KU LEUVEN

Proof[chapter]

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Termination and Non-termination in Logic Programming

Dean Voets

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2013

Termination and Non-termination in Logic Programming

Dean VOETS

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

December 2013

ISBN 978-94-6018-777-3

D/2013/7515/158

Dankwoord

Dit proefschrift, *Termination and Non-termination in Logic Programming*, bevat de belangrijkste resultaten van mijn onderzoek naar de analyze van eindigheid en oneindigheid in Logische Programmeertalen. Dit werk werd uitgevoerd binnen de groep Declaratieve Talen en Artificiële Intelligentie van het Departement Computerwetenschappen aan de Katholieke Universiteit te Leuven. Zonder hulp had dit werk niet tot stand kunnen komen. Het is dan ook vanzelfsprekend dat ik begin met het bedanken van diegenen die de grootste steun zijn geweest om mijn doctoraat tot een goed einde te brengen.

Eerst en vooral zou ik de promotor van mijn doctoraat, Prof. Dr. Danny De Schreye, willen bedanken. Ten eerste voor het gestelde vertrouwen zodat ik aan mijn doctoraat heb mogen beginnen. Ten tweede voor zijn uitstekende begeleiding, waardoor ik veel heb bijgeleerd over communicatie, presentatie en ook de formalisatie van de bestudeerde onderwerpen.

I would like to thank the members of the jury for reading and evaluating my PhD. Your remarks have been most useful to obtain this final version of my thesis. I would like to thank the chairman of my jury Prof. P. Van Houtte, my promotor Prof. D. De Schreye, my assessors Prof. F. Piessens and Prof. M. Bruynooghe and the external jury members Prof. W. Vanhoof and Prof. F. Mesnard.

Vervolgens zou ik graag mijn collega Dr. Paolo Pilozzi bedanken voor zijn steun en vriendschap tijdens ons onderzoek aan de Katholieke Universiteit Leuven en voor de inspiratie die uit onze gesprekken voortvloeide door zijn eigen kijk op de zaken.

Tenslotte zou ik graag mijn ouders, Jan Voets en Leen Van de Weerd, bedanken voor hun vertrouwen en jarenlange steun. Het is door jullie dat ik ben kunnen uitgroeien tot de persoon die ik vandaag ben en daar ben ik jullie enorm dankbaar voor.

Abstract

One of the central concerns of declarative programming is that it leads to less error-prone, more understandable and better maintainable programs. However, it is well-known that a declarative programming style also results in less efficient computations, and in the extreme case, in non-terminating computations. The latter problem has received considerable attention within the community. Much research has been done on termination analysis, loop detection and more recently, non-termination analysis.

Due to the nature of undecidability, there must be situations in which neither a termination proof nor a non-termination proof can apply; i.e., no sufficient termination/non-termination conditions are satisfied so that the user would get no conclusion. We observe that in such a situation, it is particularly useful to compute an approximate conclusion indicating possible termination or possible non-termination. To the best of our knowledge, however, no existing approximation approach was available before our work.

A first contribution of the thesis is the development of such an approximation approach called termination prediction. In the case that neither a termination nor a non-termination proof is applicable, we appeal to an approximation algorithm to predict possible termination or non-termination. The analysis constructs a finite symbolic derivation tree, representing the derivation for a class of queries. The termination behavior is then predicted by checking properties of this tree.

A second contribution of the thesis is a new non-termination analysis for logic programs. In the thesis, we define a non-termination analysis based on the symbolic derivation trees from the termination prediction approach. We show that this non-termination analysis improves on the results of the only non-termination analyzer developed before our work. We extend our non-termination analysis in several ways. Type information and use program specialization is incorporated to obtain a stronger non-termination analysis. Another extension we discuss, is an extension to handle programs using integer arithmetics. We implemented this non-termination analysis and we show its applicability on a benchmark of logic programs.

A final contribution of the thesis is the development of a termination analysis for the programming language Constraint Handling Rules (CHR). This termination analysis is the first approach without restrictions on the type of rules in the CHR program. We demonstrate the condition's applicability on a set of terminating CHR programs, using a prototype analyzer. This analyzer is the first in-language automated termination analyzer for CHR programs.

Beknopte samenvatting

Een van de belangrijkste voordelen van declaratieve programmeertalen is dat het toelaat om beknopte, verstaanbare programmas te schrijven die makkelijk te onderhouden zijn en minder gevoelig zijn fouten. Het is echter ook geweten dat zo een declaratieve stijl van programmmeren kan leiden tot onefficiënte programmas en in het ergste geval zelfs tot oneindige berekeningen. Dit probleem heeft veel aandacht gekregen in de onderzoeksgemeenschap. Er is veel onderzoek gebeurd naar technieken om eindigheid voor programmas te bewijzen (terminatie analyze), technieken om lussen te detecteren tijdens de uitvoering (lus detectie) en meer recent, technieken om oneindigheid van programmas te bewijzen (non-terminatie analyze).

Doordat de analyze van eindigheid voor programmeertalen onbeslisbaar is, moeten er situaties zijn waarbij noch de eindigheid, noch de oneindigheid van een programma bewezen kan worden. In zo een geval krijgt de gebruiker dus geen conclusie. Daarom is het volgens ons nuttig om een benaderende analyze uit te werken. Wanneer exacte terminatie en non-terminatie technieken geen conclusie geven, kan de benaderende techniek gebruikt worden om een afschatting van de eindigheid van het programma te maken. Deze benadering kan dan helpen om het programma verder aan te passen totdat een traditionele analyze de eindigheid of oneindigheid kan bewijzen.

Een eerste bijdrage van de thesis is de ontwikkeling van zo een benaderende analyze genoemd *eindigheids voorspelling*. Wanneer de eindigheid van een programma niet exact kan worden bepaald, gebruiken we een algoritme om een benaderend antwoord te zoeken. De analyze stelt een eindige, symbolische boom op die de berekening voor een klasse van queries voorstelt. Vervolgens worden er eigenschappen van deze symbolische boom onderzocht, om zo een benaderende conclusie over het eindigheid gedrag van deze queries te bekomen.

Een tweede bijdrage van de thesis is de ontwikkeling van een nieuwe non-terminatie analyze. In de thesis definiëren we een nieuwe analyze die de symbolische boom van de eindigheids voorspelling zal gebruiken om te bewijzen dat de uitvoering van een programma niet zal eindigen. We tonen aan dat onze analyze preciezer is dan de enige andere geautomatiseerde non-terminatie analyze. We breiden deze techniek op verschillende manieren uit. Type informatie en specializatie wordt gebruikt om een betere analyze te verkrijgen. Daarnaast bespreken we een uitbreiding van deze techniek die toelaat om programmas te bestuderen die rekenkundige predikaten bevatten.

Een laatste bijdrage die in de thesis wordt besproken is de ontwikkeling van een terminatie analyze voor de programmeertaal Constraint Handling Rules (CHR). Deze terminatie analyze is de eerste eindigheids analyze die geen beperking legt op het soort regels dat gebruikt mag worden in het programma. We tonen aan dat de analyze van toepassing is op een heel aantal eindigende CHR programmas, met behulp van een prototype analyzer. Deze analyzer is de eerste terminatie analyzer voor CHR die de programmas rechtstreeks, dus zonder vertaling naar een andere programmeertaal, onderzoekt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this Chapter, we summarize the works on termination and non-termination analysis in Logic Programs and explain how they motivate our research. In Section 1.2, we give an overview of the thesis.

1.1 The Termination Problem in Logic Programming

One of the central concerns of declarative programming, in particular of Logic Programming, is that the use of a declarative programming style in a declarative programming language leads to less error-prone, more understandable and better maintainable programs. However, it is well-known that a declarative programming style also results in less efficient computations, and in the extreme case, in non-terminating computations. The latter problem has received considerable attention within the community. Much research has been done on termination analysis, loop detection and more recently, non-termination analysis.

Among these areas, termination analysis has by far received most attention. [16] presents a survey of the extensive amount of work up till 1994. However, most of the more powerful approaches and techniques have been introduced in the last decade: the constrained-based approach to termination analysis [18], the local approaches [15], the use of types in termination analysis [10], powerful transformational approaches [46], termination inference [33], and the porting of TRS-techniques to the LP-context [35].

A rather recent concern in this research is the precision of the termination analysis. Since termination is undecidable in general, only sufficient conditions for termination are verified. It is important to have a good understanding of the precision of these techniques: do they actually capture most of the terminating computations?

With respect to the other two approaches, loop detection and non-termination analysis, there is often confusion concerning their relation. Because both approaches use similar techniques, their distinguishing features and aims are not always well understood. Loop detection is a run-time technique. It aims to cut infinite derivations for a concrete query at run-time. One of the possible approaches to achieve this is tabulation [43]. For an extensive overview and comparison of different loop checking algorithms, we refer to [7]. Non-termination analysis is a compile-time approach. It aims to prove that a certain class of queries will result in non-terminating computations for at least some of the queries in the considered class. Non-termination analysis is performed for classes of queries described in terms of modes (or types). One of the key concerns of non-termination analysis is to address the important issue of precision analysis of termination analysis. A termination analysis can be shown to be precise by proving that the class of queries for which termination could not be proven is actually non-terminating. This has been one of the main goals and achievements of the first non-termination analyzer for Logic Programs, *NTI*[38].

Due to the nature of undecidability, there must be situations in which neither a termination proof nor a non-termination proof can apply; i.e., no sufficient termination/non-termination conditions are satisfied so that the user would get no conclusion (see the results of the Termination Competition 2007 [1]). We observe that in such a situation, it is particularly useful to compute an approximate conclusion indicating possible termination or possible non-termination, which guides the user to continue to improve his program towards termination. To the best of our knowledge, however, no existing approximation approach was available before our work.

1.2 Overview and goals of the Thesis

Chapter 2 introduces the basic notions of Logic Programming and loop checking, which are essential for understanding the thesis.

Chapter 3 proposes an approximation approach to termination analysis, called *termination prediction*. In the case that neither a termination nor a non-termination proof is applicable, we appeal to an approximation algorithm to predict possible termination or non-termination. We develop a framework for predicting termination of general logic programs with concrete queries or *moded* queries. Moded queries allow to represent arbitrary ground terms by input modes. For example $flat(\mathcal{I}, O)$, with $\mathcal{I}$ an input mode, represents all $flat$ queries with a ground term as a first argument and a free variable as second argument. The basic idea of our analysis is that we establish a characterization of infinite (generalized) SLDNF-derivations with arbitrary queries. Then based on the characterization, we design a complete loop checking mechanism, which cuts all infinite SLDNF-derivations. Given a logic program and a query, we evaluate the query by applying SLDNF-resolution while performing loop checking. If the query evaluation proceeds without encountering potential infinite derivations, we predict *terminating* for this query; otherwise we predict *non-terminating*. It is nontrivial to characterize infinite SLDNF-derivations with moded queries. The first challenge we must address is how to formulate an SLDNF-derivation for a moded query Q_0, as the standard SLDNF-resolution is only for concrete queries [14, 28]. We will introduce a framework called a *moded-query forest*, which consists of all (generalized) SLDNF-trees rooted at an instance of Q_0 (the instance is Q_0 with each input mode replaced by a ground term). An SLDNF-derivation for Q_0 is then defined over the moded-query forest such that a logic program P terminates for Q_0 if and only if the moded-query forest contains no infinite SLDNF-derivations. A moded-query forest may have an infinite number of SLDNF-trees, so it is infeasible for us to predict termination of a logic program by traversing the moded-query forest. To handle this challenge, we will introduce a novel compact approximation for a moded-query forest, called a *moded generalized SLDNF-tree*. The key idea is to treat an input mode as a special meta-variable in the way that during query evaluation, it can be substituted by a constant or function, but cannot be substituted by an ordinary variable. As a result, SLDNF-derivations for a moded query can be constructed in the same way as the ones for a concrete query. A characterization of infinite SLDNF-derivations for moded queries is then established in terms of some key properties of a moded generalized SLDNF-tree. We have implemented a termination prediction system and obtained quite satisfactory experimental results. Our prediction is 100% correct for all benchmark programs of the Termination Competition 2007, of which eighteen programs cannot be proved by the existing state-of-the-art analyzers like $AProVE07$, NTI, *Polytool* and $TALP$.

Chapter 4 introduces a new non-termination analysis. It reuses the analysis scheme proposed in Chapter 3 to produce a finite representation of the computation for a moded query, given some logic program. We introduce a new non-termination condition expressed in terms of this finite representation of the computation.

Example 1.1.

```
flat(niltree, nil).
flat(tree(X, niltree, XS), cons(X, YS)) :- flat(XS, YS).
flat(tree(X, tree(Y, YS1, YS2), XS), ZS) :-
        flat(tree(Y, YS1, tree(X, YS2, XS)), ZS).
```

This program, $flat$, flattens a binary tree into a list denoted with the cons notation. To flatten the tree, the program repeatedly moves one element from the left to the right subtree until the left subtree is empty. When the left subtree is empty, we proceed by processing the right subtree. If the first argument of the query is a variable, this program loops w.r.t. the third clause. □

This non-termination condition detects that a sequence of derivation steps can be repeated infinitely for a class of queries. For the $flat$ program in Example 1.1, it will prove that the last clause can be repeated infinitely for queries with a variable as a first argument. We prove the correctness of the condition and extend it to increase its applicability. It turns out that our characterization of non-terminating computations is more precise than that of NTI. We have implemented the technique in the analyzer $P2P$ and performed extensive experiments with it on the basis of the benchmark of the termination analysis competition of 2007[1]. The experiments show that our technique has a 100% success-rate on this benchmark, outperforming the only competing approach, NTI.

Although we experienced this as a success, the experiment mostly shows that the benchmark does not offer sufficient challenges for non-termination analysis. Chapter 5 focuses on two new directions. One is to identify classes of programs for which non-termination analyzers developed before 2011 fail. A second is to investigate whether the inclusion of type-information, in addition to modes, may improve the power of our analyzer. Considering the first of these questions, a limitation of both NTI and $P2P$ is that they only detect non-terminating derivations if, within these derivations, some fixed sequence of clauses can be applied repeatedly. Example 1.2 shows a program that violates this restriction.

Example 1.2. *The program, longer, loops for any query* `longer(L)`*, with* `L` *a non-empty list of zeros. The predicate $zeros/1$ checks if the list contains only zeros. At the recursive call, a zero is added to the list.*

[1] http://www.lri.fr/~marche/termination-competition/

```
longer([0|L]):-
    zeros(L),
    longer([0,0|L]).
zeros([]).
zeros([0|L]):- zeros(L).
```

*The list in the recursive call is longer than the original one and thus, the number of applications of the recursive clause for zeros/*1 *increases in each recursion. Therefore, no fixed sequence of clauses can be repeated infinitely and previous non-termination analyzers fail to prove non-termination of this example.* □

In Chapter 5, we overcome this limitation by using non-failure information. *Non-failure analysis* [17] detects classes of goals that can be guaranteed not to fail, given mode and type information. Its applications include inferring minimal computational costs, guiding transformations and debugging. To use the information provided by non-failure analysis in the non-termination analysis of [58], type information must be added to the symbolic derivation tree. We add this information using regular types [60]. Extending our non-termination analysis with non-failure information allows to prove non-termination for programs such as the one in Example 1.2. Another limitation of previous non-termination analyzers is related to aliased variables. Non-termination analyzers developed before 2011 fail to prove non-termination of programs such as the one in Example 1.3.

Example 1.3.

```
append([],L,L).
append([H|T],L,[H|R]):- append(T,L,R).
```

The query `append(X,X,X)` *succeeds once with a computed answer substitution* `X/[]`*. The program loops after backtracking.* □

Program specialization [26] allows to transform this program and query into an equivalent program and query for which non-termination can be proven. In Chapter 5, we extend the technique of Chapter 4 with type information. Then, we use non-failure analysis and specialization to deal with the classes of programs illustrated by the examples above. In addition to these two classes of programs, there are combinations of them that yield a fairly large class of new programs that we can prove non-terminating using non-failure analysis combined with program specialization.

Both termination and non-termination analyzers have been rather successful in analyzing the termination behavior of definite logic programs, but their applicability on real-life Prolog programs is limited because most Prolog programs use non-logical features. Chapter 6 takes a first step towards the analysis of real-life Prolog programs, by presenting a non-termination condition for Logic Programs containing integer arithmetics. Given a program, containing integer arithmetics, and a class of queries, we infer a subset of these queries for which we prove existential non-termination. The inference and proof are done in two phases. In the first phase, non-termination of the logic part of the program is proven by assuming that all comparisons between integer expressions succeed. In the second phase, given the moded query, integer arguments are identified and constraints over these arguments are formulated, such that solutions for these constraints correspond to non-terminating queries. We will illustrate this with an example.

Example 1.4. *The following program, count_to, is a faulty implementation of a predicate generating the list starting from 0 up to a given number. The considered class of queries is represented by the moded query* $\leftarrow count\_to(\underline{N}, L)$ *with* $\underline{N}$ *an integer variable and* L *a free variable.*

```
count_to(N,L):- count(0,N,L).
count(N,N,[N]).
count(M,N,[M|L]):- M > N, M1 is M+1, count(M1,N,L).
```

In the last clause, the integer condition should be `M < N` *instead of* `M > N`*. Due to this error, the program:*

- *fails for the queries for which* $\underline{N} > 0$ *holds,*
- *succeeds for* $\leftarrow count\_to(0, L)$*,*
- *loops for the queries for which* $\underline{N} < 0$ *holds.*

The first phase of the analysis detects non-termination assuming the arithmetic predicates succeed. For the considered class of queries count is always called with a free variable as the third argument. Therefore, any considered query loops w.r.t. the last clause if the arithmetic calls succeed.

The second phase of the analysis completes the non-termination proof by inferring the constraint $\underline{N} < 0$ *on the considered class of queries.* □

Chapter 7 discusses the less related topic of Termination analysis of Constraint Handling Rules (CHR). CHR was designed to develop custom constraint solvers but proved successful as a general purpose programming language. The language rewrites multisets of constraints, the constraint store, and defines two types of rules. Simplification rules replace constraints by equivalent, simpler constraints. Propagation rules add extra constraints to the store. Because propagation rules do not remove constraints, a propagation history is used to prevent a combination of constraints of firing a propagation rule more than once. Chapter 7 presents the first termination condition for CHR with propagation rules.

Finally, Chapter 8 concludes the thesis.

Chapter 2

Preliminaries

In this chapter, we present the basic concepts and definitions of logic programming and loop checking that are necessary for understanding the thesis. Definitions and concepts that are specific to a chapter will be introduced in that chapter.

2.1 Logic Programming

Logic programming (LP) is a programming paradigm based on *first order logic* [28]. In our research, we focus on general logic programming as implemented by Prolog. A logic program consists of an alphabet, a first order language, a set of axioms and a set of inference rules. The alphabet includes sets of *variables*, *constants*, *function symbols*, *predicate symbols* and *connectives.* To denote variables, we use uppercase characters, e.g., X, Y, Z. Usually, function symbols are denoted f, g, h, and predicate symbols are denoted p, q, r. Each function or predicate symbol f is associated with a natural number n, the arity of f . We often write f/n to denote that a function or a predicate symbol f (a functor) has arity n. Constant symbols are function symbols with arity 0, denoted by lowercase letters, e.g., a, b, c. Connectives are $\leftarrow$ (implication) and $\wedge$ (conjunction).

Terms and *atoms*, two basic components of logic programming, are defined based on variables, constants, function and predicate symbols. A term is either a variable, a constant, or $f(t_1, \ldots, t_n)$ if $t_1, \ldots, t_n$ are terms and f/n is a function symbol. For simplicity, we use $\overline{T}$ to denote a set of terms $T_1, ..., T_m$. Lists are commonly used terms. A list is of the form [] or $[T|L]$ where T is a term and L is a list. For our purpose, the symbols [,] and | in a list are treated as special constant symbols. Similarly, $p(t_1, \ldots, t_n)$ is an atom if p/n is a predicate symbol and $t_1, \ldots, t_n$ are terms. Let A be an atom/term. The size of A, denoted $|A|$, is the number of occurrences of function symbols, variables and constants in A. Two atoms are called *variants* if they are the same up to variable renaming. A literal is an atom A or the negation $\neg A$ of A. A term or an atom, that is variable free, is called *ground*. Otherwise, it is called *non-ground*. For a term or atom t, we use the notation $Var(t)$ to denote the set of all variables occurring in t.

A *definite logic program* P is a finite set of clauses of the form $A \leftarrow A_1, ..., A_n$, where A and each A_i is an atom. A *general logic program* P is a finite set of clauses of the form $A \leftarrow L_1, ..., L_n$, where A is an atom and each L_i is a literal. Throughout the thesis, we consider only Herbrand models [28]. The Herbrand universe and Herbrand base of P are denoted by $HU(P)$ and $HB(P)$, respectively.

A *goal* G_i is a headless clause $\leftarrow L_1, ..., L_n$ where each literal L_j is called a *subgoal*. The goal, $G_0 = \leftarrow Q_0$, for a query Q_0 is called a *top goal*. Without loss of generality, we assume that Q_0 consists only of one atom. Q_0 is a *moded query* if some arguments of Q_0 are input modes (in this case, Q_0 is called an *abstract* atom or a *moded* atom); otherwise, it is a *concrete query*. An input mode always begins with a letter $\mathcal{I}$ and corresponds to a ground term. For more information about mode analysis in logic programming, we refer to [29].

Let P be a logic program and G_0 a top goal. G_0 is evaluated by building a *generalized SLDNF-tree* GT_{G_0} as defined in [51], in which each node is represented by $N_i : G_i$ where N_i is the name of the node and G_i is a goal attached to the node. We do not reproduce the definition of a generalized SLDNF-tree. Roughly speaking, GT_{G_0} is the set of standard SLDNF-trees for $P \cup \{G_0\}$ augmented with an ancestor-descendant relation on their subgoals. Let L_i and L_j be the selected subgoals at two nodes N_i and N_j, respectively. L_i is an *ancestor* of L_j, denoted $L_i \prec_{anc} L_j$, if L_j is selected as a subgoal in the proof of L_i. Throughout the thesis, we choose to use the best-known *depth-first, left-most* control strategy, as is used in Prolog, to select nodes/goals and subgoals (it can be adapted to any other fixed control strategies). So by the *selected subgoal* in each node $N_i : \leftarrow L_1, ..., L_n$, we refer to the left-most subgoal L_1.

Recall that in SLDNF-resolution, let $L_i = \neg A$ be a ground negative subgoal selected at N_i, then (by the negation-as-failure rule [14]) a subsidiary child SLDNF-tree $T_{N_{i+1}:\leftarrow A}$ rooted at $N_{i+1} : \leftarrow A$ will be built to solve A. In a generalized SLDNF-tree GT_{G_0}, such parent and child SLDNF-trees are connected from N_i to N_{i+1} via a dotted edge "$\cdots\triangleright$" (called a *negation arc*), and A at N_{i+1} inherits all ancestors of L_i at N_i. Therefore, a path of a generalized SLDNF-tree may come across several SLDNF-trees through dotted edges. Any such a path starting at the root node $N_0 : G_0$ of GT_{G_0} is called a *generalized SLDNF-derivation.*

We do not consider *floundering* queries; i.e., we assume that no non-ground negative subgoals are selected at any node of a generalized SLDNF-tree (see [51]).

Another feature of a generalized SLDNF-tree GT_{G_0} is that each subsidiary child SLDNF-tree $T_{N_{i+1}:\leftarrow A}$ in GT_{G_0} terminates (i.e. stops expanding its nodes) at the first success leaf. The intuition behind this is that it is absolutely unnecessary to exhaust the remaining branches because they would never generate any new answers for A (since A is ground). In fact, Prolog executes the same pruning by using a *cut* operator to skip the remaining branches once the first success leaf is generated (e.g. see SICStus Prolog [25]). To illustrate, consider the following logic program and top goal:

$$\begin{array}{lll} P_0 : & p \leftarrow \neg q. & C_{p_1} \\ & q. & C_{q_1} \\ & q \leftarrow q. & C_{q_2} \\ G_0 : & \leftarrow p. & \end{array}$$

The generalized SLDNF-tree GT_{G_0} for $P_0 \cup \{G_0\}$ is depicted in Figure 2.1. Note that the subsidiary child SLDNF-tree $T_{N_2:\leftarrow q}$ terminates at the first success leaf N_3, leaving N_4 not further expanded. As a result, all generalized SLDNF-derivations in GT_{G_0} are finite.

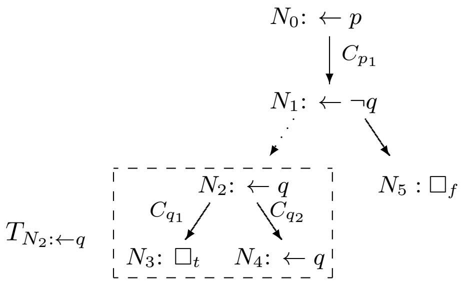


Figure 2.1: The generalized SLDNF-tree GT_{G_0} of P_0.

For simplicity, in the following sections by a derivation or SLDNF-derivation we refer to a generalized SLDNF-derivation. Moreover, for any node $N_i : G_i$ we use L_i^1 to refer to the selected subgoal in G_i.

A derivation step is denoted by $N_i : G_i \Rightarrow_{C,\theta_i} N_{i+1} : G_{i+1}$, meaning that applying a clause C to G_i produces $N_{i+1} : G_{i+1}$, where G_{i+1} is the resolvent of C and G_i on L_i^1 with the mgu (most general unifier) θ_i. Here, for a substitution of two variables, X in L_i^1 and Y in (the head of) C, we always use X to substitute for Y. When no confusion would occur, we may omit the mgu θ_i when writing a derivation step.

2.1.1 Loop Checking

Loop checking is a runtime approach to prevent non-termination during query evaluation. Non-termination is prevented by modifying the computation mechanism that searches through an SLDNF-tree by adding the possibility of pruning.

A loop checking mechanism, or more formally a *loop check* [8], defines conditions for us to cut a possibly infinite derivation at some node. By cutting a derivation at a node N we mean removing all descendants of N. Informally, a loop check is said to be *weakly sound* if for any generalized SLDNF-tree GT_{G_0}, GT_{G_0} having a success derivation before cut implies it has a success derivation after cut; it is said to be *complete* if it cuts all infinite derivations in GT_{G_0}. An ideal loop check cuts all infinite derivations while retaining success derivations. Unfortunately, as shown by Bol et al. [8], there exists no loop check that is both weakly sound and complete. In the thesis, we focus on complete loop checks, because we want to apply them to analyze the termination behavior of logic programs.

Definition 2.1. *A loop check L is* ***complete*** *w.r.t. SLDNF-resolution if for every logic program P and query Q, every infinite derivation of P for Q is cut by L.* $\square$

Chapter 3

Termination Prediction for General Logic Programs

This chapter proposes the idea of *termination prediction*, as depicted in Figure 3.1. In the case that neither a termination nor a non-termination proof is applicable, we appeal to an approximation algorithm to predict possible termination or non-termination. The prediction applies to general logic programs with concrete or moded queries.

We develop a framework for predicting termination of general logic programs with arbitrary (i.e., concrete or moded) queries. The basic idea is that we establish a characterization of infinite (generalized) SLDNF-derivations with arbitrary queries. Then based on the characterization, we design a complete loop checking mechanism, which cuts all infinite SLDNF-derivations. Given a logic program and a query, we evaluate the query by applying SLDNF-resolution while performing loop checking. If the query evaluation proceeds without encountering potential infinite derivations, we predict *terminating* for this query; otherwise we predict *non-terminating.*

The core of our termination prediction is a characterization of infinite SLDNF-derivations with arbitrary queries. In [51], a characterization is established for general logic programs with concrete queries. This is far from enough for termination prediction; a characterization of infinite SLDNF-derivations for moded queries is required. Moded queries are the most commonly used query form in static termination analysis. A moded query contains (abstract) atoms like $p(\mathcal{I}, T)$ where T is a term and $\mathcal{I}$ is an input mode. An *input mode* stands for an arbitrary ground term, so that to prove that a logic program terminates

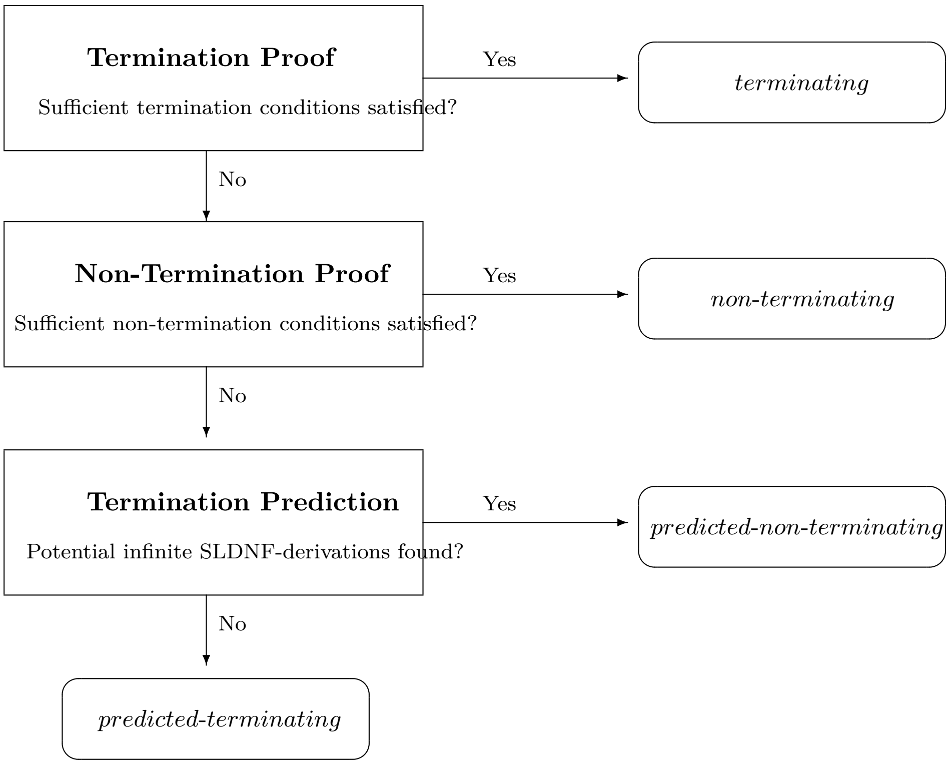

Termination Proof
Sufficient termination conditions satisfied?
Yes
terminating
No
Non-Termination Proof
Sufficient non-termination conditions satisfied?
Yes
non-terminating
No
Termination Prediction
Potential infinite SLDNF-derivations found?
Yes
predicted-non-terminating
No
predicted-terminating

Figure 3.1: A framework for handling the termination problem

for a moded query $p(\mathcal{I}, T)$ is to prove that the program terminates for any (concrete) query $p(t, T)$ where t is a ground term.

It is nontrivial to characterize infinite SLDNF-derivations with moded queries. The first challenge we must address is how to formulate an SLDNF-derivation for a moded query Q_0, as the standard SLDNF-resolution is only for concrete queries [14, 28]. We will introduce a framework called a *moded-query forest*, which consists of all (generalized) SLDNF-trees rooted at an instance of Q_0 (the instance is Q_0 with each input mode replaced by a ground term). An SLDNF-derivation for Q_0 is then defined over the moded-query forest such that a logic program P terminates for Q_0 if and only if the moded-query forest contains no infinite SLDNF-derivations.

A moded-query forest may have an infinite number of SLDNF-trees, so it is infeasible for us to predict termination of a logic program by traversing the moded-query forest. To handle this challenge, we will introduce a novel compact approximation for a moded-query forest, called a *moded generalized SLDNF-tree*. The key idea is to treat an input mode as a special meta-variable in the way that during query evaluation, it can be substituted by a constant or function, but cannot be substituted by an ordinary variable. As a result, SLDNF-derivations for a moded query can be constructed in the same way as the ones for a concrete query. A characterization of infinite SLDNF-derivations for moded queries is then established in terms of some key properties of a moded generalized SLDNF-tree.

We have implemented a termination prediction system and obtained quite satisfactory experimental results. Our prediction is 100% correct for all benchmark programs of the Termination Competition 2007, of which eighteen programs cannot be proved by the existing state-of-the-art analyzers like AProVE07, NTI, Polytool and TALP.

3.1 A Characterization of Infinite SLDNF-Derivations for Concrete Queries

In this section, we review the characterization of infinite derivations with concrete queries presented in [51].

Definition 3.1. Let T be a term or an atom and S be a string that consists of all predicate symbols, function symbols, constants and variables in T, which is obtained by reading these symbols sequentially from left to right. The *symbol string* of T, denoted S_T, is the string S with every variable replaced by $\mathcal{X}$. $\square$

For instance, let $T_1 = a$ and $T_2 = f(X, g(X, f(a, Y)))$. Then $S_{T_1} = a$ and $S_{T_2} = f\mathcal{X}g\mathcal{X}fa\mathcal{X}$.

