If the conjunction $S_p = \wedge_{i,j} S_{ij}$ for each recursive call and for each clause defining p/n is satisfiable, then there exists a valid linear level mapping for p/n.

5.3 An extension of SVG

Instead of checking satisfiability of S_p, we can project it onto μ (we do not need the top-down boundeness analysis explained subsection 5.2, all arguments are assumed bounded). Hence we get in one constraint *all* the valid linear level mappings. It remains to compute the maximal elements of $\Pi_\mu(S_p)$, given the partial order: $\mu^1 \succeq \mu^2$ if $\forall i \in [\![1; n]\!] \mu_i^1 \neq 0 \rightarrow \mu_i^2 \neq 0$.

Example 1. For app/3, let $\mu(x, y, z) = ax + by + cz$. We have $\Pi_\mu(S_p) = \{a + c \geq 1\}$. There are two maximal elements: $\mu^1(x, y, z) = x$ and $\mu^2(x, y, z) = z$.

In some sense, given a model for a program, this extension is complete. But a more precise model can lead to more maximal elements. Hence the precision of the inferred CLP model is important. From an implementation point of view, this algorithm heavily relies on the costly projection operator. We found that a good strategy is to project constraints as soon as possible.

6 Conclusion

We have presented the main algorithms of cTI, our bottom-up left-termination inference tool for logic programs and given some running for standard LP termination programs and middle-sized logic programs. The analysis requires three fixpoint computations and the inference of well-founded orders. We have described some optimizations and measured their impacts.

We have compared the quality of the results obtained by cTI with three other top-down termination checkers. Our termination inference tool is able in all cases (although we manually tuned cTI four times) to infer a larger class of terminating queries. On the other hand, the running times of cTI are also more important, but termination inference is a much more general problem than termination checking. In the worst case, an exponential number of termination checks are needed to simulate termination inference.

Right now, cTI can not directly infer termination for some programs, e.g. CHAT, as suggested by P. Tarau. A more detailed look to this program written by F.C.N. Pereira and D.H.D. Warren a shows that it contains one scc of 30 mutually recursive predicate symbols with 8 arguments per predicate symbol on the average. We cannot compute a numeric model for CHAT using the constraint solver of SICStus Prolog (the CLP($\mathbb{Q}$) solver is itself written in SICStus Prolog) in reasonable time. So we add for each computation which may be too costly (see also [9]) a timeout and if necessary we are able to return a value which does not destroy the correctness of the analysis (this is another widening!). The point is that the theoretical framework [31] only requires to have a CLP($\mathbb{N}$) model and an upper approximation of the CLP($\mathcal{B}$) least model. The drawback of this approach is that, in such a case, the quality of the inference is poorer. As a side effect, *the running time of cTI is now linear with respect to the number of sccs in the call graph*. We point out that CHAT is the only natural example we know which requires such a mechanism, although F. Henderson notified us of a similar scc in the code of the Mercury compiler [39]. Finally, Table 4 points out that most ($> 75\%$) of the analysis time lies in numeric computations. Hence we plan to link SICStus Prolog with specialized C libraries (e.g. a tool for polyhedra manipulations and a simplex solver optimized wrt to projection).

We are also developing another line of research where we try to prove the *optimality* of the termination conditions computed by cTI. Instead of looking for general classes of logic programs for which the analysis is complete, we try, for each particular (pure) logic program, to prove that the termination condition derived by cTI is as general as it can be (modulo the language describing the termination conditions). We have already implemented the analysis (called nTI for *non-Termination Inference*, available at the same URL than cTI) and its formalization is in progress.

Acknowledgements

We thank the readers of this paper for their constructive comments.

