NTI+cTI: a Logic Programming Termination Analyzer

Fred Mesnard Étienne Payet

LIM, université de La Réunion

2023

We describe NTI+cTI, our logic programming termination analyzer that takes part in the Termination Competition 2023.

The tool is built from two separate components, NTI for *Non-Termination Inference* and cTI for *constraint-based Termination Inference*, plus an overall main process.

NTI

NTI is fully written in Java. It performs automated non-termination proofs of logic programs. It implements a technique that consists in unfolding the program under analysis and in checking whether the produced unfolded clauses satisfy some non-termination criteria. When a proof is successful, NTI provides an example of a non-terminating query.

- `https://github.com/etiennepayet/nti`
- `http://lim.univ-reunion.fr/staff/epayet/Research/NTI/NTI.html`

NTI

Two kinds of criteria are used.

- The first kind relies on an extension of the “is more general than” relation. It is able to detect infinite derivations that consist of the repeated application of the same sequence ω of clauses, *i.e.*, of the form $Q_0 \Rightarrow_\omega Q_1 \Rightarrow_\omega \cdots$. If the body of an unfolded clause is more general than the head up to some predicate arguments in *neutral position*, then non-termination is detected; more precisely, every query obtained from replacing the neutral arguments of the head with ground terms is non-terminating. So, if such a non-terminating query belongs to the mode of interest then the proof is successful.

NTI

```
%query: p(o,i).
p(s(X), Y) :- p(X, s(Y)).
```

```
$ java -jar nti.jar -v loop.pl
NO

** BEGIN proof argument **
p(o,i): p(s(_0),a) starts a single loop (extracted from a looping pair)
** END proof argument **

** BEGIN proof description **
* Iteration = 0: 0 unfolded rule generated, 0 witness generated
* Iteration = 1: 1 new unfolded rule(s) generated, 3 new witness(es) generated
The mode p(o,i) starts a single loop because of the generated witness:
  Looping pair: binseq = [p(s(_0),_1) :- p(_0,s(_1))],
                         DN set of positions = {p/2 -> {1->_0}}
* All specified modes do not terminate: p(o,i)
Proof run on Mac OS X version 10.16 for x86_64
using Java version 12.0.1
** END proof description **
Total number of generated unfolded rules = 1
```

NTI

- The second kind is able to detect infinite derivations that rely on two sequences ω_1 and ω_2 of clauses, *i.e.*, that have the form $Q_0(\Rightarrow_{\omega_1}^* \circ \Rightarrow_{\omega_2}) Q_1(\Rightarrow_{\omega_1}^* \circ \Rightarrow_{\omega_2}) \cdots$. It consists in detecting pairs (c_1, c_2) of unfolded clauses of a particular form. Intuitively, c_1 and c_2 are mutually recursive and, in c_1, a context is removed from the head to the body while, in c_2, it is added again.

NTI

```
%query: p(i,i).
p(0, Y) :- p(s(Y), s(Y)).
p(s(X), Y) :- p(X, Y).
```

```
$ java -jar nti.jar -v nonloop.pl
NO

** BEGIN proof argument **
p(i,i): p(s(0),s(0)) starts a binary loop (extracted from a recurrent pair)
** END proof argument **

** BEGIN proof description **
* Iteration = 0: 0 unfolded rule generated, 0 witness generated
* Iteration = 1: 2 new unfolded rule(s) generated, 7 new witness(es) generated
The mode p(i,i) starts a binary loop because of the generated witness:
  Recurrent pair: <[p(s(_0),_1) :- p(_0,_1), p(0,_2) :- p(s(_2),s(_2))],
                   (i,j) = (1,0), c = s(□), s = 0, t = _2, (n1,n2,n3) = (0,1,1)>
* All specified modes do not terminate: p(i,i)
Proof run on Mac OS X version 10.16 for x86_64
using Java version 12.0.1
** END proof description **
Total number of generated unfolded rules = 2
```

cTI

- `https://github.com/FredMesnard/cTI`

The termination analyzers are written in SWI-Prolog. The analysis starts with applying *termination inference*.

If the mode given in the moded query of interest of the analyzed program implies the inferred termination condition, termination is ensured.

This first analysis relies on the *term-size* norm to abstract the logic program and on linear ranking functions. If necessary, termination inference is restarted using the same tool but by combining both the term-size norm and the *list-size* norm.

cTI

Otherwise, we switch to BINTERM, the termination analyzer we've built for Java bytecode termination analysis, with various termination tests:

- linear and eventual ranking functions
- multi-dimensional linear ranking
- the size-change principle

BINTERM takes as input binary Constrained Horn Clauses.

cTI

Mapping the original moded query and the original logic program to binary Constrained Horn Clauses:

- Mode inference: The original logic program goes through a tabled left-to-right top-down mode analysis starting from the original moded query of interest.
- An abstract numeric constraint logic program is built using the term-size norm.
- A numerical model is computed.

Then a binary Constrained Horn Clauses program is created by a tabled left-to-right top-down interpreter, which keeps only the input arguments of the predicates.
If necessary, a similar analysis is done by combining both the term-size and the list-size norms.

cTI

At last resort, a left-to-right top-down meta-interpreter computes a (time-bounded) LD tree for *the most general* query $:-p(X_1,\ldots,X_n)$.

If the LD-tree is completely built, then *any* query from the set of concrete queries abstracted by the original moded query universally left-terminates.

NB:

- Unification with occurs check is mandatory
- A sound termination test only for logic programs

Examples: BCGGV05/g.pl, SGST06/at.pl, SGST06/toyama.pl, lpexamples/lategen.pl, talp_plumer/pl2.3.1.pl, talp_talp/transitive_closure.pl, ...

cTI

Unification with occurs check is mandatory

Consider:

```
go :- p(X,f(X)).
p(Y,Y) :- go.
```

The query `:- go.` loops for most Prolog engines but should fail, as `p(X,f(X))` does *not* unify with the head `p(Y,Y)` of the second rule, due to the occurs check.

cTI

A sound termination test only for logic programs

Consider:

```
p(X) :- var(X), !.
p(a) :- p(a).
```

For any Prolog engine:

- the query `:-p(X).` terminates
- the query `:-p(a).` does *not* terminate.

The theoretical LP framework does not apply to Prolog.

The main process

Non-termination and termination analyses in parallel:

- a thread that runs NTI
- a thread that runs cTI

If one thread terminates successfully then the other is stopped and the result is reported.