Definition 3.2. Let S_{T_1} and S_{T_2} be two symbol strings. S_{T_1} is a *projection* of S_{T_2}, denoted $S_{T_1} \subseteq_{proj} S_{T_2}$, if S_{T_1} is obtained from S_{T_2} by removing zero or more elements. $\square$

Definition 3.3. Let A_1 and A_2 be two atoms (positive subgoals) with the same predicate symbol. A_1 is said to *loop into* A_2, denoted $A_1 \rightsquigarrow_{loop} A_2$, if $S_{A_1} \subseteq_{proj} S_{A_2}$. Let $N_i : G_i$ and $N_j : G_j$ be two nodes in a derivation with $L_i^1 \prec_{anc} L_j^1$ and $L_i^1 \rightsquigarrow_{loop} L_j^1$. Then G_j is called a *loop goal* of G_i. $\square$

Observe that if $A_1 \rightsquigarrow_{loop} A_2$ then $|A_1| \leq |A_2|$, and that if G_3 is a loop goal of G_2 that is a loop goal of G_1 then G_3 is a loop goal of G_1. Since a logic program has only a finite number of clauses, an infinite derivation results from repeatedly applying the same set of clauses, which leads to either infinite repetition of selected variant subgoals or infinite repetition of selected subgoals with recursive increase in term size. By recursive increase of term size of a subgoal A from a subgoal B we mean that A is B with a few function/constant/variable symbols added and possibly with some variables changed to different variables. Such crucial dynamic characteristics of an infinite derivation are captured by loop goals. The following result is proved in [51].

Theorem 3.1. *Let $G_0 = \leftarrow Q_0$ be a top goal with Q_0 a concrete query. Any infinite derivation D in GT_{G_0} contains an infinite sequence of goals $G_0, ..., G_{g_1}, ..., G_{g_2}, ...$ such that for any $j \geq 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}.* □

Put another way, Theorem 3.1 states that any infinite derivation D in GT_{G_0} is of the form

$$N_0 : G_0 \Rightarrow_{C_0} ... \; N_{g_1} : G_{g_1} \Rightarrow_{C_1} ... \; N_{g_2} : G_{g_2} \Rightarrow_{C_2} ... \; N_{g_3} : G_{g_3} \Rightarrow_{C_3} ...$$

where for any $j \geq 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}. This provides a necessary and sufficient characterization of an infinite generalized SLDNF-derivation with a concrete query.

Example 3.1. Consider the following logic program:

$$\begin{array}{lll} P_1: & p(a). & C_{p_1} \\ & p(f(X)) \leftarrow p(X). & C_{p_2} \end{array}$$

The generalized SLDNF-tree $GT_{\leftarrow p(X)}$ for a concrete query $p(X)$ is shown in Figure 3.2, where for simplicity the symbol $\leftarrow$ in each goal is omitted. Note that $GT_{\leftarrow p(X)}$ has an infinite derivation

$$N_0 : p(X) \Rightarrow_{C_{p2}} N_2 : p(X_2) \Rightarrow_{C_{p2}} N_4 : p(X_4) \Rightarrow_{C_{p2}} ...$$

where for any $j \geq 0$, $G_{2(j+1)}$ is a loop goal of G_{2j}. □

3.2 A Characterization of Infinite SLDNF-Derivations for Moded Queries

We first define generalized SLDNF-derivations for moded queries by introducing a framework called moded-query forests.

$$\begin{tikzcd}
 & N_0\colon p(X) \arrow[dl, "C_{p_1}"'] \arrow[d, "C_{p_2}"', "\theta_2 = \{X/f(X_2)\}"] \\
N_1\colon \Box_t & N_2\colon p(X_2) \arrow[dl, "C_{p_1}"'] \arrow[d, "C_{p_2}"', "\theta_4 = \{X_2/f(X_4)\}"] \\
N_3\colon \Box_t & N_4\colon p(X_4) \\
 & \vdots
\end{tikzcd}$$

Figure 3.2: The generalized SLDNF-tree $GT_{\leftarrow p(X)}$ of P_1 for a concrete query $p(X)$.

Definition 3.4. Let P be a logic program and $Q_0 = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$ a moded query. The *moded-query forest* of P for Q_0, denoted MF_{Q_0}, consists of all generalized SLDNF-trees for $P \cup \{G_0\}$, where $G_0 = \leftarrow p(t_1, ..., t_m, T_1, ..., T_n)$ with each t_i being a ground term from $HU(P)$. A (*generalized SLDNF-*) *derivation for the moded query* Q_0 is a derivation in any generalized SLDNF-tree of MF_{Q_0}. $\Box$

Therefore, a logic program P terminates for a moded query Q_0 if and only if there is no infinite derivation for Q_0 if and only if MF_{Q_0} has no infinite derivation.

Example 3.2. Consider the logic program P_1 again. We have $HU(P_1) = \{a, f(a), f(f(a)), ...\}$. Let $p(\mathcal{I})$ be a moded query. The moded-query forest $MF_{p(\mathcal{I})}$ consists of generalized SLDNF-trees $GT_{\leftarrow p(a)}$, $GT_{\leftarrow p(f(a))}$, etc., as shown in Figure 3.3. Note that $MF_{p(\mathcal{I})}$ has an infinite number of generalized SLDNF-trees. However, any individual tree, GT_{G_0} with $G_0 = \leftarrow p(\underbrace{f(f(...f}_{n\ items}(a)...)))$ $(n \geq 0)$, is finite. $MF_{p(\mathcal{I})}$ contains no infinite derivation, thus P_1 terminates for $p(\mathcal{I})$. $\Box$

$$\begin{tikzcd}
GT_{\leftarrow p(a)}: & N_0\colon p(a) \arrow[d, "C_{p_1}"'] & GT_{\leftarrow p(f(a))}: & N_0\colon p(f(a)) \arrow[d, "C_{p_2}"'] & \cdots \\
 & N_1\colon \Box_t & & N_2\colon p(a) \arrow[d, "C_{p_1}"'] & \\
 & & & N_3\colon \Box_t &
\end{tikzcd}$$

Figure 3.3: The moded-query forest $MF_{p(\mathcal{I})}$ of P_1 for a moded query $p(\mathcal{I})$.

In a moded-query forest, all input modes are instantiated into ground terms in $HU(P)$. When $HU(P)$ is infinite, the moded-query forest would contain infinitely many generalized SLDNF-trees. This means that it is infeasible to build a moded-query forest to represent the derivations for a moded query. An alternative yet ideal way is to directly apply SLDNF-resolution to evaluate input modes and build a compact generalized SLDNF-tree for a moded query. Unfortunately, SLDNF-resolution accepts only terms as arguments of a top goal; an input mode $\mathcal{I}$ is not directly evaluable.

Since an input mode stands for an arbitrary ground term, i.e. it can be any term from $HU(P)$, during query evaluation it can be instantiated to any term except variable (note that a ground term cannot be substituted by a variable). This suggests that we may approximate the effect of an input mode $\mathcal{I}$ by treating it as a special (meta-) variable I in the way that in SLDNF-derivations, I can be substituted by a constant or function, but cannot be substituted by an ordinary variable. Therefore, when doing unification of a special variable I and a variable X, we always substitute I for X.

Definition 3.5. Let P be a logic program and $Q_0 = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$ a moded query. The *moded generalized SLDNF-tree* of P for Q_0, denoted MT_{Q_0}, is defined to be the generalized SLDNF-tree GT_{G_0} for $P \cup \{G_0\}$, where $G_0 = \leftarrow p(I_1, ..., I_m, T_1, ..., T_n)$ with each I_i being a distinct special variable not occurring in any T_j. The special variables $I_1, ..., I_m$ for the input modes $\mathcal{I}_1, ..., \mathcal{I}_m$ are called *input mode variables* (or *input variables*). □

In a moded generalized SLDNF-tree, an input variable I may be substituted by either a constant t or a function $f(\overline{T})$. It will not be substituted by any non-input variable. If I is substituted by $f(\overline{T})$, all variables in $\overline{T}$ are also called input variables (thus are treated as special variables).

In the thesis, we do not consider *floundering moded* queries; i.e., we assume that no negative subgoals containing either ordinary or input variables are selected at any node of a moded generalized SLDNF-tree.

Definition 3.6. Let P be a logic program, $Q_0 = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$ a moded query, and $G_0 = \leftarrow p(I_1, ..., I_m, T_1, ..., T_n)$. Let D be a derivation in the moded generalized SLDNF-tree MT_{Q_0}. A *moded instance* of D is a derivation obtained from D by first instantiating all input variables at the root node $N_0 : G_0$ with an mgu $\theta = \{I_1/t_1, ..., I_m/t_m\}$, where each $t_i \in HU(P)$, then passing the instantiation θ down to the other nodes of D. □

Example 3.3. Consider the logic program P_1 again. Let $Q_0 = p(\mathcal{I})$ be a moded query and $G_0 = \leftarrow p(I)$. The moded generalized SLDNF-tree MT_{Q_0} is GT_{G_0} as depicted in Figure 3.4, where all input variables are underlined. Since

I is an input variable, X_2 is an input variable, too (due to the mgu θ_2). For the same reason, all X_{2i} are input variables ($i > 0$).

Consider the following infinite derivation D in MT_{Q_0}:

$$N_0 : p(I) \Rightarrow_{C_{p2}} N_2 : p(X_2) \Rightarrow_{C_{p2}} N_4 : p(X_4) \Rightarrow_{C_{p2}} \cdots$$

By instantiating the input variable I at N_0 with different ground terms from $HU(P_1)$ and passing the instantiation θ down to the other nodes of D, we can obtain different moded instances from D. For example, instantiating I to a (i.e. $\theta = \{I/a\}$) yields the moded instance

$$N_0 : p(a)$$

Instantiating I to $f(a)$ (i.e. $\theta = \{I/f(a)\}$) yields the moded instance

$$N_0 : p(f(a)) \Rightarrow_{C_{p2}} N_2 : p(a)$$

And, instantiating I to $f(f(a))$ (i.e. $\theta = \{I/f(f(a))\}$) yields the moded instance

$$N_0 : p(f(f(a))) \Rightarrow_{C_{p2}} N_2 : p(f(a)) \Rightarrow_{C_{p2}} N_4 : p(a)$$

□

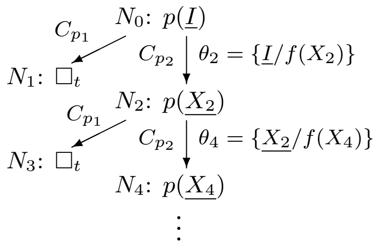

N_0: $p(\underline{I})$
C_{p1}
C_{p2}
$\theta_2 = \{\underline{I}/f(X_2)\}$
N_1: $\Box_t$
N_2: $p(\underline{X_2})$
C_{p1}
C_{p2}
$\theta_4 = \{\underline{X_2}/f(X_4)\}$
N_3: $\Box_t$
N_4: $p(\underline{X_4})$

Figure 3.4: The moded generalized SLDNF-tree $MT_{p(\mathcal{I})}$ of P_1 for a moded query $p(\mathcal{I})$.

Observe that a moded instance of a derivation D in MT_{Q_0} is a derivation in $GT_{G_0\theta}$, where $G_0\theta = \leftarrow p(t_1, ..., t_m, T_1, ..., T_n)$ with each t_i being a ground term from $HU(P)$. By Definition 3.4, $GT_{G_0\theta}$ is in the moded-query forest MF_{Q_0}. This means that any moded instance of a derivation in MT_{Q_0} is a derivation for Q_0 in MF_{Q_0}. For instance, all moded instances illustrated in Example 3.3 are derivations in the moded-query forest MF_{Q_0} of Figure 3.3.

Theorem 3.2. *Let MF_{Q_0} and MT_{Q_0} be the moded-query forest and the moded generalized SLDNF-tree of P for Q_0, respectively. If MF_{Q_0} has an infinite derivation D', MT_{Q_0} has an infinite derivation D with D' as a moded instance.* $\Box$

Proof. Let $Q_0 = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$. Then, the root node of D' is $N_0 :\leftarrow p(t_1, ..., t_m, T_1, ..., T_n)$ with each $t_i \in HU(P)$, and the root node of MT_{Q_0} is $N_0 :\leftarrow p(I_1, ..., I_m, T_1, ..., T_n)$ with each I_i being an input variable not occurring in any T_j. Note that the former is an instance of the latter with the mgu $\theta = \{I_1/t_1, ..., I_m/t_m\}$. Let D' be of the form

$$N_0 :\leftarrow p(t_1, ..., t_m, T_1, ..., T_n) \Rightarrow_{C_0} N_1 : G'_1 \cdots \Rightarrow_{C_i} N_{i+1} : G'_{i+1} \cdots$$

MT_{Q_0} must have a derivation D of the form

$$N_0 :\leftarrow p(I_1, ..., I_m, T_1, ..., T_n) \Rightarrow_{C_0} N_1 : G_1 \cdots \Rightarrow_{C_i} N_{i+1} : G_{i+1} \cdots$$

such that each $G'_i = G_i\theta$, since for any $i \geq 0$ and any clause C_i in P, if G'_i can unify with C_i, so can G_i with C_i. Note that when the selected subgoal at some G'_i is a negative ground literal, by the assumption that Q_0 is non-floundering, we have the same selected literal at G_i. We then have the proof. $\Box$

Our goal is to establish a characterization of infinite derivations for a moded query such that the converse of Theorem 3.2 is true under some conditions.

Consider the infinite derivation in Figure 3.4 again. The input variable I is substituted by $f(X_2)$; X_2 is then substituted by $f(X_4)$, ... This produces an infinite chain of substitutions for I of the form $I/f(X_2), X_2/f(X_4)$, ... The following lemma shows that infinite derivations containing such an infinite chain of substitutions have no infinite moded instances.

Lemma 3.1. *If a derivation D in a moded generalized SLDNF-tree MT_{Q_0} is infinite but none of its moded instances is infinite, then there is an input variable I such that D contains an infinite chain of substitutions for I of the form*

$$I/f_1(..., Y_1, ...), ..., Y_1/f_2(..., Y_2, ...), ..., Y_{i-1}/f_i(..., Y_i, ...), ... \tag{3.1}$$

(some f_is would be the same). $\Box$

Proof. We distinguish four types of substitution chains for an input variable I in D:

1. $X_1/I, ..., X_m/I$ or $X_1/I, ..., X_i/I, \ldots$ That is, I is never substituted by any terms.

2. $X_1/I, ..., X_m/I, I/t$ where t is a ground term. That is, I is substituted by a ground term.

3. $X_1/I, ..., X_m/I, I/f_1(..., Y_1, ...), ..., Y_1/f_2(..., Y_2, ...), ..., Y_{n-1}/f_n(..., Y_n, ...)$, $\ldots$, where $f_n(\ ..., Y_n, ...)$ is the last non-ground function in the substitution chain for I in D. In this case, I is recursively substituted by a finite number of functions.

4. $X_1/I, ..., X_m/I, I/f_1(..., Y_1, ...), ..., Y_1/f_2(..., Y_2, ...), ..., Y_{i-1}/f_i(..., Y_i, ...), \ldots$ In this case, I is recursively substituted by an infinite number of functions.

For type 1, D retains its infinite extension for whatever ground term we replace I with. For type 2, D retains its infinite extension when we use t to replace I. To sum up, for any input variable I whose substitution chain is of type 1 or of type 2, there is a ground term t such that replacing I with t does not affect the infinite extension of D. In this case, replacing I in D with t leads to an infinite derivation less general than D.

For type 3, note that all variables appearing in the $f_i(.)$s are input variables. Since $f_n(..., Y_n, ...)$ is the last non-ground function in the substitution chain for I in D, the substitution chain for every variable Y_n in $f_n(..., Y_n, ...)$ is either of type 1 or of type 2. Therefore, we can replace each Y_n with an appropriate ground term t_n without affecting the infinite extension of D. After this replacement, D becomes D_n and $f_n(..., Y_n, ...)$ becomes a ground term $f_n(..., t_n, ...)$. Now $f_{n-1}(..., Y_{n-1}, ...)$ is the last non-ground function in the substitution chain for I in D_n. Repeating the above replacement recursively, we will obtain an infinite derivation D_1, which is D with all variables in the $f_i(.)$s replaced with a ground term. Assume $f_1(..., Y_1, ...)$ becomes a ground term t in D_1. Then the substitution chain for I in D_1 is of type 2. So replacing I with t in D_1 leads to an infinite derivation D_0.

The above constructive proof shows that if the substitution chains for all input variables in D are of type 1, 2 or 3, then D must have an infinite moded instance. Since D has no infinite moded instance, there must exist an input variable I whose substitution chain in D is of type 4. That is, I is recursively substituted by an infinite number of functions. Note that some f_is would be the same because a logic program has only a finite number of function symbols. This concludes the proof. $\square$

We are ready to introduce the following principal result.

Theorem 3.3. *Let MF_{Q_0} and MT_{Q_0} be the moded-query forest and the moded generalized SLDNF-tree of P for Q_0, respectively. MF_{Q_0} has an infinite derivation if and only if MT_{Q_0} has an infinite derivation D of the form*

$$N_0 : G_0 \Rightarrow_{C_0} ... \ N_{g_1} : G_{g_1} \Rightarrow_{C_1} ... \ N_{g_2} : G_{g_2} \Rightarrow_{C_2} ... \ N_{g_3} : G_{g_3} \Rightarrow_{C_3} ... (3.2)$$

where (i) for any $j \geq 1$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (ii) for no input variable I, D contains an infinite chain of substitutions for I of form (3.1). □

Proof. ($\Longrightarrow$) Assume MF_{Q_0} has an infinite derivation D'. By Theorem 3.2, GT_{G_0} has an infinite derivation D with D' as a moded instance. By Theorem 3.1, D is of form (3.2) and satisfies condition (i).

Assume, on the contrary, that D does not satisfy condition (ii). That is, for some input variable I, D contains an infinite chain of substitutions for I of the form

$$I/f_1(..., Y_1, ...), ..., Y_1/f_2(..., Y_2, ...), ..., Y_{i-1}/f_i(..., Y_i, ...), ...$$

Note that for whatever ground term t we assign to I, this chain can be instantiated at most as long in length as the following one:

$$t/f_1(..., t_1, ...), ..., t_1/f_2(..., t_2, ...), ..., t_k/f_{k+1}(..., Y_{k+1}, ...)$$

where $k = |t|$, t_is are ground terms and $|t_k| = 1$. This means that replacing I with any ground term t leads to a finite moded instance of D. Therefore, D has no infinite moded instance in MF_{Q_0}, a contradiction.

($\Longleftarrow$) Assume, on the contrary, that MF_{Q_0} has no infinite derivation. By Lemma 3.1, we reach a contradiction to condition (ii). □

Theorem 3.3 provides a necessary and sufficient characterization of an infinite generalized SLDNF-derivation for a moded query. Note that it coincides with Theorem 3.1 when Q_0 is a concrete query, where $MF_{Q_0} = MT_{Q_0}$ and condition (ii) is always true.

The following corollary is immediate to this theorem.

Corollary 3.1. *A logic program P terminates for a moded query Q_0 if and only if the moded generalized SLDNF-tree MT_{Q_0} has no infinite derivation of form (3.2) satisfying conditions (i) and (ii) of Theorem 3.3.* □

We use simple yet typical examples to illustrate the proposed characterization of infinite SLDNF-derivations with moded queries.

Example 3.4. Consider the moded generalized SLDNF-tree MT_{Q_0} in Figure 3.4. It has only one infinite derivation, which satisfies condition (i) of Theorem 3.3 where for each $j \geq 0$, N_{g_j} in Theorem 3.3 corresponds to N_{2j} in Figure 3.4. However, the chain of substitutions for I in this derivation violates condition (ii). This means that MF_{Q_0} contains no infinite derivations; therefore, there is no infinite derivation for the moded query $p(\mathcal{I})$. As a result, P_1 terminates for $p(\mathcal{I})$. □

Example 3.5. Consider the following logic program:

$$\begin{array}{lll} P_2: & \mathit{append}([], X, X). & C_{a_1} \\ & \mathit{append}([X|Y], U, [X|Z]) \leftarrow \mathit{append}(Y, U, Z). & C_{a_2} \end{array}$$

Let us choose the three simplest moded queries:

$$\begin{array}{l} Q_0^1 = \mathit{append}(\mathcal{I}, V_2, V_3), \\ Q_0^2 = \mathit{append}(V_1, \mathcal{I}, V_3), \\ Q_0^3 = \mathit{append}(V_1, V_2, \mathcal{I}). \end{array}$$

Since applying clause C_{a_1} produces only leaf nodes, for simplicity we ignore it when depicting moded generalized SLDNF-trees. The three moded generalized SLDNF-trees $MT_{Q_0^1}$, $MT_{Q_0^2}$ and $MT_{Q_0^3}$ are shown in Figures 3.5 (a), (b) and (c), respectively. Note that all the derivations are infinite and satisfy condition (i) of Theorem 3.3, where for each $j \geq 0$, N_{g_j} in Theorem 3.3 corresponds to N_j in Figure 3.5. Apparently, the chains of substitutions for I in the derivations of $MT_{Q_0^1}$ and $MT_{Q_0^3}$ violate condition (ii) of Theorem 3.3. $MF_{Q_0^1}$ and $MF_{Q_0^3}$ contain no infinite derivation and thus there exists no infinite derivation for the moded queries Q_0^1 and Q_0^3. Therefore, P_2 terminates for Q_0^1 and Q_0^3. However, the derivation in $MT_{Q_0^2}$ satisfies condition (ii), thus there exist infinite derivations for the moded query Q_0^2. P_2 does not terminate for Q_0^2. □

$$\begin{array}{lll}
N_0\text{: } \mathit{append}(\underline{I}, V_2, V_3) & N_0\text{: } \mathit{append}(V_1, \underline{I}, V_3) & N_0\text{: } \mathit{append}(V_1, V_2, \underline{I}) \\
\quad C_{a_2}\big\downarrow \begin{array}{l}\theta_0 = \{\underline{I}/[X|Y], \\ U/V_2, V_3/[X|Z]\}\end{array} & \quad C_{a_2}\big\downarrow \begin{array}{l}\theta_0 = \{V_1/[X|Y], \\ U/\underline{I}, V_3/[X|Z]\}\end{array} & \quad C_{a_2}\big\downarrow \begin{array}{l}\theta_0 = \{V_1/[X|Y], \\ U/V_2, \underline{I}/[X|Z]\}\end{array} \\
N_1\text{: } \mathit{append}(\underline{Y}, V_2, Z) & N_1\text{: } \mathit{append}(Y, \underline{I}, Z) & N_1\text{: } \mathit{append}(Y, V_2, \underline{Z}) \\
\quad C_{a_2}\big\downarrow \begin{array}{l}\theta_1 = \{\underline{Y}/[X_1|Y_1], \\ U_1/V_2, Z/[X_1|Z_1]\}\end{array} & \quad C_{a_2}\big\downarrow \begin{array}{l}\theta_1 = \{Y/[X_1|Y_1], \\ U_1/\underline{I}, Z/[X_1|Z_1]\}\end{array} & \quad C_{a_2}\big\downarrow \begin{array}{l}\theta_1 = \{Y/[X_1|Y_1], \\ U_1/V_2, \underline{Z}/[X_1|Z_1]\}\end{array} \\
N_2\text{: } \mathit{append}(\underline{Y_1}, V_2, Z_1) & N_2\text{: } \mathit{append}(Y_1, \underline{I}, Z_1) & N_2\text{: } \mathit{append}(Y_1, V_2, \underline{Z_1}) \\
\\
\qquad\text{(a)} & \qquad\text{(b)} & \qquad\text{(c)}
\end{array}$$

Figure 3.5: (a) $MT_{Q_0^1}$, (b) $MT_{Q_0^2}$, and (c) $MT_{Q_0^3}$.

Let $\mathit{pred}(P)$ be the set of predicate symbols in P. Define

$$MQ(P) = \{p(\mathcal{I}_1, ..., \mathcal{I}_m, X_{m+1}, ..., X_n) \mid p \text{ is an } n\text{-ary predicate symbol in } pred(P) \text{ and } m > 0\}.$$

Note that $MQ(P)$ contains all most general moded queries of P in the sense that any moded query of P is an instance of some query in $MQ(P)$. Since $pred(P)$ is finite, $MQ(P)$ is finite. Therefore, it is immediate that P terminates for all moded queries if and only if it terminates for each moded query in $MQ(P)$. Note that there is no restriction on the argument positions on which we allow input modes. In example, $MQ(P)$ also contains queries such as $p(X_1, \mathcal{I}_1, ..., \mathcal{I}_m, X_{m+2}, ..., X_n)$. To simplify the notation however, moded atoms are represented as having the input modes on the first arguments if possible.

Theorem 3.4. *For any two moded queries $Q_0^1 = p(\mathcal{I}_1, ..., \mathcal{I}_l, X_{l+1}, ..., X_n)$ and $Q_0^2 = p(\mathcal{I}_1, ..., \mathcal{I}_m, X_{m+1}, ..., X_n)$ with $l < m$, that there is no infinite derivation for Q_0^1 implies there is no infinite derivation for Q_0^2.* □

Proof. Note that we consider only non-floundering queries. Then, for any concrete query Q, that there is no infinite derivation for Q implies there is no infinite derivation for any instance of Q. Assume that there is no infinite derivation for Q_0^1. Then, there is no infinite derivation for any query $Q = p(t_1, ..., t_l, X_{l+1}, ..., X_n)$, where each t_i is a ground term from $HU(P)$. Then, there is no infinite derivation for any query $Q' = p(t_1, ..., t_l, s_{l+1}, ..., s_m, X_{m+1}, ..., X_n)$, where each t_i is a ground term from $HU(P)$ and each s_i an instance of X_i. Since all X_is are variables, there is no infinite derivation for any query $Q'' = p(t_1, ..., t_l, t_{l+1}, ..., t_m, X_{m+1}, ..., X_n)$, where each t_i is a ground term from $HU(P)$. That is, there is no infinite derivation for Q_0^2. □

Applying this theorem, we can conclude that P_2 in Example 3.5 terminates for all moded queries in $MQ(P_2)$ except Q_0^2.

3.3 An Algorithm for Predicting Termination of Logic Programs

We develop an algorithm for predicting termination of logic programs based on the necessary and sufficient characterization of an infinite generalized SLDNF-derivation (Theorem 3.3 and Corollary 3.1). We begin by introducing a complete loop checking mechanism.

Definition 3.7. Given a repetition number $r \geq 2$, *LP-check* is defined as follows: Any derivation D in a generalized SLDNF-tree is cut at a node N_{g_r} if

D has a prefix of the form

$$N_0 : G_0 \Rightarrow_{C_0} ... \; N_{g_1} : G_{g_1} \Rightarrow_{C_k} ... \; N_{g_2} : G_{g_2} \Rightarrow_{C_k} ... \; N_{g_r} : G_{g_r} \Rightarrow_{C_k} \quad (3.3)$$

such that (a) for any $j < r$, $G_{g_{j+1}}$ is a loop goal of G_{g_j}, and (b) for all $j \leq r$, the clause C_k applied to G_{g_j} is the same. C_k is then called a *looping clause.* □

LP-check predicts infinite derivations from prefixes of derivations based on the characterization of Theorem 3.1 (or condition (i) of Theorem 3.3). The repetition number r specifies the minimum number of loop goals appearing in the prefixes. It appears not appropriate to choose $r < 2$, as that may lead to many finite derivations being wrongly cut. Although there is no mathematical mechanism available for choosing this repetition number (since the termination problem is undecidable), our experimental results show that in many situations, it suffices to choose $r = 3$ for a correct prediction of infinite derivations.

LP-check applies to any generalized SLDNF-trees including moded generalized SLDNF-trees.

Theorem 3.5. *LP-check is a complete loop check.* □

Proof: Let D be an infinite derivation in GT_{G_0}. By Theorem 3.1, D is of the form

$$N_0 : G_0 \Rightarrow_{C_0} ... \; N_{f_1} : G_{f_1} \Rightarrow_{C_1} ... \; N_{f_2} : G_{f_2} \Rightarrow_{C_2} ...$$

such that for any $i \geq 1$, $G_{f_{i+1}}$ is a loop goal of G_{f_i}. Since a logic program has only a finite number of clauses, there must be a (looping) clause C_k being repeatedly applied at infinitely many nodes $N_{g_1} : G_{g_1}, N_{g_2} : G_{g_2}, \cdots$ where for each $j \geq 1$, $g_j \in \{f_1, f_2, ...\}$. Then for any $r > 0$, D has a partial derivation of form (3.3). So D will be cut at node $N_{g_r} : G_{g_r}$. This shows that any infinite derivation can be cut by LP-check. That is, LP-check is a complete loop check. □

Example 3.6. Let us choose $r = 3$ and consider the infinite derivation D in Figure 3.4. $p(X_4)$ at N_4 is a loop goal of $p(X_2)$ at N_2 that is a loop goal of $p(I)$ at N_0. Moreover, the same clause C_{p_2} is applied at the three nodes. D satisfies the conditions of LP-check and is cut at node N_4. □

Recall that to prove that a logic program P terminates for a moded query $Q_0 = p(\mathcal{I}_1, ..., \mathcal{I}_m, T_1, ..., T_n)$ is to prove that P terminates for any query $p(t_1, ..., t_m, T_2, ..., T_n)$, where each t_i is a ground term. This can be reformulated in terms of a moded-query forest; that is, P terminates for Q_0 if and only if MF_{Q_0} has no infinite derivation. Then, Corollary 3.1 shows that P terminates for Q_0 if and only if the moded generalized SLDNF-tree MT_{Q_0} has no infinite derivation D of form (3.2) satisfying the two conditions (i) and (ii) of Theorem 3.3. Although this characterization cannot be directly used for automated termination test because it requires generating infinite derivations in MT_{Q_0}, it can be used along with LP-check to predict termination, as LP-check is able to guess if a partial derivation would extend to an infinite one. Before describing our prediction algorithm with this idea, we introduce one more condition following Definition 3.7.

Definition 3.8. Let D be a prefix of form (3.3). D is said to have the *term-size decrease* property if for any i with $0 < i < r$, there is a substitution $X/f(...Y...)$ between N_{g_i} and $N_{g_{i+1}}$, where X is an input variable and Y (an ordinary or input variable) appears in the selected subgoal of $G_{g_{i+1}}$. $\square$

Theorem 3.6. *Let D be a derivation such that for all $r \geq 2$ D has a prefix of form (3.3) which has the term-size decrease property. D contains an infinite chain of substitutions of form (3.1) for some input variable I at the root node of D.* $\square$

Proof. Due to the term-size decrease property of the prefix of D which holds for all $r \geq 2$, D contains an infinite number of substitutions of the form $X/f(...)$, where X is an input variable. Assume, on the contrary, that D does not contain such an infinite chain of form (3.1). Let M be the longest length of substitutions of form (3.1) for each input variable I at the root node of D. Note that each input variable can be substituted only by a constant or function. For each substitution $X/f(...)$ with X an input variable, assume $f(...)$ contains at most N variables (i.e., it introduces at most N new input variables). Then, D contains at most $K * (N^0 + N^1 + ... + N^M)$ substitutions of the form $X/f(...)$, where K is the number of input variables at the root node of D and X is an input variable. This contradicts the condition that D contains an infinite number of such substitutions. We conclude the proof. $\square$

LP-check and the term-size decrease property approximate conditions (i) and (ii) of Theorem 3.3, respectively. So, we can guess an infinite extension (3.2) from a prefix (3.3) by combining the two mechanisms, as described in the following algorithm.

Algorithm 3.1. *Input: A logic program P, a (concrete or moded) query Q_0, and a repetition number $r \geq 2$ ($r = 3$ is recommended).*
Output: terminating, predicted-terminating, or predicted-non-terminating.
Method: Apply the following procedure.

```
procedure TPoLP(P, $Q_0$, r)
{
```

1. *Initially, set $L = 0$. Construct the moded generalized SLDNF-tree MT_{Q_0} of P for Q_0 in the way that whenever a prefix D of the form*

$$N_0 : G_0 \Rightarrow_{C_0} ... N_{g_1} : G_{g_1} \Rightarrow_{C_k} ... N_{g_2} : G_{g_2} \Rightarrow_{C_k} ... N_{g_r} : G_{g_r} \Rightarrow_{C_k}$$

   is produced which satisfies conditions (a) and (b) of LP-check, if D does not have the term-size decrease property then goto 3; else set $L = 1$ and extend D from the node N_{g_r} with the looping clause C_k skipped and continue with the construction of the tree of step 1.

2. *Return* terminating *if $L = 0$; otherwise, return* predicted-terminating.

3. *Return* predicted-non-terminating.

} □

Starting from the root node $N_0 : G_0$, we generate derivations of a moded generalized SLDNF-tree MT_{Q_0} step by step. If a prefix D of form (3.3) is generated which satisfies conditions (a) and (b) of LP-check, then by Theorem 3.1 D is very likely to extend infinitely in MT_{Q_0} (via the looping clause C_k). By Theorem 3.2, however, D may not have infinite moded instances in MF_{Q_0}. So in this case, we further check if D has the term-size decrease property. If not, by Theorem 3.3 D is very likely to have moded instances that extend infinitely in MF_{Q_0}. Algorithm 3.1 then predicts non-terminating for Q_0 by returning an answer *predicted-non-terminating.* If D has the term-size decrease property, however, we continue to extend D from N_{g_r} by skipping the clause C_k (i.e., the derivation via C_k is cut at N_{g_r} by LP-check).