References

1. K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and computation*, 1(106):109–157, 1993.
2. K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.
3. K.R. Apt. *From Logic Programming to Prolog*. Prentice Hall, 1997.
4. 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.
5. B. De Backer and H. Beringer. A clp language handling disjunctions of linear constraints. In *Proc. of ICLP'93*, pages 550–563. MIT Press, 1993.
6. F. Benoy and A. King. Inferring argument size relationships with CLP(R). In J. P. Gallagher, editor, *Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
7. F. Bueno, M. Garcia de la Banda, and M. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic program parallelization. In *Proceedings of the 1994 International Symposium on Logic Programming*, pages 320–336. MIT Press, 1994.
8. M. Carlsson. Boolean constraints in SICStus Prolog. Technical Report T91:09, Swedish Institute of Computer Science, 1994.
9. M. Codish. Worst-case groundness analysis using positive boolean functions. *Journal of Logic Programming*, x:yyy–zzz, 1999.
10. M. Codish and C. Taboch. A semantics basis for termination analysis of logic programs. *Journal of Logic Programming*, 41:103–123, 1999.
11. S. Colin, F. Mesnard, and A. Rauzy. Constraint logic programming and mu-calculus. *ERCIM/COMPULOG Workshop on Constraints*, 1997.
12. P. Cousot and R. Cousot. Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the fourth Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
13. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2,3):103–179, 1992.
14. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Lecture Notes of Computer Science*, volume 631. Springer-Verlag, 1992. Proceedings of PLILP'92.
15. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the fifth Symposium on Principles of Programming Languages*, pages 84–96. ACM, 1978.
16. S. Decorte. *Enhancing the power of termination analysis of logic programs through types and constraints*. PhD thesis, Katholieke Universiteit Leuven, 1997.
17. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 21(6):1136–1195, 1999.
18. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The standard, reference manuel*. Springer-Verlag, 1996.
19. 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.
20. D. DeSchreye and S. Decorte. Termination of logic programs : the never-ending story. *Journal of Logic Programming*, 19-20:199–260, 1994.

21. S. Hoarau. *Inférer et compiler la terminaison des programmes logiques avec contraintes*. PhD thesis, Université de La Réunion, 1999.
22. C. Holzbaur. Ofai clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute, 1995.
23. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.
24. J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19:503–581, 1994.
25. G. Levi and F. Scozzari. Contributions to a theory of existential termination for definite logic programs. In M. Alpuente and M. I. Sessa, editors, *Proceedings of the GULP-PRODE'95 Joint Conference on Declarative Programming*, pages 631–641, 1995.
26. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proceedings of the 14th International Conference on Logic Programming*, pages 63–77. MIT Press, 1997.
27. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
28. M. Marchiori. Proving existential termination of normal logic programs. In *Proceedings of the 1996 AMAST*, pages xxx–yyy, 1996.
29. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs by means of approximations. In M. J. Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.
30. F. Mesnard. Entailment and projection for CLP($\mathcal{B}$) and CLP($\mathbb{Q}$) in SICStus Prolog. *1st International Workshop on Constraint Reasoning for Constraint Programming*, 1997.
31. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. Technical report, Université de La Réunion, 2001.
32. 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/`.
33. L. Plümer. Termination proofs for logic programs. *Lecture Notes in Artificial Intelligence*, 446, 1990.
34. S. Ruggieri. *Verification and Validation of Logic Programs*. PhD thesis, Università di Pisa, 1999.
35. H. Sağlam. *A Toolkit for Static Analysis of Constraint Logic Programs*. PhD thesis, University of Bristol, 1997.
36. H. Sağlam and J. Gallagher. Static analysis of logic programs using CLP as a meta-language. Technical Report CSTR-96-003, University of Bristol, Department of Computer Science, 1996.
37. A. Schrijver. *Theory of linear and integer programming*. John Wiley and Sons, 1986.
38. K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings of the 1991 International Symposium on Principles of Database Systems*, pages 216–226. ACM, 1991.
39. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative Logic Programming language. *The Journal of Logic Programming*, 29(1–3):17–64, 1996.
40. C. Speirs, Z. Somogyi, and H. Søndergaard. Termination analysis for Mercury. In P. van Hentenrick, editor, *Proceedings of the 1997 International Symposium*

on Static Analysis, volume 1302 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.
41. J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Communications of the ACM*, pages 345–373, 1988.
42. T. Vasak and J. Potter. Characterization of terminating logic programs. In *Proceedings of the 1986 International Symposium on Logic Programming*, pages 140–147. IEEE, 1986.
43. K. Verschaetse. *Static termination analysis for definite Horn clause programs.* PhD thesis, Dept. Computer Science, K.U. Leuven, 1992.