When the answer is not *predicted-non-terminating*, we distinguish between two cases: (1) $L = 0$. This shows that no derivation was cut by LP-check during the construction of MT_{Q_0}. Algorithm 3.1 concludes terminating for Q_0 by returning an answer *terminating.* (2) $L = 1$. This means that some derivations were cut by LP-check, all of which have the term-size decrease property. Algorithm 3.1 then predicts terminating for Q_0 by returning an answer *predicted-terminating.*

Note that for a concrete query Q_0, no derivation has the term-size decrease property. Therefore, Algorithm 3.1 returns *predicted-non-terminating* for Q_0 once a prefix of a derivation satisfying the conditions of LP-check is generated.

Theorem 3.7. *For any logic program P, concrete/moded query Q_0 and repetition number r, Algorithm 3.1 always terminates.* □

Proof. Algorithm 3.1 constructs MT_{Q_0} while applying LP-check to cut possible infinite derivations. Since LP-check is a complete loop check, it cuts all infinite derivations at some depth. This means that MT_{Q_0} after cut by LP-check is finite. So, Algorithm 3.1 always terminates. □

Algorithm 3.1 yields an approximate answer, *predicted-terminating* or *predicted-non-terminating*, or an exact answer *terminating*, as shown by the following theorem.

Theorem 3.8. *P terminates for Q_0 if Algorithm 3.1 returns terminating.* □

Proof. If Algorithm 3.1 returns *terminating*, no derivations were cut by LP-check; so the moded generalized SLDNF-tree MT_{Q_0} for Q_0 is finite. By Corollary 3.1, P terminates for Q_0. □

In the following examples, we choose a repetition number $r = 3$.

Example 3.7. Consider Figure 3.4. Since the prefix D between N_0 and N_4 satisfies the conditions of LP-check, Algorithm 3.1 concludes that the derivation may extend infinitely in MT_{Q_0}. It then checks the term-size decrease property to see if D has moded instances that would extend infinitely in MF_{Q_0}. Clearly, D has the term-size decrease property. So Algorithm 3.1 skips C_{p_2} at N_4 (the branch is cut by LP-check). Consequently, Algorithm 3.1 predicts terminating for $p(\mathcal{I})$ by returning an answer *predicted-terminating.* This prediction is correct; see Example 3.4. □

Example 3.8. Consider Figure 3.5. All the derivations starting at N_0 and ending at N_2 satisfy the conditions of LP-check, so they are cut at N_2. Since the derivations in $MT_{Q_0^1}$ and $MT_{Q_0^3}$ have the term-size decrease property, Algorithm 3.1 returns *predicted-terminating* for Q_0^1 and Q_0^3. Since the derivation in $MT_{Q_0^2}$ does not have the term-size decrease property, Algorithm 3.1 returns *predicted-non-terminating* for Q_0^2. These predictions are all correct; see Example 3.5. □

Example 3.9. Consider the following logic program P_3:

$$\begin{array}{ll} mult(s(X), Y, Z) \leftarrow mult(X, Y, U), add(U, Y, Z). & C_{m_1} \\ mult(0, Y, 0). & C_{m_2} \\ add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z). & C_{a_1} \\ add(0, Y, Y). & C_{a_2} \end{array}$$

$MQ(P_3)$ consists of fourteen moded queries, seven for predicate *mult* and seven for predicate *add*. Applying Algorithm 3.1 yields the following result: (1) P_3 is *predicted-terminating* for all moded queries to *add* except $add(V_1, \mathcal{I}_2, V_3)$

for which P_3 is *predicted-non-terminating*, and (2) P_3 is *predicted-terminating* for $mult(\mathcal{I}_1, \mathcal{I}_2, V_3)$ and $mult(\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3)$, but is *predicted-non-terminating* for the remaining moded queries to *mult*. For illustration, we depict two moded generalized SLDNF-trees for $mult(\mathcal{I}, V_2, V_3)$ and $mult(\mathcal{I}_1, \mathcal{I}_2, V_3)$, as shown in Figures 3.6 (a) and (b), respectively. In the two moded generalized SLDNF-trees, the prefix from N_0 down to N_2 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{m_1} is skipped when expanding N_2. When the derivation is extended to N_6, the conditions of LP-check are satisfied again, where G_6 is a loop goal of G_5 that is a loop goal of G_4. Since the derivation for $mult(\mathcal{I}, V_2, V_3)$ (Figure 3.6 (a)) does not have the term-size decrease property, Algorithm 3.1 returns an answer, *predicted-non-terminating*, for this moded query. The derivation for $mult(\mathcal{I}_1, \mathcal{I}_2, V_3)$ (Figure 3.6 (b)) has the term-size decrease property, so clause C_{a_1} is skipped when expanding N_6. For simplicity, we omitted all derivations leading to a success leaf. Because all derivations satisfying the conditions of LP-check have the term-size decrease property, Algorithm 3.1 ends with an answer, *predicted-terminating*, for $mult(\mathcal{I}_1, \mathcal{I}_2, V_3)$. It is then immediately inferred by Theorem 3.4 that P_3 is *predicted-terminating* for $mult(\mathcal{I}_1, \mathcal{I}_2, \mathcal{I}_3)$. It is not difficult to verify that all these predictions are correct. □

N_0: $mult(\underline{I}, V_2, V_3)$
$\big\downarrow C_{m_1} \; \theta_0 = \{\underline{I}/s(X_1), Y_1/V_2, Z_1/V_3\}$
N_1: $mult(\underline{X_1}, V_2, U_1), add(U_1, V_2, V_3)$
$\big\downarrow C_{m_1} \; \theta_1 = \{\underline{X_1}/s(X_2), Y_2/V_2, Z_2/U_1\}$
N_2: $mult(\underline{X_2}, V_2, U_2), add(U_2, V_2, U_1), add(U_1, V_2, V_3)$
$\big\downarrow C_{m_2} \; \theta_2 = \{\underline{X_2}/0, Y_3/V_2, U_2/0\}$
N_3: $add(0, V_2, U_1), add(U_1, V_2, V_3)$
$\big\downarrow C_{a_2} \; \theta_3 = \{Y_4/V_2, U_1/V_2\}$
N_4: $add(V_2, V_2, V_3)$
$\big\downarrow C_{a_1} \; \theta_4 = \{V_2/s(X_3), V_3/s(Z_3)\}$
N_5: $add(X_3, s(X_3), Z_3)$
$\big\downarrow C_{a_1} \; \theta_5 = \{X_3/s(X_4), Z_3/s(Z_4)\}$
N_6: $add(X_4, s(s(X_4)), Z_4)$
(a)

N_0: $mult(\underline{I_1}, \underline{I_2}, V_3)$
$\big\downarrow C_{m_1} \; \theta_0 = \{\underline{I_1}/s(X_1), Y_1/\underline{I_2}, Z_1/V_3\}$
N_1: $mult(\underline{X_1}, \underline{I_2}, U_1), add(U_1, \underline{I_2}, V_3)$
$\big\downarrow C_{m_1} \; \theta_1 = \{\underline{X_1}/s(X_2), Y_2/\underline{I_2}, Z_2/\underline{U_1}\}$
N_2: $mult(\underline{X_2}, \underline{I_2}, U_2), add(U_2, \underline{I_2}, U_1), add(U_1, \underline{I_2}, V_3)$
$\big\downarrow C_{m_2} \; \theta_2 = \{\underline{X_2}/0, Y_3/\underline{I_2}, U_2/0\}$
N_3: $add(0, \underline{I_2}, U_1), add(U_1, \underline{I_2}, V_3)$
$\big\downarrow C_{a_2} \; \theta_3 = \{Y_4/\underline{I_2}, U_1/\underline{I_2}\}$
N_4: $add(\underline{I_2}, \underline{I_2}, V_3)$
$\big\downarrow C_{a_1} \; \theta_4 = \{\underline{I_2}/s(X_3), V_3/s(Z_3)\}$
N_5: $add(\underline{X_3}, s(\underline{X_3}), Z_3)$
$\big\downarrow C_{a_1} \; \theta_5 = \{\underline{X_3}/s(X_4), Z_3/s(Z_4)\}$
N_6: $add(\underline{X_4}, s(s(\underline{X_4})), Z_4)$
(b)

Figure 3.6: Two moded generalized SLDNF-trees of P_3 generated by Algorithm 3.1.

AProVE07 [23], NTI [38, 37], Polytool [35] and TALP [36] are four well-known state-of-the-art analyzers. NTI proves non-termination, while the others prove termination. The Termination Competition 2007 [1] reports their latest performance. We borrow three representative logic programs from the competition website to further demonstrate the effectiveness of our termination prediction.

Example 3.10. Consider the following logic program coming from the Termination Competition 2007 with Problem id *LP/talp/apt - subset1* and difficulty rating 100%.
AProVE07, NTI, Polytool and TALP all failed to prove/disprove its termination by yielding an answer "don't know" in the competition [1].

$$\begin{array}{lll}
P_4: & member1(X,[Y|Xs]) \leftarrow member1(X,Xs). & C_{m_1}\\
 & member1(X,[X|Xs]). & C_{m_2}\\
 & subset1([X|Xs],Ys):-member1(X,Ys), subset1(Xs,Ys). & C_{s_1}\\
 & subset1([],Ys). & C_{s_2}
\end{array}$$

Query Mode: $subset1(o,i)$.

The query mode $subset1(o,i)$ means that the second argument of any query must be a ground term, while the first one can be an arbitrary term. Then, to prove the termination property of P_4 with this query mode is to prove the termination for the moded query $Q_0 = subset1(V,\mathcal{I})$. Applying Algorithm 3.1 generates a moded generalized SLDNF-tree as shown in Figure 3.7. The prefix from N_0 down to N_3 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{m_1} is skipped when expanding N_3. When the derivation is extended to N_{10}, the conditions of LP-check are satisfied again, where G_{10} is a loop goal of G_9 that is a loop goal of G_8. Since the derivation has the term-size decrease property, N_{10} is expanded by C_{m_2}.

At N_{11} (resp. N_{13} and N_{15}), the derivation satisfies the conditions of LP-check and has the term-size decrease property, where G_{11} (resp. N_{13} and N_{15}) is a loop goal of G_4 that is a loop goal of G_0. Therefore, N_{11} (resp. N_{13} and N_{15}) is expanded by C_{s_2}. When the derivation is extended to N_{17}, the conditions of LP-check are satisfied, where G_{17} is a loop goal of G_4 that is a loop goal of G_0, but the term-size decrease condition is violated. Algorithm 3.1 stops immediately with an answer, *predicted-non-terminating*, for the query Q_0. It is easy to verify that this prediction is correct. $\square$

Example 3.11. Consider another logic program in the Termination Competition 2007 with Problem id *LP/SGST06 - incomplete* and difficulty rating 75%. Polytool succeeded to prove its termination, while AProVE07, NTI and TALP failed [1].

$$\begin{array}{lll}
P_5: & p(X) \leftarrow q(f(Y)), p(Y). & C_{p1}\\
 & p(g(X)) \leftarrow p(X). & C_{p2}\\
 & q(g(Y)). & C_{q_1}
\end{array}$$

Query Mode: $p(i)$.

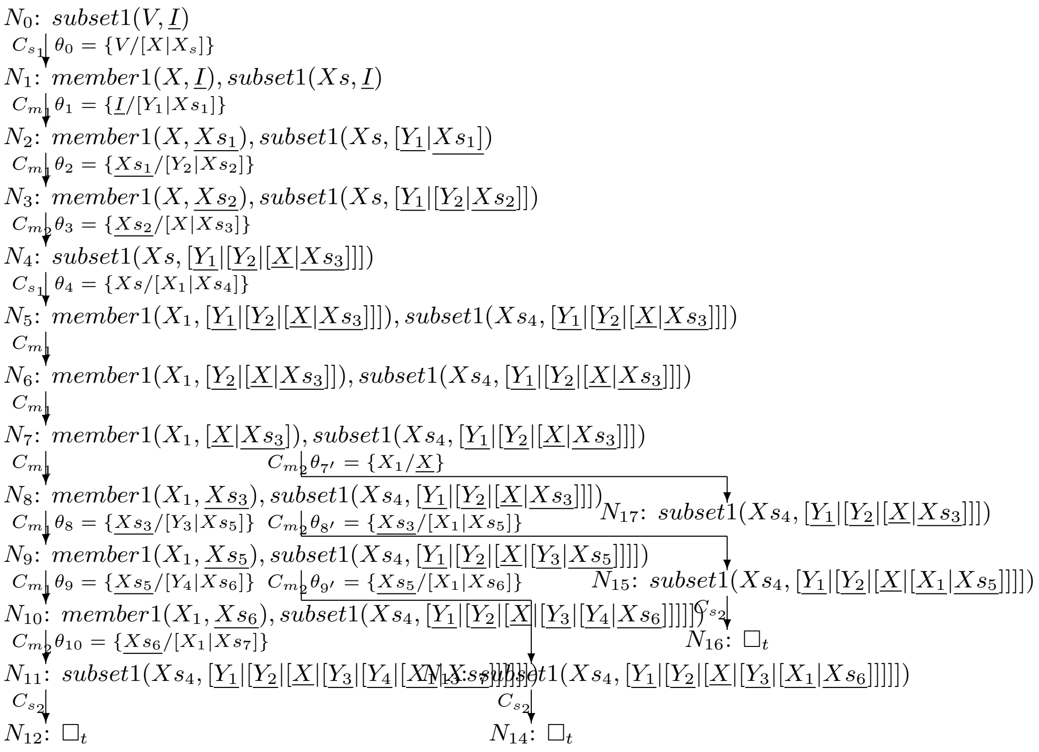

N_0: $subset1(V, \underline{I})$
C_{s_1} $\theta_0 = \{V/[X|X_s]\}$
N_1: $member1(X, \underline{I}), subset1(Xs, \underline{I})$
C_{m_1} $\theta_1 = \{\underline{I}/[Y_1|Xs_1]\}$
N_2: $member1(X, \underline{Xs_1}), subset1(Xs, [\underline{Y_1}|\underline{Xs_1}])$
C_{m_1} $\theta_2 = \{\underline{Xs_1}/[Y_2|Xs_2]\}$
N_3: $member1(X, \underline{Xs_2}), subset1(Xs, [\underline{Y_1}|[\underline{Y_2}|\underline{Xs_2}]])$
C_{m_2} $\theta_3 = \{\underline{Xs_2}/[X|Xs_3]\}$
N_4: $subset1(Xs, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
C_{s_1} $\theta_4 = \{Xs/[X_1|Xs_4]\}$
N_5: $member1(X_1, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]]), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
C_{m_1}
N_6: $member1(X_1, [\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
C_{m_1}
N_7: $member1(X_1, [\underline{X}|\underline{Xs_3}]), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
C_{m_1}
C_{m_2} $\theta_{7'} = \{X_1/\underline{X}\}$
N_8: $member1(X_1, \underline{Xs_3}), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
N_{17}: $subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|\underline{Xs_3}]]])$
C_{m_1} $\theta_8 = \{\underline{Xs_3}/[Y_3|Xs_5]\}$
C_{m_2} $\theta_{8'} = \{\underline{Xs_3}/[X_1|Xs_5]\}$
N_9: $member1(X_1, \underline{Xs_5}), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|[\underline{Y_3}|\underline{Xs_5}]]]])$
C_{m_1} $\theta_9 = \{\underline{Xs_5}/[Y_4|Xs_6]\}$
C_{m_2} $\theta_{9'} = \{\underline{Xs_5}/[X_1|Xs_6]\}$
N_{15}: $subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|[\underline{X_1}|\underline{Xs_5}]]]])$
N_{10}: $member1(X_1, \underline{Xs_6}), subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|[\underline{Y_3}|[\underline{Y_4}|\underline{Xs_6}]]]]])$
C_{s_2}
N_{16}: $\Box_t$
C_{m_2} $\theta_{10} = \{\underline{Xs_6}/[X_1|Xs_7]\}$
N_{11}: $subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|[\underline{Y_3}|[\underline{Y_4}|[\underline{X_1}|\underline{Xs_7}]]]]]])$
N_{13}: $subset1(Xs_4, [\underline{Y_1}|[\underline{Y_2}|[\underline{X}|[\underline{Y_3}|[\underline{X_1}|\underline{Xs_6}]]]]])$
C_{s_2}
N_{12}: $\Box_t$
C_{s_2}
N_{14}: $\Box_t$

Figure 3.7: The moded generalized SLDNF-tree of P_4 generated by Algorithm 3.1.

To prove the termination property of P_5 with this query mode is to prove the termination for the moded query $Q_0 = p(\mathcal{I})$. Applying Algorithm 3.1 generates a moded generalized SLDNF-tree as shown in Figure 3.8. The prefix from N_0 down to N_4 satisfies the conditions of LP-check and has the term-size decrease property, so clause C_{p_2} is skipped when expanding N_4. Algorithm 3.1 yields an answer *predicted-terminating* for the query Q_0. This prediction is correct. $\Box$

Example 3.12. Consider a third logic program from the Termination Competition 2007 with Problem id *LP/SGST06 - incomplete2* and difficulty rating 75%. In contrast to Example 3.11, for this program AProVE07 succeeded to prove its termination, while Polytool, NTI and TALP failed [1].

$$
\begin{array}{lll}
P_6: & f(X) \leftarrow g(s(s(s(X)))). & C_{f_1} \\
 & f(s(X)) \leftarrow f(X). & C_{f_2} \\
 & g(s(s(s(s(X))))) \leftarrow f(X). & C_{g_1}
\end{array}
$$

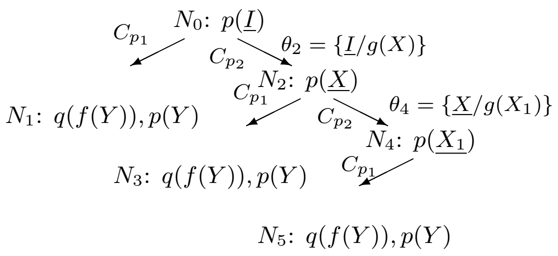

N_0: $p(\underline{I})$
C_{p_1}
C_{p_2}
$\theta_2 = \{\underline{I}/g(X)\}$
N_1: $q(f(Y)), p(Y)$
N_2: $p(\underline{X})$
C_{p_1}
C_{p_2}
$\theta_4 = \{\underline{X}/g(X_1)\}$
N_3: $q(f(Y)), p(Y)$
N_4: $p(\underline{X_1})$
C_{p_1}
N_5: $q(f(Y)), p(Y)$

Figure 3.8: The moded generalized SLDNF-tree of P_5 generated by Algorithm 3.1.

Query Mode: $f(i)$.

To prove the termination property of P_6 with this query mode is to prove the termination for the moded query $Q_0 = f(\mathcal{I})$. Applying Algorithm 3.1 generates a moded generalized SLDNF-tree as shown in Figure 3.9. C_{f_1} and/or C_{f_2} is skipped at N_4, N_5, N_6, N_9, N_{10}, N_{11}, N_{13}, N_{18}, N_{19}, N_{20}, N_{22}, N_{23}, N_{25} and N_{27}, due to the occurrence of the following prefixes which satisfy both the conditions of LP-check and the term-size decrease condition:

1. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_4 : f(\underline{X_1}) \Rightarrow_{C_{f_1}}$
2. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_5 : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
3. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_6 : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
4. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_4 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_5 : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_6 : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
5. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_1}} ... N_9 : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
6. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_1}} ... N_{10} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
7. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} ... N_9 : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_{10} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
8. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_{11} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} ... N_{13} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
9. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} ... N_{13} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
10. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_1}} ... N_2 : f(\underline{X}) \Rightarrow_{C_{f_2}} N_7 : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_{11} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
11. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} ... N_{18} : f(\underline{X_2}) \Rightarrow_{C_{f_1}}$
12. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} ... N_{19} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
13. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} ... N_{18} : f(\underline{X_2}) \Rightarrow_{C_{f_2}} N_{19} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
14. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_1}} ... N_{20} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} ... N_{22} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
15. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} ... N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_2}} ... N_{22} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
16. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} ... N_{16} : f(\underline{X_1}) \Rightarrow_{C_{f_2}} N_{20} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
17. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} ... N_{23} : f(\underline{X_1}) \Rightarrow_{C_{f_1}} ... N_{25} : f(\underline{X_2}) \Rightarrow_{C_{f_1}} ... N_{27} : f(\underline{X_3}) \Rightarrow_{C_{f_1}}$
18. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} ... N_{27} : f(\underline{X_3}) \Rightarrow_{C_{f_2}}$
19. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} ... N_{25} : f(\underline{X_2}) \Rightarrow_{C_{f_2}}$
20. $N_0 : f(\underline{I}) \Rightarrow_{C_{f_2}} N_{14} : f(\underline{X}) \Rightarrow_{C_{f_2}} ... N_{23} : f(\underline{X_1}) \Rightarrow_{C_{f_2}}$

Since there is no derivation satisfying the conditions of LP-check while violating the term-size decrease condition, Algorithm 3.1 ends with an answer *predicted-terminating* for the query Q_0. This again is a correct prediction. $\square$

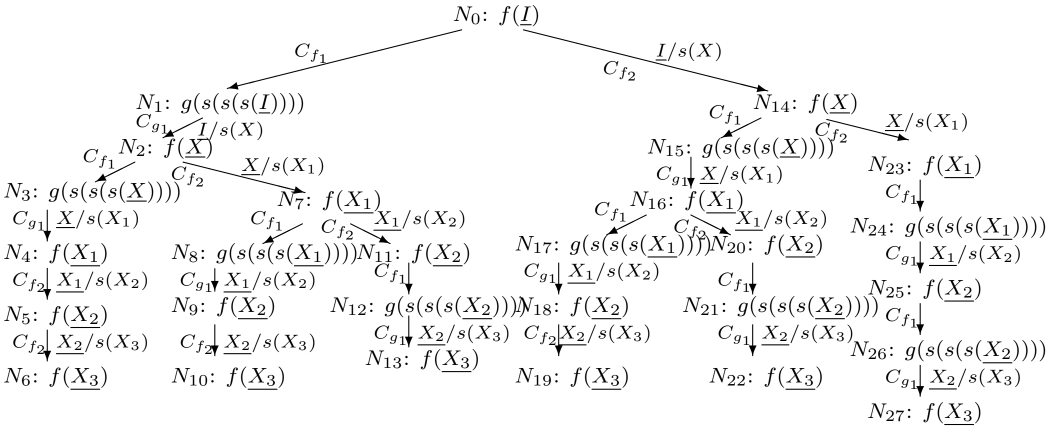


Figure 3.9: The moded generalized SLDNF-tree of P_6 generated by Algorithm 3.1.

We should remark that although for all benchmark programs of the Termination Competition 2007, choosing a repetition number $r = 3$ is enough for Algorithm 3.1 to produce a correct prediction, due to the undecidability of the termination problem, there exist cases that Algorithm 3.1 would make an incorrect prediction unless a big repetition number is used. Consider the following carefully crafted logic program:

$$\begin{array}{llr} P_7 : & p(f(X), Y) \leftarrow p(X, s(Y)). & C_{p_1} \\ & p(Z, \underbrace{s(s(...s}_{100\ items}(0)...))) \leftarrow q. & C_{p_2} \\ & q \leftarrow q. & C_{q_1} \end{array}$$

P_7 does not terminate for a moded query $Q_0 = p(\mathcal{I}, 0)$, as there is an infinite derivation

$$N_0 : p(\underline{I}, 0) \Rightarrow_{C_{p_1}} ... \ N_{101} : q \Rightarrow_{C_{q_1}} N_{102} : q \Rightarrow_{C_{q_1}} ...$$

(see Figure 3.10) which satisfies conditions (i) and (ii) of Theorem 3.3, where for any $j \geq 101$, G_{j+1} is a loop goal of G_j. Note that for any repetition number r with $3 \leq r \leq 100$, the prefix ending at N_{r-1} satisfies both the conditions of

LP-check and the term-size decrease property, where for any j with $0 \leq j < r-1$, G_{j+1} is a loop goal of G_j. However, for any $r > 100$, a prefix ending at N_{100+r} will be encountered, which satisfies the conditions of LP-check but violates the term-size decrease condition, where for any j with $101 \leq j < 100+r$, G_{j+1} is a loop goal of G_j. Therefore, Algorithm 3.1 will return *predicted-terminating* for Q_0 unless the repetition number r is set above 100.

$$\begin{tikzcd}
 & N_0\colon p(\underline{I}, 0) \arrow[d, "C_{p_1}"', "\theta_0 = \{\underline{I}/f(X_1), Y_1/0\}"] \\
 & N_1\colon p(\underline{X_1}, s(0)) \arrow[d, "C_{p_1}"', "\theta_1 = \{\underline{X_1}/f(X_2), Y_2/s(0)\}"] \\
 & N_2\colon p(\underline{X_2}, s(s(0))) \arrow[d, dotted, "C_{p_1}"'] \\
 & N_{100}\colon p(\underline{X_{100}}, \underbrace{s(s(\ \ldots\ s}_{100\ items}(0)\ \ldots\ ))) \arrow[dl, "C_{p_1}"'] \arrow[d, "C_{p_2}"', "\theta_{100} = \{Z_1/\underline{X_{100}}\}"] \\
\infty & N_{101}\colon q \arrow[d, "C_{q_1}"] \\
 & N_{102}\colon q \arrow[d, "C_{q_1}"] \\
 & \infty
\end{tikzcd}$$

Figure 3.10: The moded generalized SLDNF-tree of P_7 with a moded query $p(\mathcal{I}, 0)$.

The following results shows that choosing a sufficiently large repetition number guarantees the correct prediction for non-terminating programs.

Theorem 3.9. *Let P be a logic program and Q be a query such that P is non-terminating for Q. There always exists a number R such that Algorithm 3.1 with any repetition number $r \geq R$ produces the answer predicted-non-terminating.* $\square$

Proof. Let us assume the contrary. That is, we assume that for any number N, there exists a larger number r such that Algorithm 3.1 for P with query Q and repetition number r produces the answer $predicted - terminating$. This means that for all $r \geq 2$ the prefix of form 3.3 for some input variable I at Q. This means that D does not satisfy condition (ii) of Theorem 3.6. However, since P is non-terminating for Q, by Corollary 3.1 MT_Q has at least one infinite branch of form 3.3 satisfying conditions (i) and (ii) of Theorem 3.3. We have a contradiction and thus conclude the proof. □

The same result applies for any terminating concrete query Q. That is, there always exists a number R such that Algorithm 3.1 with any $r \geq R$ produces the answer *terminating* or *predicted-terminating* when P is terminating for Q. The proof for this is simple. When P is terminating for a concrete query Q, the (moded) generalized SLDNF-tree for Q is finite. Let R be the number of nodes of the longest branch in the tree. For any $r \geq R$, Algorithm 3.1 will produce the answer *terminating* or *predicted-terminating*, since no branch will be cut by LP-check. However, whether the above claim holds for any moded query Q when P is terminating remains an interesting open problem.

3.4 Experimental Results

We evaluated our termination prediction technique on a benchmark of 301 Prolog programs. First, we describe the benchmark and our experimental results using a straightforward implementation of Algorithm 3.1. Then, we define a pruning technique to reduce the size of the SLDNF-derivation produced by our prediction. Finally, we conclude this section with a comparison between the state-of-the-art termination and non-termination analyzers and our termination prediction tool.

Our benchmark consists of the majority of programs from the termination competition of 2007. Because the termination competition contains programs with non-logical operations such as arithmetics, 23 programs from the competition are omitted. For most of these programs neither termination nor non-termination could be shown by any of the tools in the competition. Our benchmark consists of 301 programs with moded queries: 244 terminating and 57 non-terminating programs. The most accurate termination analyzer of the competition, AProvE [23], proves termination of 238 benchmark programs. The non-termination analyzer NTI [38, 37] proves non-termination of 42 programs. Because the prediction does not produce a termination or non-termination proof, our goal is to outperform the analyzers of the competition.

We implemented our tool, TPoLP: Termination Prediction of Logic Programs, in SWI-prolog [59]. TPoLP is freely available from [2]. The moded SLDNF-derivation is represented by a graph and acquired by following Algorithm 3.1 as described in the last section. The derivation is initialized with the moded query and extended for a maximum of 4 minutes until a branch is found that does not have the term-size decrease property or until all branches are cut. If no answer is produced in 4 minutes the analysis is stopped and the result is a timeout. To improve the efficiency of the analysis, a number of optimizations were implemented, such as constant time access to the nodes and the arcs of the graph representing the derivation. The experiments have been performed using SWI-Prolog 5.6.40, on an Intel Core2 Duo 2,33GHz, 2 Gb RAM.

Repetition number:	2	3	4
Correct Predictions:	291	271	234
Wrong Predictions:	7	0	0
Out of time/memory	3	30	67
Average time (Sec):	1.7	24.9	59.3
Average Nb of nodes:	5553	56467	106057
Average Nb of lp cuts:	4082	22666	38066

Table 3.1: Predictions: full table available at [2]

Table 3.1 gives an overview of the predictions with repetition numbers two, three and four. Two does not suffice as a repetition number because some of the predictions are wrong and we want very reliable predictions. When the repetition number is set to at least three, all predictions made for the benchmark are correct. This shows that in practice, there is no need to increase the repetition number any further. However, the cost of predicting the termination behavior is very high. About 10% of the programs break the time limit of four minutes.

The component of the algorithm taking most of the time differs from program to program. When a lot of branches are cut by LP-check, constructing the LP cuts is usually the bottleneck. For programs with a low amount of LP cuts, most of the time is spend on constructing the SLDNF-derivation. Because some of the constructed SLDNF-derivations count more than a million nodes, we added a component to remove redundant parts of the derivation on backtracking. To reduce the analysis time needed by TPoLP, we implemented a pruning technique to reduce the size of the SLDNF-derivations.

3.4.1 Pruning

To explain the intuition behind the pruning technique, we revisit Example 14. Note that the paths N_0 down to N_6, N_0 down to N_{10}, and so on, contain the same rules in a different order. Therefore, it seems that most of these paths are redundant. The idea to prune redundant paths is to ignore clauses if they are already applied to a similar parent or loop goal.

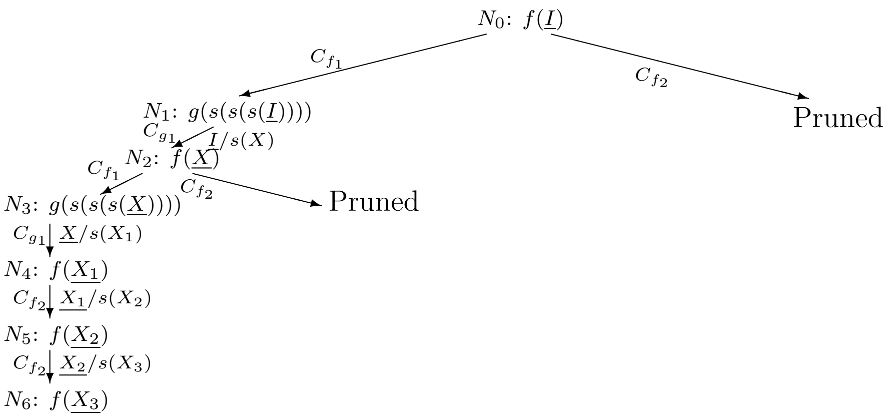
Two versions of the pruning technique are defined. The first version prunes only on variant loop goals. The second version prunes on all loop goals.

Definition 3.9 (Pruning)**.** *Let G2 be a loop goal of G1 (for which the selected literals have the same symbol string). Then, all clauses except for the looping clause that have already been applied at G2, are skipped at G1 during backtracking.* □

Example 3.13 (SLDNF-Derivation of Example 3.12 with pruning)**.** *See Figure 3.11. The algorithm starts by creating nodes N_0 down to N_6. All rules unifying with $f(X_3)$ at N_6 are cut by LP-check. Without the pruning technique we would now apply C_{f_2} at node N_2. Because N_4 is a loop goal with identical symbol string of N_2, we ignore clause C_{f_2} at node N_2. We also prune clause C_{f_2} at node N_0 because N_4 is a loop goal with identical symbol string of N_0.*

No more rules can be applied, and the algorithm with pruning ends with the correct answer: predicted terminating. Because all the loop goals in the derivation have identical symbol strings, both pruning techniques construct the same derivation for this example. The derivations with pruning contain 7 nodes, while the derivation without pruning (see Example 3.12) counts 28 nodes.

□



N_0: $f(\underline{I})$
C_{f_1}
C_{f_2}
Pruned
N_1: $g(s(s(s(\underline{I}))))$
C_{g_1}
$\underline{I}/s(X)$
N_2: $f(\underline{X})$
C_{f_1}
C_{f_2}
Pruned
N_3: $g(s(s(s(\underline{X}))))$
C_{g_1} $\underline{X}/s(X_1)$
N_4: $f(\underline{X_1})$
C_{f_2} $\underline{X_1}/s(X_2)$
N_5: $f(\underline{X_2})$
C_{f_2} $\underline{X_2}/s(X_3)$
N_6: $f(\underline{X_3})$

Figure 3.11: SLDNF-derivation of Example 3.12 with pruning

Table 3.2 gives an overview of our predictions with three as a repetition number: without pruning, with pruning on variant loop goals and pruning on all loop goals. The table shows that pruning is a good tradeoff between the precision and the efficiency of the prediction. When pruning on variant loop goals, the size

	No pruning	Pruning variants	Pruning Loop Goals
Correct Predictions:	271	296	297
Wrong Predictions:	0	0	3
Out of time/memory	30	5	1
Average time (Sec)	24.9	4.4	0.05
Average Nb of nodes	56467	7800	140
Average Nb of lp cuts	22666	380	2.8

Table 3.2: Pruning: full tables available at [2].

of the derivations drops considerably, while all predictions for the benchmark are still correct. Because the derivations are much smaller, more than 98% of the predictions finish within the time limit. When pruning on all loop goals, the size of the derivation is only a fraction of the derivation without pruning. However, we lose accuracy by pruning on all loop goals: three non-terminating programs are predicted to be terminating. The accuracy does not improve with a higher repetition number.

Answer	TPoLP	AProvE [23]	NTI [38, 37]	Polytool	TALP [36]
Terminating (244):	239	238	0	206	164
Non-terminating (57):	57	0	42	0	0

Table 3.3: Comparison between TPoLP and (non-)termination analyzers.

Table 3.3 gives a comparison between our predictions with pruning on variant loop goals and the state-of-the-art termination and non-termination analyzers. Note that our tool, TPoLP, is the only tool for Logic programs which analyzes both for termination and non-termination. The results are very satisfying. We correctly predict the termination property of all 301 benchmark programs, except for five programs which broke the time limit. It is also worth noticing that for all programs of the benchmark, either an analyzer finds a termination or non-termination proof or a correct prediction is made by our tool. This shows that our prediction tool can be a very useful addition to any termination or non-termination analyzer.

3.5 Related Work

Most approaches to the termination problem are *norm-* or *level mapping-based* in the sense that they perform termination analysis by building from the source code of a logic program some well-founded termination conditions/constraints in terms of norms (i.e. term sizes of atoms of clauses), level mappings, interargument size relations and/or instantiation dependencies, which when solved, yield a termination proof (see, e.g., [16] for a survey and more recent papers [5, 9, 10, 18, 22, 27, 30, 34]). Another main stream is *transformational* approaches, which transform a logic program into a term rewriting system (TRS) and then analyze the termination property of the resulting TRS instead [4, 6, 23, 31, 36, 44, 46, 54]). All of these approaches are used for a termination proof; i.e., they compute sufficient termination conditions which once satisfied, lead to a positive conclusion *terminating*. Recently, Payet and Mesnard [38, 37] propose an approach to computing sufficient non-termination conditions which when satisfied, lead to a negative conclusion *non-terminating*. A majority of these termination/non-termination proof approaches apply only to positive logic programs.

Our approach presented in this chapter differs significantly from previous termination analysis approaches. First, we do not make a termination proof, nor do we make a non-termination proof. Instead, we make a termination prediction (see Figure 3.1) – an approximation approach to attacking the undecidable termination problem. Second, we do not rely on static norms or level mappings, nor do we transform a logic program to a term rewriting system. Instead, we focus on detecting infinite SLDNF-derivations with the understanding that a logic program is terminating for a query if and only if there is no infinite SLDNF-derivation with the query. We have established a necessary and sufficient characterization of infinite SLDNF-derivations with arbitrary (concrete or moded) queries, introduced a new loop checking mechanism, and developed an algorithm that predicts termination of general logic programs with arbitrary queries by identifying potential infinite SLDNF-derivations. Since the algorithm implements the necessary and sufficient conditions (the characterization) of an infinite SLDNF-derivation, its prediction is very effective; see Examples 3.7 - 3.12. Especially, Examples 3.10 - 3.12 demonstrate that our algorithm can make a correct prediction even though some of the state-of-the-art analyzers like AProVE07 [23], NTI [38, 37], Polytool [35] and TALP [36] may fail to prove/disprove it.

Our termination prediction approach uses a loop checking mechanism (a loop check) to implement a characterization of infinite SLDNF-derivations. Well-known loop checks include VA-check [8, 53], OS-check [11, 32, 45], and VAF-checks [49, 52]. All apply to positive logic programs. In particular, VA-check applies to function-free logic programs, where an infinite derivation is characterized by a sequence of selected *variant subgoals*. OS-check identifies an infinite derivation with a sequence of selected subgoals with the same predicate symbol *whose sizes do not decrease*. VAF-checks take a sequence of selected *expanded variant subgoals* as major characteristics of an infinite derivation. Expanded variant subgoals are variant subgoals except that some terms may grow bigger. In this chapter, a new loop check mechanism, LP-check, is introduced in which an infinite derivation is identified with a sequence of *loop goals*. Most importantly, enhancing LP-check with Condition (c) leads to the first loop check for moded queries.

3.6 Summary

We have presented an approximation framework for attacking the undecidable termination problem of logic programs, as an alternative to current termination/non-termination proof approaches. We introduced an idea of termination prediction, established a necessary and sufficient characterization of infinite SLDNF-derivations with arbitrary (concrete or moded) queries, built a new loop checking mechanism, and developed an algorithm that predicts termination of general logic programs with arbitrary queries. We demonstrated the effectiveness of the termination prediction with representative examples including ones borrowed from the Termination Competition 2007.

Our prediction approach can be used standalone, e.g., it may be incorporated into Prolog as a termination debugging tool; or it is used along with some termination/non-termination proof tools (see Figure 3.1).

Reference

The work in this chapter was published in TPLP [50].

Chapter 4

Non-termination Analysis for Logic Programs

This chapter defines a new non-termination analysis which reuses the analysis scheme introduced in the last chapter. Our work has been inspired both by the work on termination prediction and by the work on non-termination inference [38] and its implementation NTI. The analysis scheme of the previous Chapter is used to produce a finite representation of the computation for a given logic program and moded query. We introduce a new non-termination condition expressed in terms of this finite representation of the computation. We prove its correctness and extend it to increase its applicability.

It turns out that our characterization of non-terminating computations is more precise than that of NTI. We have implemented the technique and performed extensive experiments with it on the basis of the benchmark of the termination analysis competition of 2007 [1]. The experiments show that our technique has a 100% success-rate on this benchmark, outperforming the only competing analyzer, NTI.

The chapter is organized as follows. In Section 4.1, we present our conditions implying non-termination and show that we are able to derive classes of non-terminating queries. In Section 4.2, we present our experimental evaluation and we compare our analyzer with the non-termination inference tool NTI.

4.1 A new non-termination condition

In this section, we present a new non-termination analysis technique for general logic programs with moded queries. A moded query Q corresponds to a set of concrete queries, called the *denotation* of Q. We consider a program *non-terminating* w.r.t. a moded query, if the denotation of the query contains at least one concrete query that has an infinite branch in its generalized SLDNF-tree. In the following definition, $Term_P$ denotes the set of ground terms defined by a program P.

Let $\underline{I_1}, \ldots, \underline{I_n}$ be all input variables occurring in a moded atom A. Let $t_1, \ldots, t_n \in Term_P$. $A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n})$ denotes the concrete atom obtained by replacing the input variables $\underline{I_1}, \ldots, \underline{I_n}$ by the terms $t_1, \ldots, t_n$.

Definition 4.1. *Let A be an atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables. The **denotation** of A is*

$$Den(A) = \{A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n}) \mid t_i \in Term_P,\ t_i \textit{ is ground}\}. \qquad \square$$

This concept can be adapted to moded goals, literals in a straightforward way. Note that the denotation of a concrete atom is a singleton containing the atom itself.

4.1.1 The moded more general relation

To prove non-termination, we prove that a path between two nodes N_b and N_e in a moded SLDNF-derivation can be repeated infinitely often. To find such a path, we check three properties. Because the rules in the path must be applicable independent of the values of the input variables, no substitutions on the input variables may occur in the path from N_b to N_e. Because this path should be a loop, the selected literal of N_b must be an ancestor of the selected literal of N_e. Finally, a special more general relation for moded atoms must hold between the selected literals of N_b and N_e. We will show that these three conditions imply non-termination.

A moded atom A is *moded more general* than a moded atom B, if any atom in the denotation of A is more general than some atom in the denotation of B. This means that for each atom I in the denotation of A, there must exists a substitution θ such that $I\theta$ is in the denotation of B.

Definition 4.2. *A moded atom A is **moded more general** than a moded atom B w.r.t. a program P, $A \triangleright B$, iff:*

$$\forall I \in Den(A), \exists I' \in Den(B) : I \text{ is more general than } I' \qquad \square$$

We illustrate this moded more general relation with some small examples.

Example 4.1. *The binary tree program succeeds if the argument of the query represents a binary tree.*

```
bin(empty).
bin(tree(L,_,R)):- bin(L), bin(R).
```

The following moded more general relations hold w.r.t. the this program:

- $bin(X) \triangleright bin(\underline{I})$

  The denotation of $bin(X)$ *only contains the atom itself, which is more general than any atom in the denotation of* $bin(\underline{I})$*, e.g.* $bin(empty)$*.*

- $bin(tree(tree(\underline{In}, V1, Xn), V2, Y)) \triangleright bin(tree(\underline{I}, V3, tree(X, V4, empty)))$

  Every atom A *of the denotation of* $bin(tree(tree(\underline{In}, V1, Xn), V2, Y))$*, is more general than some atom* B *denoted by* $bin(tree(\underline{I},, tree(X,, empty)))$*. For example,* $bin(tree(tree(empty, V1, Xn), V2, Y))$ *(obtained by* $\{\underline{In} \setminus empty\}$*) is more general than* $bin(tree(tree(empty, empty, empty),$ $V3, tree(X, V4, empty)))$ *(obtained by* $\{\underline{I} \setminus tree(empty, empty, empty)\}$*).* $\square$

Because the denotation of a moded atom is in general infinite, we cannot check this property for every atom in the denotation. However, there is a syntactic sufficient condition to check if the moded more general relation holds between two given moded atoms A and B. The condition is based on a particular kind of unifiability of the atoms.

We introduce the following notations. Let $InVar_P$ be the set of input variables and Var_P the set of normal variables. To every $\underline{I} \in InVar_P$ we associate a fresh normal variable I. Let $Term_P^+$ denote the set of all terms constructible in the underlying language of P augmented with the variables $\{I \mid \underline{I} \in InVar_P\}$.

Proposition 4.1. *Let* A *and* B *be two moded atoms. Let* A_1 *and* B_1 *be renamings of these atoms such that they have no shared variables. Let* A_2 *and* B_2 *denote variants of* A_1 *and* B_1 *in which every input variable* $\underline{I}$ *is replaced by* I*. Let* $N_1^a, \ldots, N_n^a$ *be a subset of the normal variables in* A_1 *and* $I_1^b, \ldots, I_m^b$ *be the fresh variables associated to the input variables in* B_2*.*

If A_2 *and* B_2 *are unifiable with a substitution* $\gamma = \{N_1^a \setminus t_1, \ldots, N_n^a \setminus t_n,$ $I_1^b \setminus t_1^+, \ldots, I_m^b \setminus t_m^+\}$ *with* $t_1, \ldots, t_n \in Term_P$ *and* $t_1^+, \ldots, t_m^+ \in Term_P^+$*, then* A *is moded more general then* B*.* $\square$

Proof. Let $\alpha = \{N_1^a \setminus t_1, \ldots, N_n^a \setminus t_n\}$ and $\beta = \{I_1^b \setminus t_1^+, \ldots, I_m^b \setminus t_m^+\}$. Because $I_1^b, \ldots, I_m^b$ cannot occur in $t_1, \ldots, t_n$, $\gamma = \beta \circ \alpha$, and by unifiability, $A_2\alpha\beta = B_2\alpha\beta$. Moreover, since B_2 does not contain $N_1^a, \ldots, N_n^a$, $B_2\alpha\beta = B_2\beta$, and since $A_2\alpha$ does not contain $I_1^b, \ldots, I_m^b$, $A_2\alpha\beta = A_2\alpha$. Thus, $A_2\alpha = B_2\beta$.

Let A_c be an element of $Den(A_1)$. Then, there exists a substitution $\psi = \{I_1^a \setminus s_1, \ldots, I_k^a \setminus s_k\}$, where $\underline{I_1^a}, \ldots, \underline{I_k^a}$ are all input variables of A_1, $s_1, \ldots, s_k \in Term_P$ and $s_1, \ldots, s_k$ are ground, such that $A_c = A_2\psi$.

Now consider the atom $B_c = B_2\beta\psi$. First, $B_c \in Den(B_1)$. This is because β replaces all I_j^b of B_2 by terms t_j^+. These terms t_j^+ may contain variables I_l^a of A_2, but these are all substituted to ordinary ground terms $s_l \in Term_P$ by ψ.

Finally, we have that $A_c\alpha = A_2\psi\alpha = A_2\alpha\psi = B_2\beta\psi = B_c$. Note that $A_2\psi\alpha = A_2\alpha\psi$ because no s_i of ψ can contain a variable N_j^a of α, nor can any t_i of α contain a variable I_j^a of ψ. Thus A_c is more general than an element of $Den(B_1)$. $\square$

We clarify this property by checking the moded more general relations of the last example.

Example 4.2. *The moded atoms of the last example are already variable disjunct. To check if the moded more general relation holds, we have to check if the atoms are unifiable with a substitution of the correct forms.*

- $bin(X) = bin(I)$ *with substitution:* $\{I \setminus X\}$
- $bin(tree(tree(In, V1, Xn), V2, Y)) = bin(tree(I, V3, tree(X, V4, empty)))$ *with substitution:* $\{I \setminus tree(In, V1, Xn), V2 \setminus V3, Y \setminus tree(X, V4, empty)\}$
  $\square$

4.1.2 Non-termination of moded more general loops

If a moded SLDNF-derivation contains a path without substitutions on input variables, such that the ancestor relation and the moded more general relation hold between the first and last selected literal in that path, we call this path a *moded more general loop.* We will show that a moded more general loop implies non-termination.

Definition 4.3. *In a moded SLDNF-derivation D, nodes $N_i : G_i$ and $N_j : G_j$ are a* ***moded more general loop****, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, iff:*

- *No substitutions on input variables occur in the path from N_i to N_j.*

- $L_i^1 \prec_{anc} L_j^1$.
- $L_j^1 \rhd L_i^1$ □

Note that when no confusion can occur, we may omit writing the goal in the moded more general loop.

A moded more general loop, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, corresponds to an infinite loop for every concrete goal in the denotation of G_i.

Theorem 4.1 (Sufficiency of the moded more general loop)**.** *Let $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ be a moded more general loop in a moded SLDNF-derivation D of a program P and a moded query I. The sequence of clauses from N_i to N_j, $\langle C_1, \dots, C_n \rangle$, can be repeated infinitely often for any goal in $Den(G_i)$.* □

Proof. Because L_i^1 is an ancestor of L_j^1, the literals of N_i different from L_i^1 are irrelevant for the derivation between N_i and N_j and hence for the repeatability.

Because no substitutions on input variables occur in the path from N_i down to N_j, $\langle C_1, \dots, C_n \rangle$ is applicable to any atom in the denotation of L_i^1. Obviously, this path is also applicable to any atom A, which is more general than some atom B in the denotation of L_i^1. Furthermore, after applying $\langle C_1, \dots, C_n \rangle$ to A, the resulting selected literal is more general than the selected literal after applying $\langle C_1, \dots, C_n \rangle$ to B.

As $L_j^1 \rhd L_i^1$, any atom in $Den(L_j^1)$ is more general than some atom in $Den(L_i^1)$.

Therefore, let S be the union of $Den(L_i^1)$ and all more general atoms. Then, $\langle C_1, \dots, C_n \rangle$ is applicable to any atom of S, and after applying these clauses, the selected literal of the resulting goal is again an atom of S. Thus, this sequence of clauses is infinitely often applicable to elements of S. □

We illustrate this non-termination condition with the *binary tree* program.

Example 4.3 (Non-termination proof of *binary tree*)**.** *Figure 4.1 shows the moded SLD-tree of $bin(X)$ using LP-check with repetition number 2.*

The path from N_0 to N_2 satisfies the conditions of Definition 4.3:

- *There are no substitutions on input variables from N_0 to N_2.*
- *The selected atom of N_0 is an ancestor of the selected atom at N_2.*
- $bin(L) \rhd bin(X)$

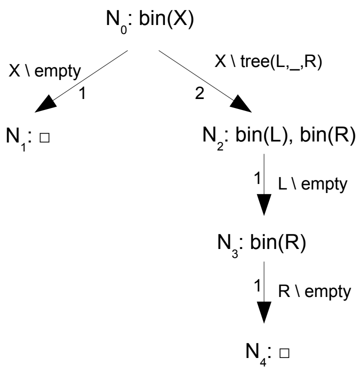


Figure 4.1: Moded SLD-tree for the *binary tree* program, using $LP - check$ with repetition number 2

so $N_0 \overset{mmg}{\rightarrow} N_2$ *is a moded more general loop. Therefore, non-termination of this example is proven by Theorem 4.1.* □

Observe that Theorem 4.1 can straightforwardly be generalized to conclude non-termination for any goal that is more general than an element of $Den(G_i)$. In particular, the analysis is not restricted to goals with ground inputs: Theorem 4.1 also holds for an "extended" denotation of G_i, with non-ground inputs. However, if the denotation is extended so that an input variable can be replaced by a non-ground term, we have to be careful not to introduce aliased variables on the input positions.

Example 4.4 (Aliasing). `p(f(X),Y):- p(X,f(Y)).`

When applying the above clause to a moded goal $p(X, \underline{I})$*, the result is* $p(X', f(\underline{I}))$*. These atoms satisfy the moded more general relation. However, if we replace the input variable* I *with the aliased variable* X *in the original goal, the result is* $p(X', f(f(X')))$*, which is not more general than any atom in the denotation of* $p(X', f(\underline{I}))$*.* □

4.1.3 Input-generalizations

Our experimental evaluation (see Section 4.2) shows that for many non-terminating programs, non-termination can be proven using the moded more general loop. But, the next example shows that there is room for further improvement.

Example 4.5 (Termination behavior of $flat$).

```
flat(niltree, nil).
flat(tree(X, niltree, XS), cons(X, YS)) :- flat(XS, YS).
flat(tree(X, tree(Y, YS1, YS2), XS), ZS) :-
        flat(tree(Y, YS1, tree(X, YS2, XS)), ZS).
```

This program, flat, flattens a binary tree into a list denoted with the cons notation. To flatten the tree, the program repeatedly moves one element from the left to the right subtree until the left subtree is empty. When the left subtree is empty, we proceed by processing the right subtree. If the first argument of the query is a variable, this program loops w.r.t. the third clause.

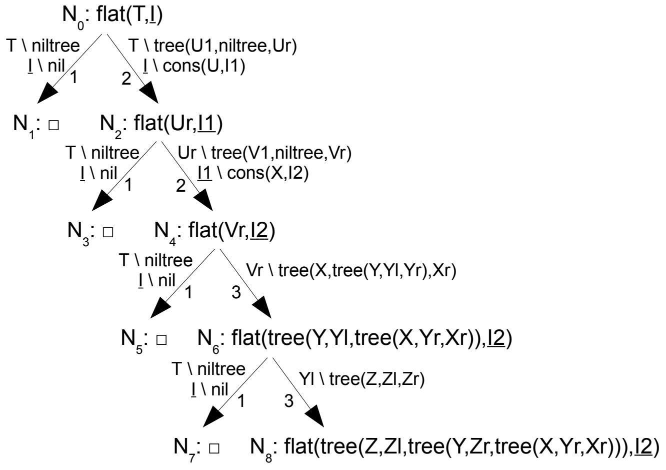


Figure 4.2: Moded generalized SLDNF-tree with LP-check of $flat$ (Example 4.5)

Figure 4.2 shows a part of the moded generalized SLDNF-tree constructed for moded query $flat(T, \underline{I})$ *using LP-check with repetition number 3. No nodes in the derivations satisfy Definition 4.3. Indeed, in deriving* N_2 *and* N_4 *substitutions on input variables are applied. Furthermore,* N_6 *is not moded more general than* N_4 *and* N_8 *is not moded more general than* N_6 *because the derivation steps replace a variable by a compound term.* □

To prove non-termination for programs such as $flat$, we define an *input-generalization.* This input-generalization is such that proving non-termination of an input-generalized goal implies non-termination of the original goal.

Definition 4.4. *We say that A^α is an input-generalization of an atom A, if there exist terms $t_1, \dots, t_n$ in A and fresh input variables $\underline{I_1}, \dots, \underline{I_n}$ such that $A^\alpha = A(\underline{I_1} \to t_1, \dots, \underline{I_n} \to t_n)$ and $Var(A^\alpha) \cap Var((t_1, \dots, t_n)) = \emptyset$.* □

Example 4.6 (Input generalizations).

- $bin(tree(\underline{I}, \underline{I_1}))$ *is an input-generalization of* $bin(tree(\underline{I}, tree(X, empty)))$
- $bin(\underline{I_2})$ *is an input-generalization of* $bin(tree(\underline{I}, \underline{I_1}))$
- $bin(tree(\underline{I_3}, X))$ *is not an input generalization of* $bin(tree(tree(X, Y), X))$ *This last example refers to the condition of the empty intersection of the variable sets. We return to this condition in Example 4.7.* □

To check if a path is non-terminating w.r.t. an input-generalized goal, we define an *input-generalized derivation.* This derivation is constructed by applying a path in a given derivation to the input-generalized selected literal of the first node in the path.

Definition 4.5. *Let D be a moded SLDNF-derivation $N_i, \dots, N_j$, such that $L_i^1 \prec_{anc} L_j^1$. Let $\langle C_1, \dots, C_n \rangle$ be the sequence of clauses applied from N_i to N_j and let A^α be an input-generalization of L_i^1.*

The ***input-generalized derivation*** *D' for A^α, is constructed by applying the sequence of clauses $\langle C_1, \dots, C_n \rangle$ to A^α. The* ***input-generalized nodes*** *N_i^α and N_j^α are the top and bottom nodes of D', respectively.* □

Next, we prove that non-termination of the input-generalized derivation implies non-termination of the original goal. First we introduce two lemmas.

Lemma 4.1. *Let A^α be an input generalization of A, then $A \rhd A^\alpha$.* □

Proof. Let $\underline{I_1}, \dots, \underline{I_n}$ be the input variables of A and $\underline{I_{n+1}}, \dots, \underline{I_m}$ be the new introduced input variables in A^α. For every concrete atom A_c in $Den(A)$, $\underline{I_1}, \dots, \underline{I_n}$ are replaced by ground terms. To construct an atom A_c^α of $Den(A^\alpha)$, for which A_c is more general then A_c^α, one replaces $\underline{I_1}, \dots, \underline{I_n}$ by the same values as in A_c and $\underline{I_{n+1}}, \dots, \underline{I_m}$ by instances of the corresponding terms, $t_{n+1}, \dots, t_m$, in A_c. Due to the condition that $Var(A^\alpha) \cap Var((t_{n+1}, \dots, t_m)) = \emptyset$, A_c is more general than A_c^α. □

Example 4.7. *To explain the condition on the intersection of the variables in Definition 4.4, consider the atom $A = a(X, f(X))$. If we omit the condition on the variables, we could consider $A^\alpha = a(X, \underline{I})$ as an input generalization. $Den(A) = \{a(X, f(X))\}$ and $a(X, f(X))$ is not more general than any element in $Den(a(X, \underline{I}))$. So, the property that $A \rhd A^\alpha$ would not hold.* □

Lemma 4.2. *Let A and B be atoms such that $A \triangleright B$ and let every atom in $Den(B)$ be non-terminating w.r.t. a program P, then, every atom in $Den(A)$ is non-terminating w.r.t. P.* □

Proof. Every atom of $Den(A)$ is more general than a non-terminating atom. □

Corollary 4.1 (Non-termination with input-generalization)**.** *Let $N_i : G_i$ and $N_j : G_j$ be nodes in a derivation D of a program P for a moded query I, such that $L_i^1 \prec_{anc} L_j^1$, and let N_i^α and N_j^α be input-generalized nodes in an input-generalized derivation D' of N_i and N_j for A.*

If $N_i^\alpha \overset{mmg}{\rightarrow} N_j^\alpha$, then every concrete goal in the denotation of G_i is non-terminating w.r.t. program P.

Proof. Follows from Theorem 4.1 and the two previous lemmas. □

We illustrate these input-generalizations by revisiting the $flat$ example.

Example 4.8 (Non-termination of $flat$)**.** *To prove non-termination, we generalize node N_6 to $flat(tree(Y, Yl, \underline{In}), \underline{I2})$, by changing the subterm $tree(X, Yr, Xr)$ to a new input variable $\underline{In}$.*

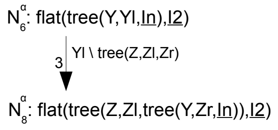

N₆^α: flat(tree(Y,Yl,In),I2)
3
Yl \ tree(Z,Zl,Zr)
N₈^α: flat(tree(Z,Zl,tree(Y,Zr,In)),I2)

Figure 4.3: Input-generalized SLDNF-derivation of $flat$

Figure 4.3, shows the input-generalized moded SLDNF-derivation for $flat(tree(Y, Yl, \underline{In}), \underline{I2})$. This derivation is a moded more general loop: $N_6^\alpha \overset{mmg}{\rightarrow} N_8^\alpha$. Therefore, non-termination of the program flat w.r.t. the concrete goals in the denotation of the goal of N_6 is proven by Corollary 4.1.

Obviously, not every query in the denotation of $flat(T, \underline{I})$ reaches node N_6. However, when we apply all substitutions on input variables between nodes N_0 and N_6, we obtain a class of non-terminating top level queries. In this case $flat(T, cons(\underline{U}, cons(\underline{X}, \underline{I2})))$. To construct a class of non-terminating queries that reaches this node and therefore loops, it is enough to apply all substitutions on input variables occurring between N_0 and N_6. □

Note that a concrete query in the denotation of a moded query might not reach the moded more general loop. However, classes of non-terminating top level queries can be obtained by applying all substitutions on the input variables between the root and the first node of the moded more general loop. In the last example, this class of top level queries is $flat(T, cons(\underline{U}, cons(\underline{X}, \underline{I2})))$.

4.2 Experimental evaluation

To evaluate our approach, we implemented a non-termination analyzer $P2P$, *from Prediction to Proof*, based on Corollary 4.1. We tested $P2P$ on a benchmark of 48 non-terminating pure logic programs. First, we describe our analyzer and the benchmark. Then, we compare our tool with the non-termination inference tool NTI [38].

4.2.1 $P2P$: *from Prediction to Proof*

We implemented $P2P$ in SWI-prolog [59]. $P2P$ consists of two components. First, the implementation of the termination prediction approach from the previous chapter, $TPoLP$, constructs the moded SLDNF-derivation and predicts the termination behavior. If $TPoLP$ predicts a derivation to be non-terminating, the second component tries to prove non-termination in the derivation.

To prove non-termination, $P2P$ checks if the derivation contains a moded more general loop or it uses a backtracking search to attempt to construct an input generalized derivation that contains a moded more general loop. Although many input generalizations can be constructed, proving non-termination in a derivation can be done rather efficiently. This is because the LP-cuts made by $TPoLP$ correctly identify an infinite loop if the repetition number is sufficiently high. Therefore, instead of checking the conditions of the moded more general loop between all pairs of nodes in the derivation, it suffices to check these conditions for the pairs of nodes of the LP-cut.

4.2.2 Benchmark of Termination Problems

Our benchmark consists of the non-terminating pure logic programs from the termination competition of 2007. The benchmark and the results from the tools that participated in the competition are available online [1]. The benchmark of the termination competition contains around 300 logic programs and moded queries representing different challenges in termination and non-termination analysis. A few programs from the competition are omitted because they contain non-logical operations such as arithmetics. The competition benchmark contains some doubles. These were also omitted. The benchmark contains 48 non-terminating programs. All programs contain between 2 and 15 clauses, except for binary4, which contains 41 clauses. Note that the benchmark contains some complex programs, e.g. *pl*.5.2.2 which is a Turing machine simulator. The only other non-termination analyzer, *NTI* [38], proves non-termination for 45 benchmark programs.

Name program	*P2P*	Size	Time	*NTI*	Name program	*P2P*	Size	Time	*NTI*
ackermann-ioi	V	9	0.33	V	permutation-fb	V	22	0.26	V
bad sublist	V	33	0.29	V	pl1.1	V	8	0.25	V
binary4	V	12	0.27	V	pl3.1.1	V	12	0.30	V
delete-bff	V	13	0.31	V	pl3.5.6	V	13	0.31	V
der-fb	V	22	0.29	V	pl4.0.1-oooi	V	33	0.27	V
doublehalfpred	V	38	0.28	V	pl4.5.2	V	481	0.36	V
example4-2	V	4	0.23	V	pl4.5.3a	V	10	0.29	V
flatlength-fbf	V	14	0.23	V	pl4.5.3b	V	10	0.24	V
flatlength-ffb	V	19	0.23	V	pl4.5.3c	V	11	0.27	V
flat-oi	V	9	0.26	X	pl5.2.2	V	59	0.27	V
frontier-fb	V	12	0.27	V	pl7.6.2.a	V	39	0.27	X
ifdiv	V	19	0.29	V	pl7.6.2.b	V	45	0.33	X
in-bf	V	18	0.29	V	quicksort-fb	V	72	0.26	V
inorder-fb	V	4	0.27	V	quicksort-oi	V	74	0.26	V
insert-bff	V	22	0.29	V	reverse-fb	V	9	0.32	V
log2a-oi	V	35	0.25	V	select-bff	V	8	0.32	V
log2b-oi	V	29	0.28	V	slowsort-fb	V	123	0.27	V
mapcolor	V	23	0.31	V	slowsort-oi	V	26	0.26	V
member-bf	V	8	0.27	V	sublist-bf	V	30	0.21	V
mergesort	V	171	0.28	V	subset-bf	V	21	0.23	V
mergesort-oi	V	54	0.28	V	subset-fb	V	14	0.26	V
mergesort_variant	V	15	0.23	V	suffix-bf	V	9	0.25	V
minimum-fb	V	8	0.29	V	transpose2	V	6	0.28	V
naive reverse-fb	V	8	0.37	V	tree_member-bf	V	12	0.28	V

Table 4.1: Benchmark of non-terminating pure logic programs.

Table 1 shows our experimental evaluation on this benchmark using LP-check with pruning, with 4 as a repetition number. The result of our tool is given in the column *P2P*, *V* denotes that non-termination is proven while *X* denotes that no non-termination proof was found. The result of *NTI* is given in the column *NTI*. The columns *Size* and *Time* show the size in the number of nodes of the SLDNF-tree and the analysis time in seconds, respectively.

The results are very satisfactory. For all programs in the benchmark, non-termination is proven and a class of non-terminating queries can be constructed. The analyzer is very fast. Any benchmark program is analyzed in less than a second and the memory use never exceeds a few megabytes.

As stated, these experiments have been performed using 4 as a repetition number. When we use 3 as a repetition number, our tool fails to prove non-termination of programs *pl*7.6.2.*a* and *pl*7.6.2.*b*. These are two erroneous implementations of a path find algorithm. When using 2 as repetition number, proving non-termination fails for about 25% of the benchmark programs. This is no surprise, since the loss of precision using 2 as a repetition number, was also encountered in the experimental evaluation of $TPoLP$.

4.2.3 Comparison with NTI

To infer non-terminating queries, NTI first transforms a given program into a binary program using *binary unfoldings*. Then, it compares the head and body of the clauses in the binary program with a special more general relation. If this relation holds, non-termination is proven.

The binary unfolding of a program represent the calls made during program execution. Thus, it corresponds to comparing the selected literals in our symbolic computation. The binary unfolding of a program can be computed using a fixpoint operator.

The special more general relation used by NTI, $\triangle$*-more general*, is based on the notion of *derivation neutral (DN) filters*. These filters are functions defining, for a clause and argument position, which terms have no influence on the applicability of the clause. Furthermore, if the head atom satisfies the filter, the body atom must satisfy the filter as well. We explain NTI's non-termination condition and compare it with our approach using some small examples.

Example 4.9 (Recursive clause of reverse-fb)**.**

```
rev([H|T],Temp,Res):- rev(T,[H|Temp],Res).
```

In this clause, the second argument is not replaced by a more general one. Therefore, NTI needs a DN filter to prove non-termination. The applicability of the clause does not depend on the value of $Temp$*, so we can use the trivial filter, instance of* X*, for the second argument position. We can also use this filter for the last argument position. Therefore, NTI concludes that this clause is non-terminating for each goal where the first argument is more general than* $[H|T]$ *and the second and third argument are instances of* X*.* □

These DN filters cannot depend on the names of the variables. Therefore, they cannot express that two argument must contain a common subterm.

Example 4.10 (Variable independent filters)**.** `a(X,X):- a(s(X),s(X)).`

Both arguments are replaced by more specific ones and the applicability does not depend on the value of X*. However, since both arguments must be bound to the same term, NTI fails to prove non-termination of this example.* □

Instead of comparing all the arguments independently, our approach compares the selected literals. Therefore, our condition does not have this restriction.

Because *NTI* requires that each argument is either replaced by a more general one or satisfies a DN filter, *NTI* fails to prove non-termination if in one argument, a subterm is replaced by a more general one while another subterm is replaced by a more specific one. This is because of the requirement that if the head atom satisfies the filter, the body atom needs to satisfy it as well.

Example 4.11 (Looping clause of *flat-oi*). *In the third clause of flat-oi in Example 4.8, the first argument of this clause contains two such subterms.* `XS` *is replaced by* `tree(X,YS2,XS)` *and* `tree(Y, YS1, YS2)` *is replaced by* `YS1`*.* □

Because we allow arguments to contain both input and ordinary variables, our condition does not have this restriction.

Table 1 shows that *NTI* fails to prove non-termination of 3 programs. These 3 programs are examples of the two classes of problems that are illustrated by Examples 4.10 and 4.11. The actual results on the termination competition were worse for *NTI*, as we have rewritten some programs that *NTI* could not parse.

4.3 Summary

We introduced a new approach to non-termination analysis of logic programs based on a finite, symbolic derivation tree for a moded query. This symbolic tree represents the derivation trees of all concrete queries denoted by the moded query. To prove non-termination we look for a loop in this symbolic derivation tree. We implemented this approach and evaluated it on a benchmark of 48 non-terminating programs from the termination competition of 2007. Our tool, *P2P*, proves non-termination of all benchmark programs. We have shown that our technique improves on the results of the only non-termination analyzer developed before our work, *NTI*, and that we can handle 2 new classes of programs.

Reference

The work in this chapter was presented at ICLP 2009 and is published in [58].

Chapter 5

Non-termination Analysis for Logic Programs using Types

In this chapter, we identify two classes of programs for which previous non-termination analyzers fail and we extend our non-termination analysis to handle such programs.

A first limitation of both NTI and $P2P$ is that they only detect non-terminating derivations if, within these derivations, some fixed sequence of clauses can be applied repeatedly.

Example 5.1. *The program, longer, loops for any query* `longer(L)`*, with* `L` *a non-empty list of zeros.*

```
longer([0|L]):-
    zeros(L),
    longer([0,0|L]).
zeros([]).
zeros([0|L]):- zeros(L).
```

The list in the recursive call is longer than the original one and thus, the number of applications of the recursive clause for $zeros/1$ increases in each recursion. Therefore, no fixed sequence of clauses can be repeated infinitely and previous non-termination analyzers fail to prove non-termination of this example. □

Example 5.1, which was also discussed in the introduction, shows a program that violates this restriction. We overcome this limitation by using non-failure information. *Non-failure analysis* [17] detects classes of goals that can be guaranteed not to fail, given mode and type information. Its applications include inferring minimal computational costs, guiding transformations and debugging. To use the information provided by non-failure analysis in the non-termination analysis of [58], type information must be added to the symbolic derivation tree. We add this information using regular types [60]. Non-failure information allows to abstract away from the details on how success was reached for these computations. For the *longer* program, non-failure analysis proves that *zeros*/1 cannot fail for the inferred types. This will allow us to prove non-termination for the *longer* program.

Example 5.2.

```
append([],L,L).
append([H|T],L,[H|R]):- append(T,L,R).
```

The query `append(X,X,X)` *succeeds once with a computed answer substitution* `X/[]`*. The program loops after backtracking. NTI needs a filter for the second argument to prove non-termination of this example. However, these filters are only allowed on argument positions that don't share variables with terms on other positions. Because all argument positions contain the same variable, NTI fails to prove non-termination of this example.*

Similarly, P2P needs an input-generalization to prove non-termination for this example. Due to the shared variables, this input-generalization is not allowed for this query. □

Another limitation of previous non-termination analyzers is related to aliased variables. Non-termination analyzers developed before 2011 fail to prove non-termination of programs such as the one in Example 5.2. We will show that program specialization can be used to overcome this limitation. In addition to these two classes of programs, there are combinations of them that yield a fairly large class of new programs that we can prove non-terminating using non-failure analysis combined with program specialization.

Note that from here on in the thesis, we will focus on definite logic programs. The extension to general logic programs should be fairly easy, but we see little advantages in practice. Therefore, we will only consider definite logic programs in the rest of the thesis.

The chapter is organized as follows. Section 5.1 introduces preliminaries. Section 5.2 extends the moded SLD-tree with types and introduces a special derivation step to handle non-failing goals. Section 5.3 adapts our non-termination condition for this setting and illustrates that Program Specialization can be used to further improve the power of our non-termination analyzer. Finally, Section 5.4 summarizes the chapter.

5.1 Preliminaries

5.1.1 Types

In this chapter, we extend the moded SLD-tree defined in the previous chapter by adding a special operation for nodes with a non-failing selected atom. To use non-failure information, type information of the partially instantiated goals must be available. Since logic programs are untyped, this type information will be inferred. Many techniques and tools exist to infer type definitions for a given logic program, for example [12]. We will describe types using *regular types* as defined in [60]. The set of type symbols is denoted by Σ_τ.

Definition 5.1. *Let P be a logic program. A **type rule** for a type symbol $T \in \Sigma_\tau$ is of the form $T \rightarrow c_1; \ldots; c_i; f_{i+1}(\bar{\tau}_{i+1}); \ldots; f_k(\bar{\tau}_k)$, $(k \geq 1)$, where $c_1, \ldots, c_i$ are constants, $f_{i+1}, \ldots, f_k$ are distinct function symbols associated with T and $\bar{\tau}_j$ $(i+1 \leq j \leq k)$ are tuples of corresponding type symbols of Σ_τ. A **type definition** $\mathcal{T}$ is a finite set of type rules, where no two rules contain the same type symbol on the left hand side, and for each type symbol occurring in the type definition, there is a type rule defining it.* □

A *predicate signature* declares one type symbol for each argument position of a given predicate. A *well-typing* $\langle \mathcal{T}, \mathcal{S} \rangle$ of a program P, is a pair consisting of a type definition $\mathcal{T}$ and a set $\mathcal{S}$ containing one predicate signature for each predicate of P, such that the types of the actual parameters passed to a predicate are an instance of the predicate's signature. For this chapter, we use *PolyTypes* ([12]) to infer signatures and type definitions. In [12], it has been proven that these inferred signatures and type definitions are a well-typing for the given program.

Types allow to give a more precise description of the possible values during evaluation at different argument positions. The set of terms constructible from a certain type definition, is called the *denotation of the type.* We represent the denotation of type T by $Den(T)$.

Given a type definition $\mathcal{T}$, for every type T defined by $\mathcal{T}$, we introduce an infinite set of fresh variables Var_T. For any two types $T_1 \neq T_2$, we impose that $Var_{T_1} \cap Var_{T_2} = \emptyset$.

Definition 5.2. *Let $T_1, \ldots, T_n$ be types defined by a type definition $\mathcal{T}$ defining type symbols $\bar{\tau}$. The* ***denotation*** *$Den(T_i)$ of $T_i (1 \leq i \leq n)$, defined by $T_i \rightarrow c_1; \ldots; c_j; f_{j+1}(\bar{\tau}_{j+1}); \ldots; f_k(\bar{\tau}_k)$, is recursively defined as:*

- *every variable in Var_{T_i} is an element of $Den(T_i)$*
- *every constant $c_p, 1 \leq p \leq j$, is an element of $Den(T_i)$*
- *for every type term $f_p(\tau_1, \ldots \tau_l), j < p \leq k$,: if $t_1 \in Den(\tau_1), \ldots, t_l \in Den(\tau_l)$, then $f_i(t_1, \ldots, t_l) \in Den(T_i)$* □

Example 5.3. *For the program longer of Example 5.1, the following types and signatures are inferred by PolyTypes:*

$T_{lz} \rightarrow [\ ]; [T_0 \mid T_{lz}]$	$longer(T_{lz})$
$T_0 \rightarrow 0$	$zeros(T_{lz})$

$Den(T_0)$ is the set containing 0 and all variables of Var_{T_0}. $Den(T_{lz})$ contains: $[\ ], Y, [0,0], [0,X,0], [X|Y], \ldots$, with $X \in Var_{T_0}$ and $Y \in Var_{T_{lz}}$. □

5.1.2 Non-failure analysis and program specialization

Given type and mode information, *non-failure analysis* detects goals that can be guaranteed not to fail, i.e. they either succeed or go in an infinite loop. We use the non-failure analysis technique of [17] in our non-termination analysis.

Example 5.4. *For Example 5.1, non-failure analysis proves that $zeros/1$ is non-failing if its argument is an input mode of type T_{lz}. It cannot show that $longer/1$ with an argument of type T_{lz} is non-failing, because $longer([\ ])$ fails.* □

Program specialization aims at transforming a given logic program into an equivalent but more efficient program for a certain query. This query can be partially instantiated, yielding a specialized program for a class of queries. Program specialization has received a lot of attention in the community, see for example [26]. In this chapter, we use the specialization tool *ECCE*, [26], to generate specialized programs.

5.2 Moded-Typed SLD-trees and the NFG transition

5.2.1 Moded-typed SLD-trees and loop checking

As stated in the introduction, we want to prove non-termination of partially instantiated queries, given mode and type information of the variables in the query. In such a partially instantiated query, variables representing unknown terms are labeled *input modes*. To every input mode, a type is assigned. An atom Q is a *moded-typed* atom if some variables of Q are input modes, otherwise, it is a *concrete* atom. The terms represented by input modes in an atom Q are restricted to terms of their respective type. Note that the user does not have to declare the types associated to input modes, because the inferred well-typing declares a unique type for every subterm of atoms defined by the program.

Definition 5.3. *Let P be a program, $Q = p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)$ a moded-typed atom and $\langle \mathcal{T}, \mathcal{S} \rangle$ a well-typing for P. The* ***moded-typed SLD-tree*** *of P for Q, is a pair $(GT_{G_0}, \langle \mathcal{T}, \mathcal{S} \rangle)$, where GT_{G_0} is the generalized SLD-tree for $P \cup \{\leftarrow p(\underline{I_1}, ..., \underline{I_m}, T_1, ..., T_n)\}$, with each $\underline{I_i}$ being a special variable not occurring in any T_j. The special variables $\underline{I_1}, ..., \underline{I_m}$ are called* ***input variables****.* □

As mentioned, an input variable $\underline{I}$ may be substituted by a term $f(t_1, \dots, t_n)$. If $\underline{I}$ is substituted by $f(t_1, \dots, t_n)$, all variables in $t_1, \dots, t_n$ also become input variables. In particular, when unifying $\underline{I}$ with a normal variable X, X becomes an input variable. We refer to Figure 5.1 for an illustration of (part of) a moded-typed SLD-tree.

A moded-typed atom A represents a set of concrete atoms, called the *denotation* of A. This is the set of atoms obtained by replacing the input modes by arbitrary terms of their respective type.

Definition 5.4. *Let A be a moded-typed atom with $\underline{I_1}, \dots, \underline{I_n}$ as its input variables with types $T_1, \dots, T_n$, respectively. The* ***denotation*** *of A, $Den(A)$, is*

$$\{A(t_1 \to \underline{I_1}, \dots, t_n \to \underline{I_n}) \mid t_1 \in Den(T_1), \dots, t_n \in Den(T_n)\} \qquad \square$$

Note that non-termination of an atom implies non-termination of all more general atoms. Therefore, we consider a moded-typed atom A to represent all atoms of the denotation of A, as well as all more general atoms. We call this set the *extended denotation* of A.

Definition 5.5. *Let A be a moded-typed atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its input variables with types $T_1, \ldots, T_n$, respectively. The* ***extended denotation*** *of A, $Ext(A)$, is*

$$\{I \mid I' \in Den(A), I \text{ is more general than } I'\} \qquad \square$$

Note that this extended denotation includes the atoms of the denotation and that this concept can be adapted to moded-typed goals in a straightforward way.

Moded-typed SLD-trees represent all derivations for a certain class of queries. For most programs and classes of queries, this tree is infinite. In order to obtain a finite tree, we can use a complete loop check, to cut all infinite derivations from the tree.

Example 5.5. *Figure 5.1 shows the moded-typed SLD-tree for longer, using LP-check with repetition number 2.*

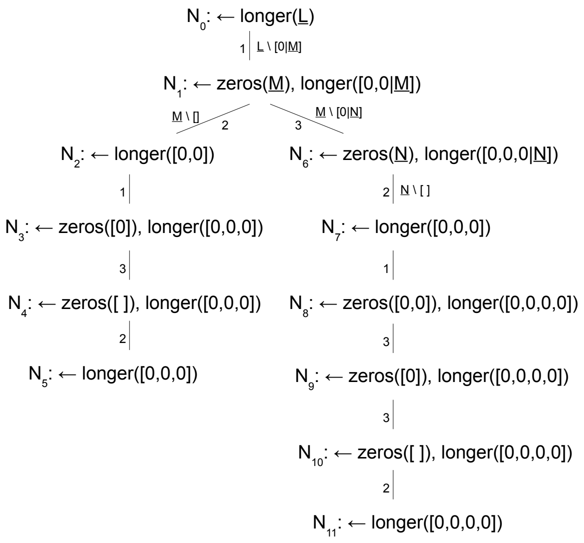

N0: ← longer(L)
1
L \ [0|M]
N1: ← zeros(M), longer([0,0|M])
M \ []
2
3
M \ [0|N]
N2: ← longer([0,0])
N6: ← zeros(N), longer([0,0,0|N])
1
2
N \ [ ]
N3: ← zeros([0]), longer([0,0,0])
N7: ← longer([0,0,0])
3
1
N4: ← zeros([ ]), longer([0,0,0])
N8: ← zeros([0,0]), longer([0,0,0,0])
2
3
N5: ← longer([0,0,0])
N9: ← zeros([0]), longer([0,0,0,0])
3
N10: ← zeros([ ]), longer([0,0,0,0])
2
N11: ← longer([0,0,0,0])

Figure 5.1: Moded-typed SLD-tree for the atom *longer*, using *LP − check* with repetition number 2

At node N_5, LP-check cuts clause 1 because node N_5 is a loop goal of node N_2. Likewise, LP-check cuts clause 3 at node N_6 and clause 1 at node N_{11}. $\square$

5.2.2 The NFG transition

As illustrated in Example 5.1, one of the main limitations of current non-termination analysis techniques is that non-termination can only be proven for programs and queries repeating a fixed sequence of clauses. To overcome this limitation, we extend the moded-typed SLD-tree of Definition 5.3 by adding a special transition NFG, to treat non-failing selected atoms. This transition allows to abstract from the sequence of clauses needed to solve the non-failing atom. Note that during the evaluation of a non-failing atom, normal variables may be bound to terms of their respective type. Therefore, we approximate the application of this unknown sequence of clauses by substituting all normal variables in the selected atom by fresh input variables of the correct type. In the next subsection, the correlation between these resulting moded-typed SLD-trees and concrete SLD-trees is given.

Definition 5.6. *Let $N_i :\leftarrow A_1, A_2, \ldots, A_n$ be a node in a moded-typed SLD-tree, with A_1 a non-failing atom. Let $V_1, \ldots, V_m$ be all normal variables of A_1, corresponding to types $T_1, \ldots, T_m$, respectively. Let $\underline{V_1}, \ldots, \underline{V_m}$ be new input variables of types $T_1, \ldots, T_m$, respectively. Then, an NFG transition, $N_i :\leftarrow A_1, \ldots, A_n \Rightarrow_{NFG} N_{i+1} :\leftarrow A_2\theta, \ldots, A_n\theta$, can be applied to N_i, with substitution $\theta = \{V_1 \setminus \underline{V_1}, \ldots, V_m \setminus \underline{V_m}\}$.* □

Note that we can also allow an NFG transition to be applied to a goal containing only one atom, resulting in a refutation.

Definition 5.7. *Let P be a logic program, Q a moded-typed atom and $\langle \mathcal{T}, \mathcal{S} \rangle$ a well-typing for P. The* ***moded-typed SLD-tree with NFG*** *is the pair $(GT_{(NFG,G_0)}, \langle \mathcal{T}, \mathcal{S} \rangle)$, where $GT_{(NFG,G_0)}$ is obtained from the generalized SLD-tree GT_{G_0} for $P \cup \leftarrow Q$, by, at each node: $N_i :\leftarrow A_1, A_2, \ldots, A_n$, with A_1 a non-failing atom, additionally applying the NFG transition.* □

Without proof, we state that $LP - check$ is also a complete loop check for moded-typed SLD-trees with NFG. From here on, when we refer to a moded-typed SLD-tree, we mean the finite part of a moded-typed SLD-tree with NFG, obtained by using $LP - check$ with some repetition number, r.

Example 5.6. *Figure 5.2 shows the moded-typed SLD-tree of Example 5.1 for the query $longer(\underline{L})$ using 3 as a repetition number. The selected atom at node N_1, $zeros(\underline{M})$, is non-failing and can be solved using an NFG transition. Since there are no normal variables in the selected atom, there are no substitutions for this transition. At node N_3, the NFG transition is applied as well.*

At node N_4, $LP - check$ cuts the derivation because of the chain of loop goals $N_0 \Rightarrow_1 \ldots N_2 \Rightarrow_1 \ldots N_4 \Rightarrow_1$. □

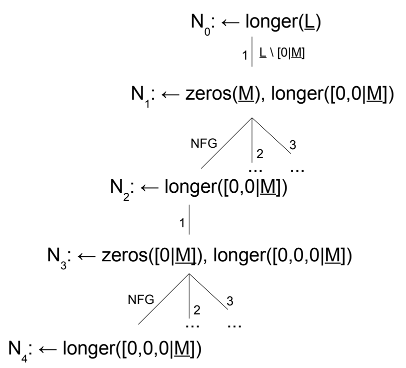


Figure 5.2: Moded-typed SLD-tree with NFG of the *longer* program

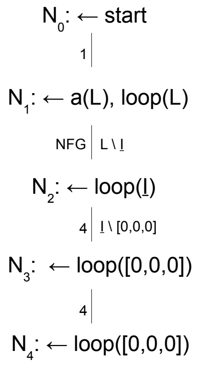


Figure 5.3: Moded-typed SLD-tree of Example 5.8

5.2.3 Correlation with concrete queries

The correlation between derivations in moded-typed SLD-trees without NFG transitions and concrete SLD-derivations is rather simple. For every derivation in such a moded-typed SLD-tree from $N_0 : G_0$ to $N_i : G_i$, there exists a non-empty subset of the concrete goals of the extended denotation of G_0 on which the same sequence of clauses can be applied. This subset of goals can be obtained from G_0 by applying the composition of all substitutions on input variables from N_0 to N_i.

Example 5.7. *In the derivation from N_0 to N_8 in Figure 5.1, there are three substitutions on input variables: $\underline{L} \setminus [0|\underline{M}]$, $\underline{M} \setminus [0|\underline{N}]$ and $\underline{N} \setminus []$. Applying these substitutions to the query longer($\underline{L}$) gives the concrete query longer$([0, 0])$.* $\square$

For typed SLD-trees with NFG transitions, there is no such clear correspondence. Solving a non-failing atom corresponds to a potentially infinite sequence of clause applications. Thus, for an NFG transition $N_i : G_i \rightarrow_{NFG} N_j : G_j$, it is possible that no concrete state corresponding to G_j ever occurs in a concrete derivation. However, this will not cause any problems for our analysis. If our analysis detects and reports a non-terminating computation for a branch in the moded-typed SLD-tree which has an NFG transition, then a corresponding concrete atom either finitely succeeds or non-terminates. In both cases, reporting non-termination is correct.

There is a second problem with the correspondence between NFG transitions and concrete derivations. Input variables introduced by NFG give an overestimation of the possible values after evaluating the non-failing selected atom in a concrete derivation. Therefore, substitutions on such input variables further down the tree might be impossible and thus, cannot be allowed. We illustrate this with an example.

Example 5.8.

```
start:- a(L), loop(L).
a([0]).
a([0,0]).
loop([0,0,0]):- loop([0,0,0]).
```

T_{lz} is a correct type definition for all arguments of atoms. Non-failure analysis ([17]) shows that `a(L)` *is non-failing if* `L` *is a free variable.*

Figure 5.3 shows a part of the moded-typed SLD-tree using repetition number 2 for the query `start`*. At node N_1, the non-failing atom* `a(L)` *is solved by replacing the variable* `L` *by a new input variable $\underline{\mathtt{I}}$. The path from N_3 to N_4 is a loop. However, this loop cannot be reached because $\underline{\mathtt{I}}$ is substituted by* `[0,0,0]`*, but the program only allows it to be* `[0]` *or* `[0,0]`*.* □

We introduce the following proposition showing the correlation between moded-typed SLD-trees and concrete SLD-trees.

Proposition 5.1. *Let T be a moded-typed SLD-tree with NFG for a goal G_0 and N_i a node of T. Let θ be the composition of all substitutions on input variables from N_0 to N_i.*

If the derivation from N_0 to N_i does not contain a substitution on input variables introduced by NFG transitions, then all goals in $Ext(G_0\theta)$ can be evaluated to goals in $Ext(G_i)$ or loop w.r.t. an NFG transition. □

Proof. By induction on the length of the derivation.

Base case. Since we assume the query to consist only of one atom, applying an NFG transition to the top level goal G_0 results in a refutation. Since G_0 is non-failing, every element of $Ext(G_0)$ either results in a refutation or loops w.r.t. this NFG transition. Applying a clause, $N_0 : G_0 \Rightarrow_C N_1 : G_1$, to G_0 only succeeds if the actual values of the input variables are such that the substitutions on the corresponding terms succeed. Therefore, let σ_T be the substitutions on input variables corresponding to this derivation step, then clause C is applicable to $Ext(G_0\sigma_T)$.

Induction step for NFG application Assume the lemma holds for $N_0 : G_0 \Rightarrow N_i : G_i$ and that we apply $N_i : G_i \Rightarrow_{NFG} N_{i+1} : G_{i+1}$. Since A_i^1 is non-failing, this atom either succeeds or loops w.r.t. the program. If it succeeds, free variables of A_i^1 may be substituted with terms of their corresponding type. Due to Definition 5.5, the resulting goal is an element of $Ext(G_{i+1})$ whether or not the variables are bound during the evaluation of A_i^1. Thus, if θ is the composition of substitutions on input variables from N_0 to N_i, goals in $Ext(G_0\theta)$ either loops w.r.t. an NFG transition or can be evaluated to a goal in $Ext(G_{i+1})$

Let θ_i be the substitutions on input variables from N_0 to N_i. Then, every atom in the extended denotation of $A_i^1\theta_i$ is non-failing and thus either succeeds or loops. If the free variables in the selected atom of G_0 are bound, they are bound to an element of their respective type and thus included in $Ext(G_1)$. If some free variables stay unbound they are still elements of $Ext(G_1)$ because of Definition 5.5. Thus, every element in $G_0\theta_i$ can be evaluated to a goal in $Ext(G_{i+1})$ or loops w.r.t. an NFG transition.

Induction step for clause application Assume the lemma holds for $N_0 : G_0 \Rightarrow N_i : G_i$ and that we apply $N_i : G_i \Rightarrow_c N_{i+1} : G_{i+1}$. Let σ_T and be the substitutions on input variables due to this derivation step. Let θ_i be the substitutions on input variables from N_0 to N_i.

As in the base case, this clause is applicable to G_i if the actual values of the input variables are such that the substitutions on the corresponding terms succeed and thus, they are applicable to $Ext(G_i\sigma_T)$. Therefore, every goal in $Ext(G_0\theta_i\sigma_T)$ can be evaluated to a goal in $Ext(G_{i+1})$ or loops w.r.t. an NFG transition. $\square$

5.3 Typed non-termination analysis with non-failing goals

In this section, we reformulate our non-termination conditions of the previous chapter for moded-typed SLD-trees with NFG. We prove that these conditions imply non-termination. We then show that program specialization can be used to further extend the applicability of these conditions.

5.3.1 Non-termination of inclusion loops

To prove non-termination, we prove that a path between two nodes N_b and N_e in a moded-typed SLD-derivation can be repeated infinitely often. To find such a path, we check three properties. Because the rules in the path must be applicable independent of the values of the input variables, no substitutions on the input variables may occur in the path from N_b to N_e. Because this path should be a loop, A_b^1 must be an ancestor of A_e^1. Finally, because the goals corresponding to N_e must be able to repeat the loop, $Ext(A_e^1)$ must be a subset of $Ext(A_b^1)$. We can show that these three conditions imply non-termination.

Definition 5.8. *In a moded-typed SLD-derivation with NFG D, nodes $N_i : G_i$ and $N_j : G_j$ are an* ***inclusion loop****, $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$, if:*

- *No substitutions on input variables occur in the path from N_i to N_j.*
- $A_i^1 \prec_{anc} A_j^1$.
- $A_j^1 \rhd A_i^1$ □

An inclusion loop $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ corresponds to an infinite loop for every goal of the extended denotation of G_i.

Theorem 5.1 (Sufficiency of the inclusion loop)**.** *Let $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ be an inclusion loop in a moded-typed SLD-derivation with NFG D of a program P and a moded-typed query Q, then, every goal of the extended denotation of G_i is non-terminating w.r.t. P.* □

Proof. In the path from N_i to N_j, non-failing atoms can be solved using the NFG transition. If such an atom loops, the theorem holds. Thus, it is sufficient to prove that the path from N_i to N_j can be applied infinitely often if all selected atoms, from N_i to N_j, corresponding to an NFG transition finitely succeed.

Because A_i^1 is an ancestor of A_j^1, only the selected atom of N_i influences if the sequence of clauses can be repeated infinitely often.

Because no substitutions on input variables occur in the path from N_i down to N_j, the corresponding sequence of derivation steps and NFG transitions is applicable to any atom in $Ext(A_i^1)$.

Since $A_j^1 \rhd A_i^1$, this proves non-termination. □

Due to Proposition 5.1, an inclusion loop $N_i : G_i \overset{mmg}{\rightarrow} N_j : G_j$ proves non-termination for a subset of concrete goals of the extended denotation of the top level goal, if no substitutions on input variables introduced by NFG transitions, occur in the path from N_0 to N_i.

Example 5.9. *The moded-typed SLD-tree of Figure 5.2 can be used to prove non-termination of the longer program. The path from N_2 to N_4 satisfies the conditions of the inclusion loop. There are no substitutions on input variables between these nodes, the ancestor relation holds and $Ext(longer([0, 0, 0 \mid \underline{M}]))$ is a subset of $Ext(longer([0, 0 \mid \underline{M}]))$.*

Since there are no input variables added by NFG transitions, Proposition 5.1 proves that all atoms in $Ext(longer([0 \mid \underline{M}]))$, with $\underline{M}$ of type T_{lz}, are non-terminating. □

Because extended denotations are usually infinite sets, the third condition of Definition 5.8, $A^1_j \rhd A^1_i$, is hard to verify. In the previous chapter, we introduced a syntactic condition, based on a unifiability test between atoms, to verify whether the inclusion holds. This syntactic condition, formulated in Proposition 1 of the previous chapter, is still applicable in our current setting and can be used to automatically check Definition 5.8.

5.3.2 Input-generalizations

In the previous chapter, we defined *input-generalizations* to increase the applicability of our non-termination condition. The next example shows that this is also needed in the current setting.

Example 5.10. *We will illustrate the need for input generalizations with an adapted version of the program of Example 5.1.*

```
longer([0|L], X):-
        zeros(L),
        longer([0,0|L], f(X)).
zeros([]).
zeros([0|L]):- zeros(L).
```

For this program, the following types and signatures are inferred:

$T_{lz} \rightarrow [\,]; [T_0 \mid T_{lz}]$	$T_f \rightarrow f(T_f)$	$longer(T_{lz}, T_f)$
$T_0 \rightarrow 0$		$zeros(T_{lz})$

Figure 5.4 shows the moded-typed SLD-derivation for the moded-typed query `longer(`$\underline{\texttt{L}}$`, X)`*, with* $\underline{\texttt{L}}$ *of type T_{lz}, using 3 as a repetition number. All pairs of nodes in the derivations fail the third condition of Definition 5.8.* □

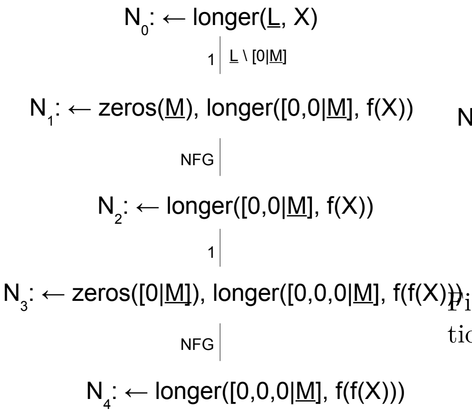

N0: ← longer(L, X)
1 L \ [0|M]
N1: ← zeros(M), longer([0,0|M], f(X))
NFG
N2: ← longer([0,0|M], f(X))
1
N3: ← zeros([0|M]), longer([0,0,0|M], f(f(X)))
NFG
N4: ← longer([0,0,0|M], f(f(X)))

Figure 5.4: Moded-typed SLD-tree of Example 5.10

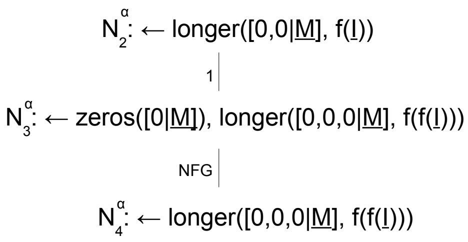

N2α: ← longer([0,0|M], f(I))
1
N3α: ← zeros([0|M]), longer([0,0,0|M], f(f(I)))
NFG
N4α: ← longer([0,0,0|M], f(f(I)))

Figure 5.5: Input-generalized derivation

To prove non-termination for such a program, we define an input-generalization. This input-generalization is such that proving non-termination of an input-generalized selected atom implies non-termination of the original goal.

Definition 5.9. *A moded-typed atom A^α is an input-generalization of a moded-typed atom A, if there exist terms $t_1, \ldots, t_n$ in A and fresh moded-typed variables $\underline{I_1}, \ldots, \underline{I_n}$ with the same types as $t_1, \ldots, t_n$, respectively, such that $A^\alpha = A(\underline{I_1} \to t_1, \ldots, \underline{I_n} \to t_n)$ and $Var(A^\alpha) \cap Var((t_1, \ldots, t_n)) = \emptyset$.* □

Example 5.11. *Let A be the atom $longer([0, X, X], Y)$:*

- *$longer([0|\underline{I}], Y)$ is an input-generalization of A*
- *$longer(\underline{I_1}, \underline{I_2})$ is an input-generalization of A*
- *$longer([0, \underline{I}, X], Y)$ is not an input-generalization of A. This last example refers to the condition of the empty intersection of the variable sets.* □

To check if a path is non-terminating w.r.t. an input-generalized goal, we define an *input-generalized derivation.* This derivation is constructed by applying a path in a given derivation to the input-generalized selected atom of the first node in the path.

Definition 5.10. *Let N_i and N_j be nodes in a moded-typed SLD-derivation with NFG D, such that $A_i^1 \prec_{anc} A_j^1$. Let $\langle C_1, \ldots, C_n\rangle$ be the sequence of derivation* *steps and transitions from N_i to N_j and let A^α be an input-generalization of A_i^1.*

If $\langle C_1, \ldots, C_n \rangle$ can be applied to $\leftarrow A^\alpha$, the resulting derivation is called the ***input-generalized derivation*** *D' for A^α. The* ***input-generalized nodes*** *N_i^α and N_j^α are the first and last node of D', respectively.* $\square$

Note that such a path in a moded-typed SLD-derivation is applicable to the input-generalized atom, if the selected atoms at all NFG transitions are still non-failing.

Non-termination of the input-generalized derivation implies non-termination of the original goal. Before we prove that this holds, we introduce an alternative definition of the extended denotation and give a lemma.

The following definition is an alternative characterization of the extended denotation. This definition contains the same elements as the extended denotation of Definition 5.5, up to variable renamings. This characterization will also be used in the following proofs.

Definition 5.11. *Let A be a moded-typed atom with $\underline{I_1}, \ldots, \underline{I_n}$ as its typed variables with types $T_1, \ldots, T_n$, respectively. The* ***extended denotation*** *of A, $Ext(A)$, is*

$$\textit{Let } S = \{A(t_1 \rightarrow \underline{I_1}, \ldots, t_n \rightarrow \underline{I_n}) \mid t_1 \in Den(T_1), \ldots, t_n \in Den(T_n), t_1 \textit{ is ground}, \ldots, t_n \textit{ is ground}\}$$

$$Ext(A) = \{I \mid I' \in S, I \textit{ is more general than } I'\}$$ $\square$

Lemma 5.1. *Let A^α be an input-generalization of A, then $A \rhd A^\alpha$.* $\square$

Proof. Note that again it is enough to prove that every element in S of Definition 5.11 is an element of $Ext(A^\alpha)$.

Let $\underline{I_1}, \ldots, \underline{I_n}$ be the input variables of A and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ be the newly introduced moded-typed atoms in A^α. For every concrete atom A_c of $Den(A)$, $\underline{I_1}, \ldots, \underline{I_n}$ are replaced by ground terms. To construct an atom A_c^α of $Den(A^\alpha)$, for which A_c is more general then A_c^α, one replaces $\underline{I_1}, \ldots, \underline{I_n}$ by the same values as in A_c and $\underline{I_{n+1}}, \ldots, \underline{I_m}$ by instances of the corresponding terms, $t_{n+1}, \ldots, t_m$, in A_c. Due to the condition that $Var(A^\alpha) \cap Var((t_{n+1}, \ldots, t_m)) = \emptyset$, A_c is more general than A_c^α and thus A_c is an element of $Ext(A^\alpha)$. $\square$

Example 5.12. *To explain the condition on the intersection of the variables in Definition 5.9, consider the atom $A = a(X, f(X))$. If we omit the condition on the variables, we could consider $A^\alpha = a(X, \underline{I})$ as an input-generalization.*

$Ext(A)$ contains $a(X, f(X))$, which is not an element of $Ext(a(X, \underline{I}))$. So, the property that $A \rhd A^\alpha$ would not hold. □

Proposition 5.2 (Non-termination with input-generalization)**.** *Let $N_i : G_i$ and $N_j : G_j$ be nodes in a moded-typed SLD-derivation with NFG D of a program P for a query I and let A^α an input-generalization of A_i^1. Let N_i^α and N_j^α be input-generalized nodes in the input-generalized derivation D' for A^α.*

If $N_i^\alpha \overset{mmg}{\rightarrow} N_j^\alpha$, then every concrete goal of the extended denotation of G_i is non-terminating w.r.t. program P. □

Proof. According to Lemma 5.1, $A \rhd A^\alpha$ and according to Theorem 5.1, every goal in $Ext(A^\alpha)$ is non-terminating. Since $N_i^\alpha \overset{mmg}{\rightarrow} N_j^\alpha$, N_i is an ancestor of N_j and thus, every goal in $Ext(G_i)$ must be non-terminating as well. □

Example 5.13. *To prove non-termination of the program in Example 5.10, $longer([0, 0|\underline{M}], f(\underline{I}))$ can be used as an input-generalization of A_2^1 in Figure 5.4.*

Figure 5.5, shows the input-generalized derivation of N_2 to N_4 for $longer([0, 0 \mid \underline{M}], f(\underline{I}))$. This derivation is an inclusion loop: $N_2^\alpha \overset{mmg}{\rightarrow} N_4^\alpha$. Therefore, non-termination of this program w.r.t. the concrete goals of $Ext(\leftarrow longer([0, 0 \mid \underline{M}], f(X)))$ is proven by Proposition 5.2. □

5.3.3 Program specialization

In Example 5.2, we discussed a class of programs, related to aliased variables, for which current non-termination analyzers fail to prove non-termination. We illustrated that program specialization can be used to prove non-termination of such programs. The main intuition is that non-termination analysis techniques have difficulties with treating queries with aliased variables. Program specialization often reduces the aliasing, due to argument filtering. So, in the context of aliasing, applying program specialization often improves the applicability of the analysis.

Program specialization can also be used in combination with the NFG transition. When solving a non-failing atom, all variables in the atom are substituted with new input variables. These input variables give an overestimation of the possible values after evaluating the non-failing atom. Program specialization can produce more instantiated, but equivalent clauses. These more instantiated clauses give a better approximation of the possible values after evaluating the non-failing atom. We illustrate this with an example.

Example 5.14. *The following program generates the infinite list of Hamming numbers in symbolic notation. hamming/0 starts the computation initializing the list of hamming numbers to* `[s(0)]`. `hamming([N|Ns])` *keeps a list of hamming numbers, ordered from small to large. Three new hamming numbers are generated using times/3, which defines multiplication on the symbolic notation. Then, insert/3 merges these three numbers with* `Ns` *and removes duplicates, resulting in the list for the next iteration. The code for insert/3 and times/3 is omitted.*

```
hamming:- hamming([s(0)]).
hamming([N|Ns]):- times(s(s(0)),N,N2),
        times(s(s(s(0))),N,N3),
        times(s(s(s(s(s(0))))),N,N5),
        insert([N2,N3,N5],Ns,Res),
        hamming(Res).
```

PolyTypes infers the following type definition:

$$T_s \rightarrow 0; s(T_s) \qquad\qquad T_l \rightarrow [\,]; [T_s \mid T_l]$$

The arguments of hamming/1 and insert/3 are of type T_l, the other arguments are of type T_s. Non-failure analysis shows that insert/3 and times/3 are non-failing if their last arguments are free variables.

When building the moded-typed SLD-tree for the query `hamming`*, solving the non-failing atoms by applying an NFG transition, the moded-typed goal* `hamming(`$\underline{\texttt{Res}}$`)` *is obtained at the recursive call. In this goal,* $\underline{\texttt{Res}}$ *is introduced by an NFG transition. When applying clause 2 to this goal,* $\underline{\texttt{Res}}$ *is substituted by a compound term and thus, non-termination of* `hamming` *cannot be proven.*

To prove non-termination of `hamming`*, one needs to know that the list in the recursive call* `hamming(Res)` *is again of the form* `[N|Ns]`*. This can be done using the program specialization technique: more specific programs. This technique generates a more instantiated version of the program. For the running program, it generates the following recursive clause for hamming:*

```
hamming([N|Ns]):- times(s(s(0)),N,N2),
        times(s(s(s(0))),N,N3),
        times(s(s(s(s(s(0))))),N,N5),
        insert([N2,N3,N5],Ns,[R|Res]),
        hamming([R|Res]).
```

Because of this more instantiated clause, proving non-termination for hamming using NFG transitions to solve times/3 and insert/3 is straightforward. □

5.4 Summary

In this chapter, we identified classes of logic programs for which previous analyzers fail to prove non-termination and we extended our non-termination analysis to overcome these limitations. As in the previous chapter, non-termination is proven by constructing a symbolic derivation tree, representing all derivations for a class of queries, and then proving that a path in this tree can be repeated infinitely.

The most important class of programs for which previous analyzers fail, are programs for which no fixed sequence of clauses can be repeated infinitely. We have shown that non-failure information ([17]) can be used to abstract away from the exact sequence of clauses needed to solve non-failing goals. To use this non-failure information, type information is added to the symbolic derivation tree and a special NFG transition is introduced to solve non-failing atoms. As far as we know, this is the first time that non-failure information is used for non-termination analysis.

We have shown that program specialization ([26]) can be used to overcome another limitation of current analyzers. If non-termination cannot be proven due to aliased variables, redundant argument filtering may remove these duplicated variables from the program. Specialization can also be used in combination with the NFG transition. Program specialization may produce more instantiated clauses, giving a better approximation of the possible values after solving the non-failing goal.

Reference

The work in this chapter was presented at LOPSTR 2010 and is published in [55].

Chapter 6

Non-termination Analysis for Logic Programs with Integer Arithmetics

This chapter extends our non-termination condition for programs containing integer arithmetics. We modify the moded SLD-tree to handle a subset of the build-in predicates for integer arithmetics, commonly found in Prolog implementations. LP-check ensures finiteness of the tree and detects paths that may correspond to infinite loops. For every such path, two analyses are combined to identify classes of non-terminating queries. In the first phase, non-termination of the logic part of the program is proven by assuming that all comparisons between integer expressions succeed. We will show that only a minor adaption of our non-termination analysis is needed to achieve this. In the second phase, given the moded query, integer arguments are identified and constraints over these arguments are formulated, such that solutions for these constraints correspond to non-terminating queries. The chapter is structured as follows. Section 6.1 defines the subset of integer predicates that we consider in the chapter. Section 6.2 introduces the first phase of our analysis, proves non-termination for the logic part of the computation. Section 6.3 the second phase of our analysis, proves non-termination for the integer part of the computation. Section 6.4 describes our prototype analyzer and discusses our results. Finally, Section 6.5 summarizes the chapter.

6.1 Integer arithmetics

Prolog implementations contain special purpose predicates for handling integer arithmetics. Examples are $is/2, \geq /2, =:= /2, \ldots$.

Definition 6.1. *An expression Expr is an integer expression if it can be constructed by the following recursive definition.*

$$Expr = z \in \mathbb{Z} \mid -Expr \mid Expr + Expr \mid Expr - Expr \mid Expr * Expr \quad \square$$

An atom `"V is Expr"`, with V a free variable and $Expr$ an integer expression, is called an *integer constructor*. An atom $Expr1 \circ Expr2$ is called an *integer condition* if $Expr1$ and $Expr2$ are integer expressions and $\circ \in$ {`>`,`>=`,`=<`,`<`,`=:=`,`=/=`}.

6.2 Non-termination of the Logic part

This section introduces the first part of our non-termination analysis for programs containing integer arithmetics. In the first phase, we focus on the logic part of the computation and assume the integer conditions succeed. This section introduces a condition to detect that a program and moded query is either non-terminating or fails due to the evaluation of an integer condition. This condition is obtained by adapting our non-termination condition from Chapter 4. In the next section we will generated additional constraints on the class of queries to obtain classes of non-terminating queries.

6.2.1 Moded SLD-tree for programs with integer arithmetics

The first step of the extension is rather straightforward. The extensions to the moded SLD-tree of Chapter 3 are limited to the introduction of the label *integer variable* and additional transitions to handle integer constructors and integer conditions. Integer variables denote an unknown integer and will also be represented by underlining the name of the variable. An integer constructor, i.e., $is/2$, is applicable if the first argument is a free variable and the second argument is an integer expression. The application of an integer constructor labels the free variable as an integer variable. An integer condition, e.g., $\geq /2$, is applicable if both arguments are integer expressions. Since integer variables denote unknown integers, integer expressions are allowed to contain integer variables. Applications of integer constructors and integer conditions in the moded SLD-tree are denoted by derivation steps $N_i : G_i \Longrightarrow_{cons} N_{i+1} : G_{i+1}$ and $N_i : G_i \Longrightarrow_{cond} N_{i+1} : G_{i+1}$, respectively.

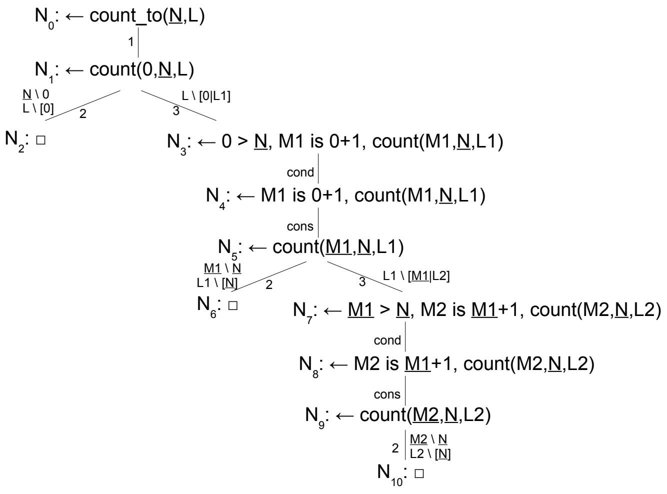


Figure 6.1: Moded SLD-tree *count_to*

Example 6.1. *The following program, count_to, is a faulty implementation of a predicate generating the list starting from 0 up to a given number. The considered class of queries is represented by the moded query* $\leftarrow count\_to(\underline{N}, L)$ *with* $\underline{N}$ *an integer variable.*

```
count_to(N,L):- count(0,N,L).
count(N,N,[N]).
count(M,N,[M|L]):- M > N, M1 is M+1, count(M1,N,L).
```

In the last clause, the integer condition should be `M < N` *instead of* `M > N`*. Due to this error, the program:*

- *fails for the queries for which* $\underline{N} > 0$ *holds,*
- *succeeds for* $\leftarrow count\_to(0, L)$*,*
- *loops for the queries for which* $\underline{N} < 0$ *holds.*

Figure 6.1 shows the moded SLD-tree for the considered query, constructed using LP-check. LP-check cuts clause 3 at node N_9*.* □

Note that by ignoring the possible values for the integer variables when constructing the tree, some derivations in it may not be applicable to any considered query. For example the refutations at nodes N_6 and N_{10} in the previous example cannot be reached by the considered queries.

Definition 6.2. *Let C be an integer condition or expression and N_i and N_j two nodes in a moded SLD-tree D. Let $Cons$ be the set of all integer constructors occurring as selected atom in a node N_p $(i \leq p \leq j)$ in D.*

The function $apply\_cons(C, N_i, N_j)$ returns the integer condition or expression obtained by exhaustively applying $\underline{I} \setminus Expr$ to C, for any $\underline{I}$ is $Expr \in Cons$. $\square$

Since we only apply integer constructors if the second argument is an integer expression, all integer conditions in a derivation can be expressed in terms of the integer variables of the query.

Lemma 6.1. *Let N_i be a node in a moded SLD-tree for a query $\leftarrow Q$ with integer variables Int. Let cond be an integer condition in the derivation to N_i, then $apply\_cons(cond, N_0, N_i)$ returns an integer condition over integer variables Int.*

Proof. This follows from the requirement of the derivation steps *cond* and *cons* for the arguments to be of the correct type. $\square$

First, we formalize the correspondence between moded SLD-derivations and concrete SLD-derivations.

Theorem 6.1. *Let N_i be a node in a moded SLD-tree for a query $\leftarrow Q$ with integers Int. Let $Cons$ be the set of all integer constructors in the derivation to N_i, let $Cond$ be the set of integer conditions on the derivation and let σ be the composition of all substitutions on input variables in the derivation to N_i.*

Then, the derivation to N_i is applicable to any query in $Den(\leftarrow Q\sigma)$ such that all conditions in $\{apply\_cons(cond, N_0, N_i) \mid cond \in Cond\}$ hold.

Proof. By induction on the number of derivation steps in a moded SLD-derivation for a query $\leftarrow Q$ with (non-integer) input variables $\underline{I_1}, \ldots, \underline{I_n}$ and integer variables $Int_1, \ldots, Int_m$.

Note that queries in $Den(\leftarrow Q)$ corresponds to a grounding substitution on the (non-integer) input variables of Q and a value for the integer variables in Q. Also note that because integer constructors are only applied with integer expressions as second argument, integer conditions *cond* in a derivation to N_i can always be expressed over the integers variables of the query using $apply\_cons(cond, N_0, N_i)$.

Base case: clause application. Let c be the clause applied to the query $\leftarrow Q$ in the considered derivation, let σ be the substitution corresponding to this clause application and let σ_I be the subset of bindings on input variables. Then, c is applicable to any element of $Den(\leftarrow Q)$ for which the ground terms (due to the grounding substitution) and the terms in σ_I are unifiable for all input variables. Thus, c is applicable to any query in $Den(\leftarrow Q\sigma_I)$.

Base case: Integer condition. If Q is an integer condition, then solving it results in a refutation and it's clear that $N_0 :\leftarrow Q \Rightarrow_{cond} N_1$ is applicable to any query in $Den(\leftarrow Q)$ for which the values corresponding to the integer variables of Q are such that the condition holds.

Induction step: clause application. Assume the theorem holds for $N_0 :\leftarrow G_0 \Rightarrow N_i :\leftarrow G_i$ and that we apply $N_i :\leftarrow G_i \Rightarrow_c N_{i+1} :\leftarrow G_{i+1}$. Let σ_I and be the substitutions on input variables due to this derivation step. Let θ be the composition of substitutions on input variables from N_0 to N_i and let Cs be the constraints due to integer conditions in the derivation up to this node.

As in the base case, this clause is applicable to G_i if the actual values of the input variables due to the grounding substitution unify with the terms in σ_I. Therefore, every goal in $Den(G_0\theta\sigma_I)$ can be evaluated to a goal in $Den(G_{i+1})$ if the values corresponding to the integer variables are such that Cs holds.

Induction step: Integer condition. Assume the theorem holds for $N_0 :\leftarrow G_0 \Rightarrow N_i :\leftarrow G_i$ and that we apply $N_i :\leftarrow G_i \Rightarrow_{cond} N_{i+1} :\leftarrow G_{i+1}$. Let θ be the composition of substitutions on input variables from N_0 to N_i and let Cs be the constraints due other integer conditions in the derivation.

The selected atom of G_i, A_i^1, only contains integer variables and $apply\_cons(A_i^1, N_0, N_i)$ results in a condition over the integer variables of the query and thus, this derivation is applicable to any query in $Den(\leftarrow Q\theta)$ such that $Cs \wedge apply\_cons(A_i^1, N_0, N_i)$ holds.

This concludes the proof. □

6.2.2 Adapting the non-termination condition

In this part of the analysis, we focus on the logical part of the computation and assume the integer conditions to succeed. We adapt our non-termination condition of Chapter 4 to prove that every instance of a moded query is either non-terminating or terminates due to the evaluation of an integer condition. However, while we assume the integer conditions succeed, we will adapt our condition to ensure that integer constructors and integer conditions are called with correct arguments. We need to guarantee that integer constructors are repeatedly evaluated with a free variable and an integer expression as arguments and that integer conditions are repeatedly evaluated with integer expressions as arguments.

To prove the repeated behavior on integer constructors and integer expressions, the *integer-similar to* relation is defined. Intuitively, given some loop in the computation, if an atom at the end of the loop is integer-similar to an atom at the start of the loop, then it will provide the required integer expressions to the first atom. First, we introduce positions to identify subterms and a function to obtain a subterm from a given position.

Definition 6.3. *Let L be a list of natural numbers, called a* position*, and A a moded atom or term. The function* subterm(L,A) *returns the subterm obtained by:*

- *if* $L = [I]$ *and* $A = f(A_1, \ldots, A_I, A_{I+1}, \ldots, A_n)$ *then* $subterm(L, A) = A_I$
- *else if* $L = [I|T]$ *and* $A = f(A_1, \ldots, A_I, A_{I+1}, \ldots, A_n)$ *then* $subterm(L, A) = subterm(T, A_I)$ □

A moded atom A is integer-similar to a moded atom B if it has integer expressions on all positions corresponding to integer expressions in B.

Definition 6.4. *Let A and B be moded atoms. A is integer-similar to B if for every integer expression t_B of B, with $subterm(L, B) = t_B$, there exists an integer expression t_A of A, with $subterm(L, A) = t_A$.* □

Example 6.2. *Let M be an integer variable in following examples.*

- $count(0, \underline{N}, L)$ *is integer-similar to* $count(\underline{M}, \underline{N}, L)$
- $count(\underline{M}, \underline{N}, L)$ *is integer-similar to* $count(0, \underline{N}, L)$
- $count(\underline{M} + 1, \underline{N}, L)$ *is integer-similar to* $count(\underline{M}, \underline{N}, L)$
- $count(\underline{M}, \underline{N}, L)$ *is not integer-similar to* $count(\underline{M} + 1, \underline{N}, L)$

Note that the last one is a counterexample because $count(\underline{M} + 1, \underline{N}, L)$ *has integer expressions on* $[1, 1]$ *and* $[1, 2]$*, while* $count(\underline{M}, \underline{N}, L)$ *does not have any subterms on these positions.* □

We adapt our non-termination condition of Chapter 4 by requiring that the selected atoms of N_b and N_e are integer-similar.

Definition 6.5. *Let N_b and N_e be nodes in a moded SLD-tree for a moded query Q, then $N_b \overset{mmg}{\rightarrow} N_e$ is called* a moded more general loop *if:*

- *A^1_b is an ancestor of A^1_e*
- *no substitutions on input variables occur from N_b to N_e*
- *A^1_e is moded more general than A^1_b*
- *A^1_e is integer-similar to A^1_b* □

Now we prove that a moded more general loop implies non-termination if there are values for the integers such that all integer conditions succeed. First, we introduce a lemma.

Lemma 6.2. *Let $N_b \overset{mmg}{\rightarrow} N_e$ be a moded more general loop and let D be the derivation to N_e followed by repeatedly applying the derivation steps from N_b to N_e, then all integer conditions in D are evaluated with two integer expressions as arguments and all integer constructors in D are evaluated with a free variable and an integer expression as arguments.* □

Proof. Since integer constructors and integer conditions are only applied if their arguments are of the correct type, the lemma holds for the derivation to N_e.

Due to the definition of the moded more general relation, a position corresponding to a free variable in A^1_b must correspond to a free variable in A^1_e. This also holds for all following iterations and thus integer constructors are always evaluated with a free variables as a first argument.

To prove that the second argument of an integer constructor and both arguments of integer conditions are integer expressions when they are evaluated, it is enough to prove that positions corresponding to integer expressions in A^1_b also correspond to integer expressions in the following iterations. This is guaranteed by Definition 6.4. □

Theorem 6.2. *Let $N_b \overset{mmg}{\rightarrow} N_e$ be a moded more general loop in a derivation for a moded query $\leftarrow Q$ and let θ be the composition of the substitutions on input variables in the derivation to N_e, then every query in $Den(\leftarrow Q\theta)$ is non-terminating or fails due to the evaluation of an integer condition.*

Proof. We need to prove that the sequence of derivation steps from N_b to N_e can be applied infinitely often to any goal in $Den(Q')$, assuming that all integer conditions in this derivation succeed. Although we can assume all integer conditions succeed, integer conditions and constructors must still be evaluated with terms of the correct type. Lemma 6.2 already proves this is the case.

Because A^1_b is an ancestor of A^1_e, only the selected atom of N_b influences if the sequence of derivation steps can be repeated infinitely often.

Because no substitutions on input variables occur in the path from N_b down to N_e, the corresponding sequence of derivation steps is applicable to any atom in $Den(A^1_b)$ (assuming that integer conditions are evaluated with values such that they succeed). Furthermore, it is also applicable to queries that are more general, but integer-similar to queries in $Den(\leftarrow A^1_b)$. Let this set of queries be S.

Since $A^1_j \rhd A^1_i$, every query in $Den(\leftarrow A^1_e)$ is also a member of S. This concludes the proof. $\square$

Example 6.3. *The path between nodes N_5 and N_9 in Figure 6.1 satisfies the conditions of Theorem 6.5. There are no substitutions on input variables from N_5 to N_9 and thus, every query in $Den(\leftarrow count\_to(\underline{N}, L))$ is either non-terminating or fails due to the evaluation of an integer condition. Note that although $\leftarrow count\_to(0, L)$ has a succeeding derivation to N_2, its derivation to N_9 fails due to the integer condition $0 > \underline{N}$.* $\square$

To verify the moded more general relation and the integer-similar to relation of Definition 6.5, we strengthen Proposition 1 of Chapter 4 to imply both the moded more general relation and the integer-similar to relation.

Proposition 6.1. *Let A and B be moded atoms. Let A_1 and B_1 be renamings of these atoms such that they do not share variables. A is moded more general than B and A is integer-similar to B, if A_1 and B_1 are unifiable with most general unifier $\{V_1 \setminus t_1, \ldots, V_n \setminus t_n\}$, such that for each binding $V_i \setminus t_i$, $1 \leq i \leq n$, either:*

- *$V_i \in Var(B_1)$ and V_i is labeled as integer and t_i is an integer expression, or*
- *$V_i \in Var(B_1)$ and V_i is labeled as input but not as integer variable, or*
- *$V_i \in Var(A_1)$, V_i is not labeled as input, no variable of $Var(t_i)$ is labeled as input and t_i does not contain integers.* $\square$

Proof. To prove that A is integer-similar to B, we must show that every position corresponding to an integer expression in B corresponds to an integer expression in A. This follows from the restriction that normal variables are not allowed to be substituted by terms containing integer variables or integers (third item). To prove that the proposition implies that A is moded more general than B we refer to the proof of Proposition 1 in Chapter 4. □

Example 6.4. *Since the selected atoms of nodes N_5 and N_9 in Figure 6.1 are variants, Proposition 6.1 holds.* □

6.3 Non-termination of the Integer part

This section introduces the second part of our analysis. If Theorem 6.2 holds, we generate extra constraints on the integer variables proving non-termination. These constraints consist of reachability constraints, identifying queries for which the derivation up till the last node is applicable, and an implication proving that the integer conditions will also succeed in the following iterations. First we define how to obtain these constraints and then we apply transformations to solve these constraints automatically.

Example 6.5. *As a first example, we introduce the constraints for the path between N_5 and N_9 in the moded SLD-tree of count_to in Figure 6.1. For this path, Theorem 6.5 holds and thus every query denoted by $\leftarrow count\_to(\underline{N}, L)$ is either non-terminating or terminates due to an integer condition.*

To restrict the class of considered queries to those for which the derivation to N_9 is applicable, all integer conditions in the derivation are expressed in terms of the integers of the query, yielding $0 > \underline{N}$ and $0 + 1 > \underline{N}$.

For this program and considered class of queries, the condition $0 > \underline{N}$ implies that the derivation is applicable until node N_9. The following implication states that if the condition of node N_7 holds for any two values M and N, then it also holds for the values of the next iteration.

$$\forall M, N \in \mathbb{Z} : M > N \Longrightarrow M + 1 > N$$

This implication is correct and thus proves non-termination for the considered queries if the precondition holds in the first iteration. This is the case for all queries in $Den(\leftarrow count\_to(\underline{N}, L))$ with $0 > \underline{N}$ since the value corresponding to M in the first iteration is 0 and the value corresponding to N is $\underline{N}$. This proves non-termination of all considered queries for which $0 > \underline{N}$. □

In the following example, applicability of the derivation does not imply non-termination. To detect a class of non-terminating queries, a domain constraint is added to the pre- and postcondition of the implication.

Example 6.6.

```
constants(I,J):- I =:= 2, In is J*2, Jn is I-J, constants(In,Jn).
```

The clause in constants is applicable to any goal with $constants(2, \underline{J})$ *as selected atom, with* $\underline{J}$ *an integer variable. Since the first argument in the next iteration is the value corresponding to* $\underline{J} * 2$*, only goals with the selected atom* $constants(2, 1)$ *are non-terminating for this program.*

Since applicability of the derivation does not imply non-termination, a similar implication as in the previous example is false, $\forall I, J \in \mathbb{Z} : I = 2 \implies J * 2 = 2$*. To overcome this, a constraint is added to the pre- and post-condition of this implication, restricting the considered values of* $\underline{J}$ *to an unknown set of integers, called its domain.*

$$\exists Dom_j \subset \mathbb{Z}, \forall I, J \in \mathbb{Z} : I = 2, J \in Dom_j \implies J * 2 = 2, I - J \in Dom_j$$

The resulting implication is true for $Dom_j = \{1\}$*. By requiring that the considered moded query satisfies both the reachability constraint and the additional constraint in the pre-condition, the non-terminating query* $\leftarrow constants(2, 1)$ *is obtained.* □

All information needed to construct these constraints can be obtained from the moded SLD-tree.

The constraints guaranteeing a derivation to N_j to be applicable, can be obtained using $apply\_cons(Cond, N_0, N_i)$ for any integer condition $Cond$ in a node N_i in the considered derivation. For a path from N_b to N_e, the precondition of the implication is obtained using $apply\_cons(Cond, N_b, N_i)$, for each condition $Cond$ in a node N_i between nodes N_b to N_e and universally quantifying the integer variables of N_b.

Example 6.7. *The derivation to* N_9 *in Figure 6.1, contains integer conditions in nodes* N_3 *and* N_7*. These are expressed on the integer variable of the query,* $\underline{N}$*, using apply_cons.*

- $apply\_cons(0 > \underline{N}, N_0, N_3) = 0 > \underline{N}$
- $apply\_cons(\underline{M1} > \underline{N}, N_0, N_7) = 0 + 1 > \underline{N}$

To obtain the precondition of the implication, the integer condition in N_7 *is expressed in terms of the integer variables of* N_5*.*

- $apply\_cons(\underline{M1} > \underline{N}, N_5, N_7) = \underline{M1} > \underline{N}$

Universally quantifying these variables yields the precondition. □

To obtain the consequence of the implication for a path from N_b to N_e, one first replaces the integer variables of N_b in the precondition by the corresponding integer variables of N_e. Then, *apply_cons* is used to express the consequence in terms of the values in the previous iteration.

Definition 6.6. *Let LHS be the precondition of an implication, consisting of integer conditions and constraints of the form $I \in Dom_I$. Let N_i and N_j be two nodes in a moded SLD-derivation such that all integer variables in LHS are in A_i^1 and let $\underline{I_1}, \ldots, \underline{I_n}$ be all integer variables of A_i^1.*

If there exist subterms of A_j^1, $t_1, \ldots, t_n$, such that $\forall L : subterm(L, A_i^1) = \underline{I_p} \Longrightarrow subterm(L, A_j^1) = t_p, 1 \leq p \leq n$, then $replace(LHS, N_i, N_j)$ is obtained by applying $\{\underline{I_1} \setminus t_1, \ldots, \underline{I_n} \setminus t_n\}$ to all constraints in LHS. □

Example 6.8. *In Example 6.7, we generated the precondition of the implication, $\underline{M1} > \underline{N}$. To obtain the consequence, $replace(\underline{M1} > \underline{N}, N_5, N_9)$ is applied, yielding $\underline{M2} > \underline{N}$. Then, the integer variable of N_9, $\underline{M_2}$, is expressed in terms of the integer variables of N_5 using $apply\_cons(\underline{M2} > \underline{N}, N_5, N_9) = \underline{M_1} + 1 > \underline{N}$.*

Adding the domains to the pre- and postcondition yields the desired implication:
$\exists Dom_N, Dom_{M1} \subset \mathbb{Z}, \forall N, M1 \in \mathbb{Z} : M1 > N,\ N \in Dom_N,\ M1 \in Dom_{M1} \Longrightarrow$
$M1 + 1 > N,\ N \in Dom_N,\ M1 + 1 \in Dom_M$ □

Adding these constraints to the class of queries detected by Theorem 6.5, yields a class of non-terminating queries.

6.3.1 Proving that the constraints on integers are solvable

The previous subsection introduced constraints, implying that all integer conditions in a considered derivation succeed. In this subsection, we introduce a technique to check if these constraints have solutions, using a constraint-based approach. Symbolic coefficients represent values for the integers in the query and domains in the implication, for which the considered path is a loop. After these coefficients are introduced, the implication is transformed into a set of equivalent implications over natural numbers. These implications can then be solved automatically in the constraint-based approach, based on Proposition 3 of [35].

Proposition 6.2 (Proposition 3 of [35]). *Let prem be a polynomial over n variables and conc a polynomial over 1 variable, both with natural coefficients,* *where conc is not a constant. Moreover, let* $p_1, \ldots, p_{n+1}, q_1, \ldots, q_{n+1}$ *be arbitrary polynomials with integer coefficients*[1] *over the variables* $\overline{X}$*. If*

$$\forall \overline{X} \in \mathbb{N} : conc(p_{n+1}) - conc(q_{n+1}) - prem(p_1, \ldots, p_n) + prem(q_1, \ldots, q_n) \geq 0$$

is valid, then $\forall \overline{X} \in \mathbb{N} : p_1 \geq q_1, \ldots, p_n \geq q_n \Longrightarrow p_{n+1} \geq q_{n+1}$ *is also valid.* $\square$

Introducing the symbolic coefficients.

To represent half-open domains in the implication by symbolic coefficients, the domains are described by two symbolic coefficients, one upper or lower limit and one for the direction. Constraints of the form $Exp \in Dom_I$ in the implication, are replaced by constraints of the form $d_I * Exp \geq d_I * c_I$ with d_I either 1 or -1, describing the domain $\{c_I, c_I - 1, \ldots\}$ for $d_I = -1$ and $\{c_I, c_I + 1, \ldots\}$ for $d = 1$. The values to be inferred for the integers of the query should satisfy the precondition of the implication. Of course, the symbolic coefficients c_I should also be consistent with the values of the integers in the query.

Example 6.9. *In Example 6.5, we introduced constraints on the integer variable* $\underline{N}$*,* $0 > \underline{N}$ *and* $0 + 1 > \underline{N}$*, proving non-termination for queries in* $Den(\leftarrow count\_to(\underline{N}, L))$*. By convention, we denote the symbolic coefficients as constants. For the integer variable* $\underline{N}$*, we introduce the symbolic coefficient* n*.*

The implication introduced in Example 6.5, for the path from N_5 *to* N_9 *in Figure 6.1, does not contain constraints on the domains. When adding these constraints to the pre- and postcondition, we obtain the following implication.*

$$\begin{aligned}&\forall M, N \in \mathbb{Z} :\ M > N,\ N \in Dom_N,\ M \in Dom_M \Longrightarrow \\ &M + 1 > N,\ N \in Dom_N,\ M + 1 \in Dom_M\end{aligned}$$

Representing these domains by symbolic coefficients yields the following implication.

$$\begin{aligned}&\forall M, N \in \mathbb{Z} :\ M > N,\ d_N * N \geq d_N * c_N,\ d_M * M \geq d_M * c_M \Longrightarrow \\ &M + 1 > N,\ d_N * N \geq d_N * c_N,\ d_M * (M + 1) \geq d_M * c_M\end{aligned}$$

To guarantee that the precondition succeeds for the considered derivation, c_M *and* c_N *are required to be the values for* $\underline{M}$ *and* $\underline{N}$ *in node* N_5*. Combining these constraints implies non-termination for the query* $\leftarrow count\_to(n, L)$*, for which the following constraints are satisfied with some unknown integers* c_N, c_M, d_N *and* d_M*.*

[1]Proposition 3 in [35] states natural coefficients, but the proposition also holds for polynomials with integer coefficients.

(1) $0 > n,\ 0+1 > n$ *to guarantee applicability of the derivation*

(2) $c_N = n,\ c_M = 0+1$ *to guarantee that the precondition holds*

(3) $d_N = 1 \vee d_N = -1,\ d_M = 1 \vee d_M = -1,$

(4) $\forall M, N \in \mathbb{Z} : M > N, d_N * N \geq d_N * c_N, d_M * M \geq d_M * c_M \Longrightarrow$
$M+1 > N, d_N * N \geq d_N * c_N, d_M * (M+1) \geq d_M * c_M$ *to prove that the condition succeeds infinitely often.*

Due to the implication, d_M *has to be* 1*.* d_N *can be either* 1 *or* -1*.* □

To be able to infer singleton domains, we allow the constant describing the direction of the interval to be 0. If in such a constant, d_I, is zero, the constraints on the domain are satisfied trivially because they simplify to $0 \geq 0$. To guarantee that the domain is indeed a singleton when d_I is inferred to be zero, a constraint of the form $(1-d_I^2)Exp = (1-d_I^2) * c_I$ is added to the postcondition for every constraint $d_I * I \geq d_I * c_I$. This constraint is trivially satisfied for half-open domains and proves that $\{c_I\}$ is the domain in the case that $d_I = 0$.

Example 6.10. *In Example 6.6, we introduced constraints on the integer variables* $\underline{I}$ *and* $\underline{J}$*, proving non-termination for queries in* $Den(\leftarrow constants(\underline{I}, \underline{J}))$*. Introducing symbolic coefficient i and j for the integers of the query and for the domains of* $\underline{I}$ *and* $\underline{J}$*, yields the following constraints.*

(1) $i = 2$ *to guarantee applicability of the derivation*

(2) $c_I = i,\ c_J = j$ *to guarantee that the precondition holds*

(3) $d_I \leq 1,\ d_I \geq -1,\ d_J \leq 1,\ d_J \geq -1,$

(4) $\forall I, J \in \mathbb{Z} : I = 2,\ d_I * I \geq d_I * c_I,\ d_J * J \geq d_J * c_J \Longrightarrow$
$J * 2 = 2,\ d_I * (J * 2) \geq d_I * c_I, (1-d_I^2) * (J*2) = (1-d_I^2) * c_I,$
$d_J * (I - J) \geq d_J * c_J, (1-d_J^2) * (I-J) = (1-d_J^2) * c_J$

The implication in (4) *can only be satisfied with* d_J *equal to zero.* □

To implications over the natural numbers

The symbolic coefficients to be inferred which represent the domains, allow us to transform the implication over $\mathbb{Z}$ to an equivalent implication over $\mathbb{N}$.

- for $d_I = 1$, any integer in $\{c_I,\ c_I + 1,\ \ldots\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$

- for $d_I = -1$, any integer in $\{c_I,\ c_I - 1,\ \ldots\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$
- for $d_I = 0$, any integer in $\{c_I\}$ that satisfies the precondition is in $\{c_I + d_I * N \mid N \in \mathbb{N}\}$

Therefore, we obtain an equivalent implication over the natural numbers by replacing each integer I by its corresponding expression $c_I + d_I * N$ and replacing the universal quantifier over I by a quantifier over N.

Automation by a translation to Diophantine constraints

To solve the resulting constraints, we use the approach of [35]. Constraints of the form $A =:= B$ in the implication, are replaced by the conjunction $A \geq B,\ B \geq A$. Constraints of the form $A = / = B$, yield two disjunctive cases. One obtained by replacing the $= / =$ in the pre- and postcondition by $>$ and one obtained by replacing it by $<$. The other conditions – i.e. $>, <$ and $\leq$ – are transformed into $\geq$-constraints in the obvious way. Implications with only one consequence are obtained by creating one implication for each consequence, with the pre-condition of the original implication.

The resulting implications allow to apply Proposition 6.2. These inequalities of the form, $p \geq 0$, are then transformed into a set of *Diophantine constraints*, i.e. constraints without universally quantified variables, by requiring that all coefficients of p are non-negative. As proposed in [35], the resulting Diophantine constraints are then transformed into a SAT-problem. The constraints are then proven to have solutions by a SAT solver by inferring one possible solution.

6.4 Evaluation

We implemented the analysis introduced in the chapter, in our previous non-termination analyzer *pTNT*, which is written in SWI-Prolog [59] and can be downloaded from http://www.cs.kuleuven.be/~dean/iclp2011.html. We tested our analysis on a benchmark of 14 programs similar to those used in this chapter. These programs are also available online. To solve the resulting SAT-Problem, MiniSat [19] is used.

We experimented with different bit-sizes in the translation to SAT and different classes of functions for the *prem* functions in Proposition 6.2. As *conc* functions, the identity function was used. Table 6.1 shows the results for the considered settings, $+$ denotes that non-termination is proven successfully, $-$ denotes that

	linear-class, 3 bits	linear-class, 4 bits	max2-class, 3 bits	max2-class, 4 bits
count_to	+	+	+	+
constants	+	+	+	*OS*
int1	+	+	+	+
int2	+	+	+	+
int3	+	+	+	*OS*
int4	+	+	+	*OS*
int5	+	+	+	*OS*
int6	+	+	+	*OS*
int7	+	+	*OS*	*OS*
int8	+	+	*OS*	*OS*
int9	−	+	*OS*	*OS*
int10	−	−	+	*OS*
int11	−	+	−	*OS*
int12	−	+	−	*OS*
int13	+	+	+	+
int14	+	+	+	*OS*

Table 6.1: An overview of the experiments

non-termination could not be proven and OS denotes that the computation went out of stack. The considered settings are 3 and 4 as bit-sizes and *linear* and $max2$ as forms for the symbolic *prem*-functions. The *linear* class is a weighted sum of each argument. The $max2$ class contains a weighted term for each multiplication of two arguments. The analysis time is between 1 and 20 seconds for all programs and settings.

Table 6.1 shows non-termination can be proven for any program of the benchmark when choosing the right combination of parameters, but no setting succeeds in proving non-termination for all programs. Programs $int9$ and $int12$ require a constant that cannot be represented with bit-size 3. Linear prem-functions cannot prove non-termination for $int10$. However, the setting with 4 as a bit-size and $max2$ as class of *prem*-function usually fails, because these settings cause an exponential increase in memory use during the translation to SAT.

From the experiments, we have learned that it could be useful to apply SMT solvers, instead of SAT solvers, to reduce the memory usage.

6.5 Summary

In this chapter, we introduced a technique to detect classes of non-terminating queries for logic programs with integer arithmetic. The analysis starts with a given program and class queries, specified using modes, and detects subclasses of non-terminating queries. First, the derivations for the given class of queries are abstracted by building a moded SLD-tree with additional transitions to handle integer arithmetic. Then, this moded SLD-tree is used to detect subclasses of non-terminating queries in two phases. In the first phase, we ignore the conditions over integers, e.g., $> /2$, and detect paths in the moded SLD-tree that correspond to infinite derivations if all conditions on integers in those derivations succeed. For every such path, the corresponding subclass of queries is generated. In the second phase, the obtained classes of queries are restricted to classes of non-terminating queries, by formulating constraints implying that all conditions on integers will succeed. These constraints are then solved by transforming them into a SAT problem.

We implemented this approach in our non-termination analyzer $pTNT$ and evaluated it on small benchmark of non-terminating Prolog programs with integer arithmetic. The evaluation shows that the proposed technique is rather powerful, but also that the parameters in the transformation to SAT must be chosen carefully to avoid excessive memory use.

Reference

The work in this chapter was presented at ICLP 2011 and is published in [56].

Chapter 7

Termination Analysis of CHR

This chapter presents an approach to termination analysis of Constraint Handling Rules (CHR). The presented analysis is the first termination condition for CHR without restrictions on the kind of rules in the CHR program. We propose a termination condition that verifies conditions imposed on the dynamic process of adding constraints to the store, instead of a termination argument based on the comparison of sizes of consecutive computation states. We demonstrate the condition's applicability on a set of terminating CHR programs, using a prototype analyzer. This analyzer is the first in-language automated termination analyzer for CHR programs.

7.1 Introduction

Constraint Handling Rules (CHR), created by Thom Frühwirth [20], is a relatively young member of the declarative programming languages family. It is a concurrent, committed-choice, logic programming language. CHR is constraint-based and has guarded rules that rewrite multisets of atomic formulas until they are solved. CHR defines three kinds of rules: *simplification* rules remove constraints and add new ones, *propagation* rules only add new constraints and *simpagation* rules are a combination of both. CHR's simple syntax and semantics make it well-suited for implementing custom constraint solvers [20, 47, 48]. Particularly the latter feature of the language accounts for its success and impact on the research community.

Due to its multi-headed rules, CHR provides extra declarative expressivity compared to the single-headed rules of Logic Programming (LP). But at the same time, the operational behavior of multi-headed rules is more complex and harder to predict than that of single-headed ones. As a consequence, the importance of automated analysis tools, that assist a CHR-programmer to predict runtime properties of his/her programs is high. One of these properties is the termination of such programs.

Although the language is strongly related to LP, termination analysis of CHR programs has received little attention. Termination of CHR without propagation has been addressed before in [21] and [42]. The approach of [21] is based on adapting well-known LP termination techniques directly to CHR. The approach of [42] is based on a transformation of CHR into Prolog.

Because propagation rules only add constraints, a fire-once policy is used for these rules. A condition based on this fire-once policy is needed to prove termination for such programs. In this chapter we present a condition based on an interpretation of the constraints, such that propagation rules only add 'smaller' constraints and for other rules, the number of the 'largest' constraints decreases. Because of the fire-once policy, this condition implies termination. We implemented the method in a prototype analyzer and performed an experimental evaluation. The results were very satisfactory. Since the approach presented in [21] was not implemented, this analyzer is the first in-language automated termination analyzer for CHR.

The chapter is organized as follows. In the next section, we introduce the basic aspects of CHR and adapt some concepts from termination analysis of LP to the CHR context. In Section 7.3, we introduce a termination condition for general CHR programs that is sufficient for proving termination. Next, we further refine the condition so that it can be automated. In Section 7.4, we discuss our implementation and experimental evaluation. Finally, Section 7.5 summarizes the chapter.

7.2 Preliminaries

7.2.1 Constraint Handling Rules

Syntax. A *constraint* in CHR is a first-order predicate. We distinguish between *built-in constraints*, predefined and solved by the underlying constraint theory, CT, and *CHR constraints*, user-defined and solved by a *CHR program P*. A CHR program is a finite set of *CHR rules*. Rules are of the form:

Simplification rule:	**Propagation rule:**	**Simpagation rule:**
$true \setminus H \Leftrightarrow [G \,\|]\, B.$	$H \setminus true \Leftrightarrow [G \,\|]\, B.$	$H_1 \setminus H_2 \Leftrightarrow [G \,\|]\, B.$
or $H \Leftrightarrow [G \,\|]\, B.$	**or** $H \Rightarrow [G \,\|]\, B.$	

The head H or $H_1 \setminus H_2$ is a conjunction of CHR constraints. The optional guard G is a conjunction of built-in constraints. The body B is a conjunction of built-in and CHR constraints. Empty conjuncts are denoted by the built-in constraint *true*. As in Prolog syntax, conjuncts are separated by commas. Note that all rules can be written as simpagation rules.

Example 7.1 (Fibonacci)**.** *The CHR program below implements a Fibonacci algorithm. Natural numbers are written in the symbolic notation, using 0 and the successor functor s. In this program, add/3 is a built-in constraint which defines addition on natural numbers written in symbolic notation.*

$$fib(N, M1), fib(N, M2) \Leftrightarrow M1 = M2, fib(N, M1).$$

$$fib(0, M) \Rightarrow M = s(0).$$

$$fib(s(0), M) \Rightarrow M = s(0).$$

$$fib(s(s(N)), M) \Rightarrow fib(s(N), M1), fib(N, M2), add(M1, M2, M).$$

The first rule is a simplification rule that removes doubles. The other rules are propagation rules. Base cases are solved by the second and third rule. The last rule adds CHR constraints representing Fibonacci numbers and a built-in constraint relating their arguments. □

Operational Semantics. A *CHR program* defines a state transition system, where a *state* is defined as a conjunction of CHR and built-in constraints, called the *constraint store*. The *initial state* or *query* is an arbitrary conjunction of constraints. Each state where either the built-in constraints are inconsistent (*failed state*), or no more transitions are possible, is called a *final state*.

Definition 7.1 (Transition relation)**.** *The* ***transition relation****, $\longmapsto$, between states, given a constraint theory CT and a CHR program P, is defined as:*

$$\begin{array}{l} H_1' \wedge H_2' \wedge D \longmapsto (B \wedge H_1' \wedge D)\theta\theta' \\ if\ H_1 \setminus H_2 \Leftrightarrow G \mid B.\ in\ P\ and\ \theta, \theta'\ are\ substitutions\ such\ that \\ CT \models D \rightarrow (((H_1\theta \wedge H_2\theta) = (H_1' \wedge H_2')) \wedge G\theta\theta') \end{array}$$

A CHR rule, $H_1 \setminus H_2 \Leftrightarrow G \mid B$, is applicable to a conjunction of CHR constraints, $H' = H_1' \wedge H_2'$, if H_1' matches head H_1 and H_2' matches head H_2 with matching *substitution θ, such that the guard G evaluates to true, with answer substitution θ', given the built-ins in the constraint store. The body B is a conjunction of built-in and CHR constraints. $B\theta\theta'$ is added to the constraint store. Rule application is non-deterministic and committed-choice.* □

For more information about the operational semantics of CHR, we refer to [20].

A CHR program P with query I *terminates*, if all computations for P with query I end in a final state. Because propagation rules do not remove constraints from the store, a fire-once policy is used to prevent trivial non-termination. This fire-once policy ensures that no propagation rule fires more than once on the same multi-set of constraints. The *propagation history* implements this policy. For a more detailed description of the propagation history, we refer to [3].

CHR has no fairness guarantees: constraints in the body of a rule might never be selected in a computation. This means that our technique will not be able to prove termination if termination depends on the evaluation of built-in constraints from the body.

In the following example, we discuss a computation for the Fibonacci program discussed earlier.

Example 7.2 (Fibonacci continued)**.** *With a typical query $fib(s(s(0))), N)$, the last propagation rule adds two new $fib/2$ constraints to the store with lower first arguments. However, it does not remove the constraint that has fired the rule from the store. This propagation rule fires again on the added constraint $fib(s(s(0)), N)$, adding again two CHR constraints with a lower first argument to the store. The other two propagation rules resolve base cases, while the simplification rule removes duplicates. The constraint store:*
$fib(s(s(s(0))), s(s(s(0)))) \wedge fib(s(s(0)), s(s(0))) \wedge fib(s(0), s(0)) \wedge fib(0, s(0))$
is the final state. Without the simplification rule the answer would contain an additional $fib(s(0), s(0))$ constraint. □

Observe that, given any reasonable way to measure the size of the constraint store, the size of the store increases for this program. This means that standard approaches for proving termination of LP, which prove decreases in size between consecutive computation states, are not easily and immediately applicable to CHR. One would need to explicitly add an encoding of the propagation history to the representation of the store to make this work. In this paper, we have chosen for a different type of termination condition for programs with propagation rules.

7.2.2 Termination Analysis

Termination analysis of logic programs has received a lot of attention (see for example [16]). In LP, consecutive computation states are compared using a *norm* and *level mapping*. A norm is a function which maps terms to natural numbers. A level mapping is a function which maps atoms to natural numbers. In this subsection we redefine some well-known concepts from LP to the CHR context.

We define $Term_P$ as the set of all terms constructible from a program P and Con_P as the set of all constraints constructible from P, with arguments from $Term_P$.

Definition 7.2 (norm, level mapping)**.** *A* ***norm*** *is a mapping* $||.|| : Term_P \to \mathbb{N}$. *A* ***level mapping*** *is a mapping* $|.| : Con_P \to \mathbb{N}$. □

We refer to $|C|$ as the *level value* of C. Several examples of norms and level mappings can be found in the literature on LP termination analysis [16]. Two well-known norms are *list-length* and *term-size.*

Definition 7.3 (list-length, term-size)**.**

List-length *is defined as:*
$|| [t_1|t_2] ||_l = 1 + ||t_2||_l$
$||t||_l = 0$ *otherwise.*
with t_1, t_2 *and* t *any term.*

Term-size *is defined as:*
$||f(t_1, t_2, ..., t_n)||_t = 1 + \sum_{1 \leq i \leq n} ||t_i||_t$
$||t||_t = 0$ *otherwise*
with $t_1, \ldots, t_n$ *and* t *any term*

□

The most common kind of level mapping is the *linear level mapping.*

Definition 7.4 (linear level mapping)**.** *A* ***linear level mapping*** *is any level mapping which can be defined as:* $|con(t_1, ..., t_n)| = con_0 + \sum_{1 \leq i \leq n} con_i.||t_i||$, *with* $con_i \in \mathbb{N}$ *only depending on the constraint symbol con and* $||.||$ *a norm.* □

We also adapt the notion of rigidity to the CHR context.

Definition 7.5 (Rigidity)**.** *A CHR constraint C is* ***rigid*** *w.r.t. a level mapping* $|.|$ *iff* $\forall$ *substitutions* $\theta : |C| = |C\theta|$. □

To prove termination, we need some information about the CHR constraints which can be added to the store during an execution of a program for a query. For this purpose, we define the *call set.*

Definition 7.6 (Call Set)**.** *Given a program P and a query I, the* ***call set*** *for P with query I,* $\mathbf{Call(P, I)}$*, is the union, over all possible computations of P* *for I, of all CHR-constraints which are added to the constraint store during that computation.*

Usually, $Call(P, I)$ is specified using an abstraction. In what follows we describe the call set using types.

7.3 A new termination condition for CHR

In [21], concepts and ideas from LP termination analysis are adapted to CHR with simplification only. Termination is proved by showing a decrease between the removed and the added constraints, for each CHR rule in the program.

The extension to programs with propagation rules gives a totally new termination problem. In LP or CHR without propagation, each rule removes one predicate, for LP, and at least one constraint, for CHR without propagation. Termination is proven by measuring a decrease of the goal, or of the constraint store, for each rule application. For CHR with propagation, this approach seems infeasible because new constraints are added and no existing constraints are removed. As mentioned before, one would need to keep track of information regarding the propagation history to observe a decrease. Instead of a termination argument based on a comparison of sizes of consecutive computation states, we formulate and verify conditions imposed on the dynamic process of adding constraints to the store. We formulate conditions which guarantee that the entire computation only adds a finite number of constraints to the store. Due to the use of a propagation history, this implies termination.

First we prove that if only a finite number of CHR constraints are added to the constraint store, program P with query I terminates. Note that, if the same constraint is added multiple times to the constraint store, then we consider these additions as different.

Lemma 7.1 (Termination of a CHR program)**.** *A CHR program P with query I terminates iff there are a finite number of additions of CHR constraints to the constraint store during any execution of P for I.* $\square$

Proof. $\Longrightarrow$: If P terminates for I, then for any execution of P for I, only a finite number of rules are applied. Therefore, only a finite number of CHR constraints are added to the store.
$\Longleftarrow$: Suppose there are only a finite number of CHR constraints added to the store during any execution of program P for I. Each propagation rule can only be fired a finite number of times because of the propagation history. Each simplification or simpagation rule removes at least one CHR constraint from

the store. Therefore, a simpagation or simplification rule can only be applied a finite number of times. Since every rule can only be applied a finite number of times, P terminates for I. □

7.3.1 Ranking Condition (RC) for CHR with substitutions

The next definition gives the first version of our Ranking Condition (RC). It is applicable to general CHR programs. Informally, a program satisfies the RC if each propagation rule only adds constraints which are smaller than the constraints in the head and each simplification or simpagation rule reduces the number of largest constraints in the rule.

Definition 7.7 (Ranking condition for CHR with substitutions)**.** *A program P and a query I satisfy the RC for CHR, w.r.t. level mapping $|.|$ iff every CHR constraint in $Call(P, I)$ is rigid w.r.t. $|.|$ and for each rule in P and for every matching substitution θ and answer substitution θ' from Definition 7.1:*

1. *For a simplification or simpagation rule $H \backslash H_1, ..., H_n \Leftrightarrow G \mid B_1, ..., B_m$, with $n > 0$ and body-CHR constraints $B_k, ..., B_m$,*
   let $p = max\,\{|H_1\theta|, ..., |H_n\theta|, |B_k\theta\theta'|, ..., |B_m\theta\theta'|\}$. Then, the number of CHR constraints with level value p is higher in $\{H_1\theta, ..., H_n\theta\}$ than in $\{B_k\theta\theta', ..., B_m\theta\theta'\}$
2. *For a propagation rule: $H_1, ..., H_n \Rightarrow G \mid B_1, ..., B_m$, with body-CHR constraints $B_k, ..., B_m$:*
   for all $i = 1, ..., n$ and $j = k, ..., m$: $|H_i\theta| > |B_j\theta\theta'|$. □

Our condition on simplification and simpagation rules in Definition 7.7 is more strict than the corresponding ranking condition for such rules in [21]. The reason for this is that in the presence of propagation rules, a multiset order decrease, as in [21], is insufficient to guarantee termination.

Example 7.3 (Counterexample multiset order decrease)**.** *Consider the program:*

$$a(s(N)), a(N) \Leftrightarrow a(s(N)). \qquad\qquad a(s(N)) \Rightarrow a(N).$$

The program does not terminate for any query $a(s(n))$, with n any term. Our RC cannot be fulfilled for the simplification rule. However, using a level mapping $|a(t)| = ||t||_t$ and a multiset order on conjunctions of constraints, as in [21], there is a decrease from head to body constraints for this rule. So a straightforward extension of the ranking condition of [21] is incorrect. □

Theorem 7.1 (Sufficiency of the RC with substitutions)**.** *Let program P with query I satisfy the RC with substitutions w.r.t. $|.|$, then all computations for P with query I terminate.* □

Proof. In order to prove termination of a CHR program P with a query I, Lemma 7.1 shows that it is sufficient to prove that the total number of CHR constraints, added during an execution of P for I, is finite. As a base of the induction, we show that there is a maximal level value for the CHR constraints in the store and that only a finite number of CHR constraints with that level mapping can be added to the store. Using induction we prove that for each level value only a finite number of CHR constraints can enter the store.

Before we present the induction proof, first note that because of the rigidity of $Call(P, I)$ under $|.|$, the level value of a constraint that matches the head of a rule cannot change anymore due to instantiations caused by answer substitutions of guards or built-in body constraints. So, level values of constraints in the store are static: the level values cannot change over time.

Base case. Let max be the maximal level value of the constraints in I. Propagation rules only add constraints with a lower level value than max. Because of the RC, simplification or simpagation rules only add constraints with a level value smaller than or equal to the largest level value of the constraints matching the head of the rule. Every time a CHR constraint of level value max is added by such a rule, the number of CHR constraints with level value max decreases. So only a finite number a_{max} of CHR constraints with level value max are added to the store during any execution of P for I.

Induction step. Let $a_{max}, ..., a_{n+1}$ be upper limits for the number of CHR constraints with level value $max, ..., n + 1$, w.r.t. $|.|$, which can be added to the constraint store during an execution of P with query I.

- I only contains a finite number of constraints with level value n.
- If an instance of a propagation rule adds a CHR constraint with level value n to the store, the CHR constraints matching the head all have a level value larger than n. Because of the upper limits $a_{max}, ..., a_{n+1}$ and the propagation history, every propagation rule can only add a finite number of CHR constraints with level value n.
- For every instance of a simplification or simpagation rule which adds a CHR constraint with level value n to the store, there exists an i, $i : n \leq i \leq max$, such that the number of CHR constraints of level value i decreases, and no constraint with a level value higher than i is added to the store by this simplification or simpagation rule. This implies that only a finite number of constraints with level mapping n can enter the store by

simpagation or simplification rules, because after enough rule executions, there are no CHR constraints with a level value n or higher left.

By induction, this proves that only a finite number of CHR constraints are added to the store. Therefore, Lemma 7.1 proves termination. □

Example 7.4 (Fibonacci continued)**.** *We prove termination for the Fibonacci example with the RC with substitutions.*

Let fib(n, m) be any query, with n a ground term, representing a natural number in successor-notation and m a free variable. One can infer that the call set is the set $\{fib(n_1, m), fib(n_2, n_3) \mid n_1, n_2, n_3$ ground terms, representing natural numbers and m a free variable$\}$. As a norm, we use term-size. The level mapping is defined as $|fib(n, m)| = ||n||_t$. Clearly, the call set is rigid w.r.t. $|.|$.

For the first rule, we have that for every matching substitution θ, the first term in every fib/2 constraint is substituted by the same ground term. The answer substitutions θ' are the identity substitutions because this rule has no guard:

$$|fib(N, M1)\theta| = |fib(N, M2)\theta| = |fib(N, M1)\theta\theta'| = ||N\theta\theta'||_t$$

All constraints have the same level value. There are two constraints in the head and one in the body, so this rule satisfies the RC with substitutions.

For the fourth rule we have that for every matching substitution θ, the term matching $s(s(N))$ is a ground term. The answer substitutions θ' are empty, because the rule has no guard. For all matching and answer substitutions:

$$|fib(s(s(N)), M)\theta| = 2 + ||N\theta||_t > |fib(s(N), M1)\theta\theta'| = 1 + ||N\theta\theta'||_t$$
$$|fib(s(s(N)), M)\theta| = 2 + ||N\theta||_t > |fib(N, M2)\theta\theta'| = ||N\theta\theta'||_t$$

The RC with substitutions is therefore satisfied, which implies termination for the considered queries. □

7.3.2 Ranking Condition for CHR

A disadvantage of the ranking condition of Definition 7.7, is that one has to consider all matching and answer substitutions. This cannot be done automatically because in general there can be infinitely many of such substitutions. In this section, we present a RC which is more suitable for automation. In order to estimate the effects of the matching substitutions, *abstract norms* and *abstract level mappings* are adapted from [18] and [35].

These functions map a variable to itself, instead of to zero. In order to estimate the effects of the answer substitutions, *interargument relations* are used.

Let $\mathbb{N}[Var_P]$ denote the set of polynomials over Var_P, with natural coefficients.

Definition 7.8 (abstract norm, abstract level mapping)**.** *An* ***abstract norm*** *is a mapping* $||.|| : Term_P \rightarrow \mathbb{N}[Var_P]$*, which is the identity function on* Var_P*. An* ***abstract level mapping*** *is a mapping* $|.| : Con_P \rightarrow \mathbb{N}[Var_P]$*.* □

Abstract list-length, abstract term-size and abstract linear level mappings are the obvious adaptations of Definitions 7.3 and Definition 7.4.

Example 7.5 (Abstract linear level mappings)**.** *Let L,List, N and M be variables. Typical examples of abstract linear level mappings are:*

$$\begin{array}{lclcl} |mergesort([L|List])|^{\alpha} & = & 0 + 1.||[L|List]||^{\alpha}_{l} & = & 1 + List \\ |fib(s(s(N)), M)|^{\alpha} & = & 0 + 1.||s(s(N))||^{\alpha}_{t} + 0.||M||^{\alpha}_{t} & = & 2 + N \end{array}$$

□

In general, an abstract level mapping maps constraints to arbitrary polynomials over $\mathbb{N}$. In order to compare the level values of the constraints w.r.t. an abstract level mapping, we define an ordering on polynomials over $\mathbb{N}$ [35].

Definition 7.9 (orderings on $\mathbb{N}[Var_P]$)**.** *Let p and q be two polynomials with n variables. The quasi-ordering* $\succeq$ *is defined as* $p \succeq q$ *iff* $p(x_1, \ldots, x_n) \geq q(x_1, \ldots, x_n)$ *for all* $x_1, \ldots, x_n \in \mathbb{N}$*. The strict ordering* $\succ$ *is defined as* $p \succ q$ *iff if* $p(x_1, \ldots, x_n) > q(x_1, \ldots, x_n)$ *for all* $x_1, \ldots, x_n \in \mathbb{N}$*. The equality between polynomials is defined as* $p \approx q$ *iff* $p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n)$ *for all* $x_1, \ldots, x_n \in \mathbb{N}$*.* □

The next example shows the orderings between three polynomials.

Example 7.6 (polynomial ordering)**.**
Let $p(X,Y) = 1 + XY + 2X$*,* $q(X) = 2X$ *and* $z(X,Y) = XY + X$*:*

- $p(X,Y) \succ q(X)$*,* $p(X,Y) \succ z(X,Y)$*.*
- *Neither* $q(X) \succeq z(X,Y)$ *nor* $z(X,Y) \succeq q(X)$*.* □

As stated, interargument relations are used to estimate the effect of the answer substitutions from Definition 7.1.

Definition 7.10 (Interargument relation)**.** *Let P be a program and p/n a built-in constraint in P. An* ***interargument relation*** *for p/n is a relation* $R_p \in \mathbb{N}^n$*.* R_p *is a valid interargument relation for p/n w.r.t. a norm* $||.||$*, iff*

$\forall t_1, ..., t_n \in Term_P : \; CT \models p(t_1, ..., t_n) \Longrightarrow (||t_1||, ..., ||t_n||) \in R_p.$ □

We illustrate this with some small examples.

Example 7.7 (Interargument relations).

built-in constraint	*norm and interargument relation*
delete(N1,N2,N3)	*list-length:* $\{(n_1, n_2, n_3) \in \mathbb{N}^3 \mid n_1 \geq n_3\}$
append(N1,N2,N3)	*list-length:* $\{(n_1, n_2, n_3) \in \mathbb{N}^3 \mid n_1 + n_2 = n3\}$
leq(N1,N2)	*term-size:* $\{(n_1, n_2) \in \mathbb{N}^2 \mid n_1 \leq n_2\}$

□

We define rigidity for abstract level mappings.

Definition 7.11 (Rigidity). *A CHR constraint C is **rigid** w.r.t. an abstract level mapping $|.|^\alpha$ iff $\forall$ substitutions θ : $|C|^\alpha \approx |C\theta|^\alpha$.* □

For an instance of a simplification or simpagation rule, the maximal level value is used in our RC with substitutions from Definition 7.7. Because the maximum is not defined for a set of polynomials, we introduce a $\succeq$*-maximal subset of constraints.*

Definition 7.12 ($\succeq$-maximal subset of constraints). *Let C be a multiset of CHR constraints and $|.|^\alpha$ an abstract level mapping. D is a **$\succeq$-maximal subset of C** w.r.t. $|.|^\alpha$ iff D is a non-empty multi-subset of C such that:*

- *The constraints in D have the same level value:*
  $\forall$ *CHR-constraints* $C1, C2 \in D : \; |C1|^\alpha \approx |C2|^\alpha$
- *There are no $|.|^\alpha$-larger constraints in $C \setminus D$:*
  $\neg(\exists C1 \in D, \exists C2 \in C \setminus D : \; |C2|^\alpha \succeq |C1|^\alpha)$ □

Note that a multiset C may have several $\succeq$-maximal subsets.

We now reformulate our ranking condition from Definition 7.7, using abstract level mappings and interargument relations.

Definition 7.13 (Ranking condition for CHR).
A program P and a query I satisfy the RC for CHR, w.r.t. an abstract level mapping $|.|^\alpha$ and a set of interargument relations iff every constraint in $Call(P, I)$ is rigid w.r.t. $|.|^\alpha$ and for each rule in P and for each substitution σ such that the built-in constraints in the guard all satisfy their associated interargument relations:

1. *For a simplification or simpagation rule $H \backslash H_1, ..., H_n \;\Leftrightarrow\; G \;\mid\; B_1, ..., B_m$, with $n > 0$, body-CHR constraints $B_k, ..., B_m$ and $D_1, ..., D_c$ all $\succeq$-maximal*

subsets of $\{H_1, ..., H_n, B_k\sigma, ..., B_m\sigma\}$*, then*

$$\forall i \in \{1, ..., c\} : \textit{let } D_i = \{H_{i_1}, ..., H_{i_p}, B_{i_1}\sigma, ..., B_{i_q}\sigma\}: \#\{H_{i_1}, ..., H_{i_p}\} > \#\{B_{i_1}\sigma, ..., B_{i_q}\sigma\}$$

2. *For a propagation rule:* $H_1, ..., H_n \Rightarrow G \mid B_1, ..., B_m$*, with body-CHR constraints* $B_k, ..., B_m$*:*
$$\forall i \in \{1, ..., n\}, \forall j \in \{k, ..., m\} : |H_i|^\alpha \succ |B_j\sigma|^\alpha. \qquad \square$$

Observe that we still have substitutions in the formulation of this condition. But, we can avoid reasoning about them explicitly in an automation. It is enough to reason about all instances that satisfy the interargument relations.

Theorem 7.2 (Sufficiency of the RC)**.** *Let program P with query I satisfy the RC for CHR w.r.t. $|.|^\alpha$ and some associated interargument relations for all built-in constraints in the program, then all computations for P with query I terminate.* $\square$

Proof. In order to prove termination of a CHR program P with a query I, it is sufficient to prove that the total number of CHR constraints, added during an execution of P with query I, is finite. We will prove this using induction.

Since the call set is rigid w.r.t. the level mapping $|.|^\alpha$, for each CHR constraint C, which is added to the constraint store by the query or an instance of a rule, $|C|^\alpha$ is a natural number.

Base case. The query is rigid w.r.t. $|.|^\alpha$, so there exists a maximal level value of the CHR constraints in the query, max. Because of the ranking condition, each instance of a propagation rule can only add constraints with a level value lower than max and each simplification or simpagation rule can only add CHR constraints with a level value max or lower to the store.

If program P with query I satisfies the RC for CHR with $|.|^\alpha$, all CHR constraints which are added to the store during any computation of P with query I have a natural number as level value w.r.t. $|.|^\alpha$ and there is a maximal level value max. Every time a CHR constraint of level value max is added by such a rule, the number of CHR constraints with level value max decreases. So only a finite number a_{max} of CHR constraints with level value max are added to the store during any execution of P for I.

Induction step. Let $a_{max}, ..., a_{n+1}$ be upper limits for the number of CHR constraints with level value $max, ..., n+1$ which can be added to the constraint store during an execution of P with query I.

- The query only contains a finite number of constraints with level value n.
- For each propagation rule which satisfies the RC w.r.t. $|.|^{\alpha}$, every head constraint has a larger level value, w.r.t. $\succ$, than every CHR constraint in the body.
  If an instance of a propagation rule adds a CHR constraint with level value n to the store, the CHR constraints matching the head all have a natural number larger than n as a level mapping. Because of the upper limits $a_{max}, ..., a_{n+1}$ and the propagation history, every propagation rule can only add a finite number of CHR constraints with level value n.
- For each simplification or simpagation rule which satisfies the RC w.r.t. $|.|^{\alpha}$, every maximal subset of CHR constraints has more constraints from the head than from body of the rule. For every instance of a simplification or simpagation rule with maximal subsets $D_1, ..., D_n$, there are maximally ranked subsets $D_{i_1}, ..., D_{i_p}$, which contain the CHR constraints with the highest level value, m, w.r.t. $|.|^{\alpha}$. Since the program satisfies the RC for CHR, the number of CHR constraints with level value m decreases and all CHR constraints added by the body of the rule have a level mapping smaller than or equal to m. This implies that only a finite number of constraints with level mapping n can be added to the store by instances of simpagation and simplification rules, because after enough rule executions, the constraint store has no CHR constraints with a level value n or higher left.

By induction, this proves that only a finite number of CHR constraints are added to the store. Therefore, Lemma 7.1 proves termination. □

We prove termination for the Fibonacci program with this RC.

Example 7.8. *As in Example 7.4, the query is $fib(n, m)$ with n a ground term, representing a natural number in successor-notation and m a free variable. The call set is $\{fib(n_1, m), fib(n_2, n_3) \mid n_1, n_2, n_3$ ground terms, representing natural numbers and m a free variable$\}$. We use the abstract level mapping: $|fib(n, m)|^{\alpha} = ||n||_t^{\alpha}$.*

The call set is rigid w.r.t. the chosen abstract level mapping. For the first rule, $\{fib(N, M1), fib(N, M2), fib(N, M1)\}$ is a $\succeq$-maximal subset w.r.t. $|.|^{\alpha}$. With two of these constraints in the head of the rule and only one in the body, this rule satisfies the RC for CHR. The second and third rule trivially satisfy the RC for CHR because they have no CHR constraints in the body. The last rule is a propagation rule. To satisfy the RC for CHR, every constraint matching *the head of the rule must be larger than every constraint added by the body of the rule.*

$$|fib(s(s(N)), M)|^{\alpha} = 2 + N \succ |fib(s(N), M1)|^{\alpha} = 1 + N$$
$$|fib(s(s(N)), M)|^{\alpha} = 2 + N \succ |fib(N, M2)|^{\alpha} = N. \qquad \square$$

Because no interargument relations are needed for the Fibonacci example, another small example is given.

Example 7.9 (Greatest common divisor - gcd)**.** *We prove termination for gcd with the RC for CHR. This program calculates the greatest common divisor of a set of positive integer numbers for a query: $gcd(n)$ ($n \in \mathbb{N}$).*

$$gcd(0) \Leftrightarrow true.$$
$$gcd(M) \setminus gcd(N) \Leftrightarrow N >= M, M > 0, NN \ is \ N - M \mid gcd(NN).$$

The call set is $\{gcd(n) \mid n \text{ a natural number}\}$. As an abstract norm, we map each natural number or variable to itself. The associated interargument relations for the guard of the second rule are obvious (e.g. $\{(n,m) \in \mathbb{N}^2 \mid n \geq m\}$ for $N >= M$). For any instance $gcd(m) \setminus gcd(n) \Leftrightarrow n >= m, m > 0, nn \ is \ n - m \mid gcd(nn)$. satisfying the interargument relations of the guard, we have $n > nn$.

The abstract level mapping $|gcd(n)|^{\alpha} = n$, proves termination w.r.t. the associated interargument relations and the chosen norm.

The call set is rigid w.r.t. the abstract level mapping. Because the first rule has no CHR constraints in the body, it trivially satisfies the RC with $\{gcd(0)\}$ as a $\succeq$-maximal subset. The second rule replaces a constraint $gcd(n)$, mapped to n, by the constraint $gcd(nn)$, mapped to nn. Since the associated interargument relations for this rule need to be satisfied: $n > nn$. Therefore, $\{gcd(N)\}$ is the only $\succeq$-maximal subset of the rule. Since both rules satisfy the RC, this program terminates for the considered queries. $\square$

7.4 Automation and experimental evaluation

We have implemented a prototype analyzer to allow us to experimentally evaluate our approach. The prototype is implemented in CHR, with SWI-Prolog as host-language. Space restrictions do not allow us to give a full account of the system. In this section, we focus on the most central components only and illustrate the general strategy with an example.

Example 7.10 (Employee availability)**.** *This program computes a list of all employees who have no appointment on a certain time point.*

> $app(Tb, Te, E), empl(L), time(T) \;\Rightarrow\; member(E, L), Tb \leq T, T \leq Te \mid remove(E).$
>
> $empl(L_1), remove(E) \Leftrightarrow select(E, L_1, L_2) \mid empl(L_2).$

Here, $select(e, l_1, l_2)$ only succeeds if the list l_1 contains e and l_2 is the list obtained from l_1 by removing one occurrence of e.

The intention is that a query consists of constraints $app/3$, with a begin time, end time and employee's name, representing appointments, a constraint $empl/1$, with as only argument the list of all employee's names and a constraint $time/1$, with as its argument the time point on which we want the list of available employees. The propagation rule determines which employees need to be removed from the list. The simplification rule deletes employees from the list. The program terminates for all considered queries. □

To prove termination of a CHR program with the RC for CHR, one has to find a norm, associated interargument relations and an appropriate abstract level mapping, such that the RC is satisfied. We use abstract linear level mappings. In the philosophy of the constraint based approach to termination analysis, as described in [18], we introduce a symbolic form of the level mapping.

Example 7.11 (Employee availability continued)**.** *For the constraints in employee availability we have the symbolic forms: $|time(t)|^\alpha = time_0 + time_1.||t||^\alpha$,*
$|app(t_b, t_e, e)|^\alpha = app_0 + app_1.||t_b||^\alpha + app_2.||t_e||^\alpha + app_3.||e||^\alpha$,
$|empl(e)|^\alpha = empl_0 + empl_1.||e||^\alpha$,
$|remove(e)|^\alpha = remove_0 + remove_1.||e||^\alpha$. □

Following the constraint based approach, the aim is to translate the conditions imposed by the RC into constraints of the symbolic coefficients (e.g. $app_0, app_1, app_2, app_3$) of the level mapping. This translation must be such that every solution for the resulting constraints corresponds to one way of satisfying the RC, and therefore, of obtaining a termination proof.

First, we perform a simple type inference to compute an overestimation of the call set. It is based on four basic types: *ground list*, *nil-terminated list*, *ground term* and *any term*. We initialize the call set with the given query type. Until a fix point is reached, constraints from the call set are matched with each rule's head. For every applicable rule, the effect of the guard is analyzed and the call types of the CHR body constraints are added to the call set.

Example 7.12 (Employee availability continued)**.** *We initialize the call set as the set of all typed constraints in the query:* $Call = \{app(t1, t2, e), empl(l), time(t3) \mid t1, t_2, t_3, e$ *ground terms and* l *a ground list*$\}$*. Only the first rule adds a new typed constraint, namely* $remove(e)$ *with* e *a ground term. The fix point is reached and the call set is* $Call \cup \{remove(e) \mid e$ *a ground term*$\}$*.* □

Next, we choose between term-size and list-length as the norm. The call set can provide us with information on how to select between them. But, the choice is backtrackable.

Example 7.13 (Employee availability continued)**.** *Due to the groundness of all terms in the call set, we first select term-size. It turns out that the analyzer fails for this choice. We then set the norm to list-length.* □

Now, we impose rigidity of the level mapping on the call set. We also ensure that the level mapping only measures arguments of the appropriate type.

Example 7.14 (Employee availability continued)**.** *The constraints* $app/3$*,* $time/1$ *and* $remove/1$ *have no list-arguments. Thus, we measure no arguments:*

$$|app(t_b, t_e, e)|^\alpha = app_0 \qquad |time(t)|^\alpha = time_0 \qquad |remove(e)|^\alpha = remove_0$$

The only argument of $empl/1$ *is a ground list, so* $|empl(l)|^\alpha = empl_0 + empl_1.||l||_l^\alpha$ *is rigid on the call set.* □

In a next step, interargument relations are inferred for the guards. Our prototype implementation contains predefined interargument relations under list-length and term-size for built-in predicates. It requires the user to provide interargument relations for other guards. This can easily be further automated in the future by using interargument relation inference from another analyzer (e.g. Polytool [35]).

Example 7.15 (Employee availability continued)**.** *For* $member/2$ *and* $=< /2$ *there are no 'real' interargument relations under list-length, so that the relations are* $\mathbb{N}^2$*. For* $select/3$*, the associated interargument relation is*
$\big\{(n_1, n_2, n_3) \in \mathbb{N}^3 \mid n_2 = n_3 + 1\big\}$*.* □

We then reach the most central part of the system. Here, we set up constraints on the remaining symbolic coefficients of the level mapping, corresponding to the conditions expressed in the RC.

For propagation rules, this is fairly easy: provided that interargument relations hold of the guards, the level mapping should decrease from head to body constraints.

possible $\succeq$-maximal subsets	Resulting set of equations
$\{empl(L)\}$	$\lvert empl(L_1)\rvert^\alpha \succ \lvert empl(L_2)\rvert^\alpha \wedge \lvert empl(L_1)\rvert^\alpha \succ \lvert remove(E)\rvert^\alpha$
$\{remove(E)\}$	$\lvert remove(E)\rvert^\alpha \succ \lvert empl(L_1)\rvert^\alpha \wedge \lvert remove(E)\rvert^\alpha \succ \lvert empl(L_2)\rvert^\alpha$
$\{empl(L_1)\}, \{remove(E)\}$	$(\lvert remove(E)\rvert^\alpha \succ \lvert empl(L_2)\rvert^\alpha \vee \lvert empl(L_1)\rvert^\alpha \succ \lvert empl(L_2)\rvert^\alpha)$ $\wedge\neg(\lvert empl(L_1)\rvert^\alpha \succeq \lvert remove(E)\rvert^\alpha)$ $\wedge\neg(\lvert remove(E)\rvert^\alpha \succeq \lvert empl(L_1)\rvert^\alpha)$
$\{empl(L_1), remove(E)\}$	$\lvert empl(L_1)\rvert^\alpha \succ \lvert empl(L_2)\rvert^\alpha \wedge \lvert empl(L_1)\rvert^\alpha \approx \lvert remove(E)\rvert^\alpha$
$\{empl(L_1), remove(E), empl(L_2)\}$	$\lvert empl(L_1)\rvert^\alpha \approx \lvert empl(L_2)\rvert^\alpha \wedge \lvert empl(L_1)\rvert^\alpha \approx \lvert remove(E)\rvert^\alpha$

Table 7.1: $\succeq$-maximal subsets for employee availability

Example 7.16 (Employee availability continued). *For the propagation rule, since the interargument relations for the guard are trivial, we get:*

$$\begin{array}{lclclcl}
\lvert app(T_b,T_e,E)\rvert^\alpha & \succ & \lvert remove(E)\rvert^\alpha & & app_0 & \succ & remove_0 \\
\lvert time(T)\rvert^\alpha & \succ & \lvert remove(E)\rvert^\alpha & or & time_0 & \succ & remove_0 \\
\lvert empl(L_1)\rvert^\alpha & \succ & \lvert remove(E)\rvert^\alpha & & empl_0 + empl_1.L_1 & \succ & remove_0 \quad (1)
\end{array}$$

□

For simplification and simpagation rules this step is more difficult. The problem is that we need to reason about $\succeq$-maximal subsets of constraints but that these subsets depend on the used level mapping – which, as yet, has not been fixed.

To solve this, we will compute all the possibilities of candidate $\succeq$-maximal subsets, which are such that the condition in the RC is fulfilled. For each of these possibilities, we then express what conditions on the level mapping are required to make these subsets the $\succeq$-maximal ones. If any of these conditions on the level mapping can be satisfied, the RC holds for the rule.

Example 7.17 (Employee availability continued). *In the first column of Table 1, we present all the different cases of possible $\succeq$-maximal subsets for the simplification rule, which are such that the RC is fulfilled. The second column of the table contains the corresponding condition on the level mapping needed for $\succeq$-maximality of these sets.*

Observe that for the four last rows, the resulting equations are inconsistent with the constraint (1) obtained for the propagation rule. For instance, for the second row, $\lvert remove(E)\rvert^\alpha \succ \lvert empl(L_1)\rvert^\alpha$ is inconsistent with $\lvert empl(L_1)\rvert^\alpha \succ \lvert remove(E)\rvert^\alpha$.

As a result, the first row is the only remaining candidate constraint. Since the interargument relation of the guard in the simplification rule is non-trivial, we obtain the condition:

$L_1 = L_2 + 1 \implies |empl(L_1)|^\alpha \succ |empl(L_2)|^\alpha \wedge |empl(L_1)|^\alpha \succ |remove(E)|^\alpha$

or $L_1 = L_2 + 1 \implies empl_0 + empl_1.L_1 \succ empl_0.L_2 + empl_1.L_2 \wedge$

$$empl_0 + empl_1.L_1 \succ remove_0 \qquad \square$$

Finally, all conditions on the symbolic coefficients are collected and transformed into Diophantine constraints, by using the method described in [35]. These equations are solved using the constraint solver of AProVE ([23]).

Example 7.18 (Employee availability continued).
AProVE finds the solution $app_0 = empl_0 = time_0 = empl_1 = 1$*,* $remove_0 = 0$*, corresponding to the level mapping:*

$\|app(Tb, Te, E)\|^\alpha = 1$	$\|empl(L)\|^\alpha = 1 + \|\|L\|\|_l^\alpha$
$\|remove(E)\|^\alpha = 0$	$\|time(T)\|^\alpha = 1$

$\square$

7.4.1 Experimental evaluation

There are some benchmarks available in the CHR community. Because they are aimed at testing performance issues, they are relatively small and not useful for testing a termination analyzer. Therefore, we collected a number of programs from various sources to set up a new benchmark. It contains 52 programs: 39 without propagation and 13 with propagation. All programs are terminating.

We compared our prototype analyzer with the transformational analyzer of [42], which cannot deal with propagation. The results are very satisfactory. Our analyzer proves termination for 31 programs without propagation. The analyzer of [42] proves termination of 26 of these programs. The transformational analyzer proves termination of two programs where our approach fails. This is because these programs need a more complex norm as list-length or term-size. The results are presented in Tables 7.2 and 7.3. The upper half of the tables are programs from [42]. The lower part are programs from webCHR[1] except for employee, which is the employee availability example from Section 4. **Imp** gives the results of our prototype implementation. **[42]** gives the results of the transformational approach in [42].

Our implementation proves 10 programs with propagation from the benchmark terminating. The failing programs increase an arguments size till a certain bound. Our approach cannot deal with bounded increases. For term rewrite systems, a technique to prove termination for such programs is discussed in [24].

[1]http://chr.informatik.uni-ulm.de/ webchr/

Name	**Imp**	[42]	**Name**	**Imp**	[42]
ackermann	-	-	modulo	+	+
average	+	+	oddeven	+	+
binlog	+	+	pathcons	-	+
booland	+	+	power	+	+
boolcard	+	+	revlist	+	+
concat	-	-	som	+	+
convert	+	-	toyama	-	+
diff	+	+	weight	+	-
factorial	+	+	bool and$_{tr}$	+	+
gcd	+	+	even$_{tr}$	+	+
genint1	+	+	fib tabulation$_{tr}$	+	-
joinlists	+	+	fibbonaci$_{tr}$	+	-
max	+	+	genint$_{tr}$	-	-
mean	+	+	primes$_{tr}$	+	-
mergesort	+	+			
dfsearch	+	+	primes3	-	-
lex	+	+	succ add	+	+
min	+	+	sudoku	-	-
nqueen2	+	-	tak	-	-
primes2	+	+	zebra	+	-

Table 7.2: Result for programs without propagation

Name	**Imp**
boolean and	+
even	+
fib tabulation	+
fibonacci	+
genint	-
primes	+
employee	+
autogen booland	+
family	+
fib bottomup	-
fib top down	+
primes1	+
sorting	-

Table 7.3: Programs for programs with propagation

7.5 Summary

We discussed a new approach to termination analysis of CHR programs. Before 2008, automated termination analysis was restricted to CHR programs with simplification only. Our condition allows for termination analysis of general CHR programs, that is, CHR programs with propagation as well. We have implemented the technique in an automated system. Experimental results with this system show that it is successful in proving termination for a majority of the test set programs.

The condition on simplification rules, as proposed in [21], was strengthened in our RC in order to be able to extend it with a condition for propagation rules. Therefore, a small class of CHR programs cannot be proved terminating with our approach, where the (non-automated) approach of [21] succeeds.

7.5.1 Later works on Termination analysis of CHR

After the publication of the work described in this chapter, P. Pilozzi introduced a new ranking condition for CHR with propagation in [39]. The analysis of [39] allows to combine our ranking condition for propagation rules with multiset decreases for simplification rules as in [21]. The resulting analysis is strictly stronger than both the termination condition of [21] and the termination condition described in this chapter.

Further improvements on termination analysis of CHR were published in [40] and [41].

Reference

The work in this chapter was presented at LOPSTR 2008 and is published in [57].

Chapter 8

Conclusion

Chapter 3 presented an approximation framework for attacking the undecidable termination problem of logic programs, as an alternative to current termination/non-termination proof approaches. We introduced an idea of termination prediction, established a necessary and sufficient characterization of infinite SLDNF-derivations with arbitrary (concrete or moded) queries, built a new loop checking mechanism, and developed an algorithm that predicts termination of general logic programs with arbitrary queries. We implemented the analysis and demonstrated the effectiveness of the termination prediction with representative examples including ones borrowed from the Termination Competition 2007.

Chapter 4 introduced a new approach to non-termination analysis of logic programs based on the symbolic derivation tree for a moded query defined in Chapter 3. This symbolic derivation tree represents the derivations of all concrete queries denoted by the moded query. We introduced a non-termination condition that identifies paths in this tree that can be repeated infinitely. We implemented the approach and evaluated it on a benchmark of 48 non-terminating programs from the termination competition of 2007. Our tool, $P2P$, proves non-termination of all benchmark programs. We have shown that our technique improves on the results of the only non-termination analyzer developed before our work, NTI, and that we can handle 2 new classes of programs.

In Chapter 5, we identified classes of logic programs for which previous analyzers fail to prove non-termination and we extended our non-termination analysis to overcome these limitations. The most important class of programs for which previous analyzers fail, are programs for which no fixed sequence of clauses can be repeated infinitely. We have shown that non-failure information [17] can be used to abstract away from the exact sequence of clauses needed to solve non-failing goals. To use this non-failure information, type information is added to the symbolic derivation tree and a special transition is introduced to solve non-failing atoms. We have shown that program specialization [26] can be used to overcome another limitation of previous analyzers. If non-termination can not be proven due to aliased variables, redundant argument filtering may remove these duplicated variables from the program. Specialization can also be used in combination with the NFG transition. Program specialization may produce more instantiated clauses, giving a better approximation of the possible values after solving the non-failing goal.

In Chapter 6, we took a first step towards the analysis of real life Prolog programs by extending our non-termination analysis for logic programs with integer arithmetic. The analysis starts with a given program and class queries, specified using modes, and detects subclasses of non-terminating queries. First, the derivations for the given class of queries are abstracted by building a symbolic derivation tree with transitions to handle integer arithmetic. Then, this symbolic derivation tree is used to detect subclasses of non-terminating queries in two phases. The first phase focuses on the logic part of the computation and assumes the conditions over integers, e.g. $> /2$, succeed. It detects classes of queries that are non-terminating if all conditions on integers in those derivations succeed. The second phase of the analysis adds constraints over the integer arguments of the query to obtain a class of non-terminating queries. These constraints guarantee that all conditions on integers in the derivation will succeed. The constraints are then solved by transforming them into a SAT problem.

Chapter 7 introduced a termination analysis approach for Constraint Handling Rules, a constraint based programming language. Before 2008, automated termination analysis was restricted to CHR programs with simplification only. Our condition allows for termination analysis of general CHR programs, that is, CHR programs with propagation as well. We have implemented the technique in an automated system. Experimental results with this system show that it is successful in proving termination for a majority of the benchmark programs.

8.1 Symbolic execution

To finish the conclusion, we will situate our work in the broader context of symbolic execution [13]. Symbolic execution entails different analyses that use symbolic values instead of concrete inputs to test program properties. Program paths can then be explored by keeping a symbolic state which can identify concrete inputs for which the path is applicable. Applications of symbolic execution include test input generation, error detection such as detection of uncaught exceptions but also more complex analyses such as security testing.

This description fits our work on termination prediction and non-termination analysis very well. Like in symbolic execution, we keep a symbolic state and explore different program paths to detect a non-terminating goal. When such a goal is found we can generate a class of concrete non-terminating queries using information in the moded SLD-tree.

As the similarities between symbolic execution and our work suggest, our termination prediction and non-termination analyses could be extended using ideas from symbolic execution. Using techniques from symbolic execution concerning constraint solving for example, could make big improvements in analyzing programs with integer arithmetics.

Bibliography

[1] Termination competition 2007. `http://www.lri.fr/~marche/termination-competition/2007`, 2007.

[2] Tpolp: Termination prediction of logic programs. `http://users.skynet.be/fa891603/TPoLP`, 2009.

[3] Abdennadher, S. Operational semantics and confluence of constraint propagation rules. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming, CP97* (1997), springer, pp. 252–266.

[4] Aguzzi, G., and Modigliani, U. Proving termination of logic programs by transforming them into equivalent term rewriting systems. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science* (London, UK, UK, 1993), Springer-Verlag, pp. 114–124.

[5] Apt, K. R., and Pedreschi, D. Reasoning about termination of pure Prolog programs. *Information and Computation 106* (1993), 109–157.

[6] Arts, T., and Zantema, H. Termination of logic programs using semantic unification. In *Logic Programming Synthesis and Transformation, 5th International Workshop, LOPSTR 95* (1996), Springer Verlag, pp. 219–233.

[7] Bol, R. *Loop checking in logic programming.* CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1995.

[8] Bol, R. N., Apt, K. R., and Klop, J. W. An analysis of loop checking mechanisms for logic programs. *Theoretical Computer Science 86*, 1 (Aug. 1991), 35–79.

[9] Bossi, A., Cocco, N., Rossi, S., and Etalle, S. On modular termination proofs of general logic programs. *Theory and Practice of Logic Programming 2*, 3 (May 2002), 263–291.

[10] Bruynooghe, M., Codish, M., Gallagher, J. P., Genaim, S., and Vanhoof, W. Termination analysis of logic programs through combination of type-based norms. *ACM Transactions on Programming Languages and Systems (TOPLAS) 29*, 2 (apr 2007).

[11] Bruynooghe, M., De Schreye, D., and Martens, B. A general criterion for avoiding infinite unfolding during partial deduction. *New Generation Computing 11*, 1 (1992), 47–79.

[12] Bruynooghe, M., Gallagher, J., and Humbeeck, W. V. Inference of well-typing for logic programs with application to termination analysis. In *Static Analysis, 12th International Symposium, SAS 2005* (2005), vol. 3672 of *LNCS*, Springer, pp. 35–51.

[13] Cadar, C., and Sen, K. Symbolic execution for software testing: Three decades later. *Commun. ACM 56*, 2 (Feb. 2013), 82–90.

[14] Clark, K. L. Negation as failure. In *Logic and Data Bases*, J. Minker, Ed., vol. 1. Plenum Press, New York, London, 1978, pp. 293–322.

[15] Codish, M. Proving termination with (boolean) satisfaction. In *The 17th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2007* (2008), A. King, Ed., vol. 4915 of *Lecture Notes in Computer Science*, pp. 1–7.

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

[17] Debray, S. K., López-García, P., and Hermenegildo, M. V. Non-failure analysis for logic programs. In *Proceedings of the 14th International Conference on Logic Programming, ICLP 1997* (1997), pp. 48–62.

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

[19] Eén, N., and Sörensson, N. An extensible SAT-solver. In *SAT* (2003), E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *Lecture Notes in Computer Science*, Springer, pp. 502–518.

[20] Frühwirth, T. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming 37*, 1–3 (oct 1998), 95–138.

[21] Frühwirth, T. Proving termination of constraint solver programs. In *Selected Papers of New Trends in Contraints, Joint ERCIM/Compulog Net Workshop* (2000), vol. 1865 of *LNCS*, springer, pp. 298–317.

[22] Genaim, S., and Codish, M. Inferring termination conditions for logic programs using backwards analysis. In *Proceedings of the Eighth International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (London, UK, UK, 2001), LPAR '01, Springer-Verlag, pp. 685–694.

[23] Giesl, J., Schneider-kamp, P., and Thiemann, R. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *In Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR '06* (2006), Springer-Verlag, pp. 281–286.

[24] Giesl, J., Thiemann, R., Swiderski, S., and Schneider-Kamp, P. Proving termination by bounded increase. In *CADE* (2007), F. Pfenning, Ed., vol. 4603 of *Lecture Notes in Computer Science*, Springer, pp. 443–459.

[25] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, 2002.

[26] Leuschel, M. Logic program specialisation. In *Partial Evaluation* (1998), vol. 1706 of *LNCS*, Springer, pp. 155–188.

[27] Lindenstrauss, N., and Sagiv, Y. Automatic termination analysis of logic programs. In *Proceedings of the Fourteenth International Conference on Logic Programming* (1997), MIT Press, pp. 63–77.

[28] Lloyd, J. W. *Foundations of Logic Programming*, 2nd ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[29] Lu, L. A mode analysis of logic programs by abstract interpretation. In *Ershov Memorial Conference* (1996), D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181 of *Lecture Notes in Computer Science*, Springer, pp. 362–373.

[30] Marchiori, E. Practical methods for proving termination of general logic programs. *Journal of Artificial Intelligence Research 4*, 1 (apr 1996), 179–208.

[31] Marchiori, M. Proving existential termination of normal logic programs. In *Proceedings of the 5th AMAST, LNCS 1101* (1996), pp. 375–390.

[32] Martens, B., and De Schreye, D. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming 28*, 2 (1996), 89–146.

[33] Mesnard, F., and Bagnara, R. cTI: a constraint-based termination inference tool for iso-Prolog. *Theory and Practice of Logic Programming 5*, 1-2 (jan 2005), 243–257.

[34] Mesnard, F., and Neumerkel, U. Applying static analysis techniques for inferring termination conditions of logic programs. In *Proceedings of the 8th International Symposium on Static Analysis, SAS 2001* (London, UK, UK, 2001), SAS '01, Springer-Verlag, pp. 93–110.

[35] Nguyen, M. T., De Schreye, D., Giesl, J., and Schneider-Kamp, P. Polytool: Polynomial interpretations as a basis for termination analysis of logic programs. *Theory and Practice of Logic Programming 11*, 1 (2011), 33–63.

[36] Ohlebusch, E., Claves, C., and Marché, C. Talp: A tool for the termination analysis of logic programs. In *In Proceedings of 11th RTA, LNCS 1833* (2000), Springer-Verlag, pp. 270–273.

[37] Payet, E. Detecting non-termination of term rewriting systems using an unfolding operator. In *Proceedings of the 16th international conference on Logic-based program synthesis and transformation* (Berlin, Heidelberg, 2007), LOPSTR 06: International Symposium on Logic-based Program Synthesis and Transformation, Springer-Verlag, pp. 194–209.

[38] Payet, É., and Mesnard, F. Nontermination inference of logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 28*, 2 (2006), 256–289.

[39] Pilozzi, P., and De Schreye, D. Termination analysis of chr revisited. In *ICLP* (2008), M. G. de la Banda and E. Pontelli, Eds., vol. 5366 of *Lecture Notes in Computer Science*, Springer, pp. 501–515.

[40] Pilozzi, P., and De Schreye, D. Proving termination by invariance relations. In *ICLP* (2009), P. M. Hill and D. S. Warren, Eds., vol. 5649 of *Lecture Notes in Computer Science*, Springer, pp. 499–503.

[41] Pilozzi, P., and De Schreye, D. Improved termination analysis of chr using self-sustainability analysis. In *LOPSTR* (2011), G. Vidal, Ed., vol. 7225 of *Lecture Notes in Computer Science*, Springer, pp. 189–204.

[42] Pilozzi, P., Schrijvers, T., and De Schreye, D. Proving termination of CHR in Prolog: a transformational approach. In *9th International Workshop on Termination* (2007).

[43] Ramakrishnan, I. V., Rao, P., Sagonas, K. F., Swift, T., and Warren, D. S. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38*, 1 (1999), 31–54.

[44] Rao, M. R. K. K., Kapur, D., and Shyamasundar, R. K. Transformational methodology for proving termination of logic programs. *Journal of Logic Programming 34*, 1 (1998), 1–41.

[45] Sahlin, D. Mixtus: An automatic partial avaluator for full Prolog. *New Generation Computing 12*, 1 (1993), 7–51.

[46] Schneider-Kamp, P., Giesl, J., Serebrenik, A., and Thiemann, R. Automated termination proofs for logic programs by term rewriting. *ACM Trans. Comput. Log. 11*, 1 (2009).

[47] Schrijvers, T. *Analyses, Optimizations and Extensions of Constraint Handling Rules.* PhD thesis, K.U.Leuven, Leuven, Belgium, june 2005.

[48] Schrijvers, T., and Frühwirth, T. Optimal Union-Find in Constraint Handling Rules. *Theory and Practice of Logic Programming 6*, 1&2 (2006).

[49] Shen, Y.-D. An extended variant of atoms loop check for positive logic programs. *New Generation Computing 15* (1997), 187–204.

[50] Shen, Y.-d., De Schreye, D., and Voets, D. Termination prediction for general logic programs. *Theory and Practice of Logic Programming 9*, 6 (2009), 751–780.

[51] Shen, Y.-D., huai You, J., yan Yuan, L., Shen, S. S. P., and Yang, Q. A dynamic approach to characterizing termination of general logic programs. *ACM Transactions on Computational Logic 4*, 4 (2003), 417–430.

[52] Shen, Y.-D., yan Yuan, L., and huai You, J. Loop checks for logic programs with functions. *Theoretical Computer Science 266* (2001), 441–461.

[53] Van Gelder, A. Efficient loop detection in Prolog using the tortoise-and-hare technique. *Journal of Logic Programming 4*, 1 (1987), 23–31.

[54] van Raamsdonk, F. Translating logic programs into conditional rewriting systems. In *Proceedings of the Fourteenth International Conference on Logic Programming, ICLP 1997* (1997), pp. 168–182.

[55] Voets, D., and De Schreye, D. Non-termination analysis of logic programs using types. In *Proceedings of the 20th international conference on Logic-based program synthesis and transformation, LOPSTR 2010* (Berlin, Heidelberg, 2011), LOPSTR'10, Springer-Verlag, pp. 133–148.

[56] Voets, D., and De Schreye, D. Non-termination analysis of logic programs with integer arithmetics. *Theory and Practice of Logic Programming 11*, 4-5 (2011), 521–536.

[57] VOETS, D., PILOZZI, P., AND DE SCHREYE, D. A new approach to termination analysis of CHR. In *Pre-Proceedings of the 18th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2008* (2008), M. Hanus, Ed., pp. 28–42.

[58] VOETS, D., AND SCHREYE, D. A new approach to non-termination analysis of logic programs. In *Proceedings of the 25th International Conference on Logic Programming, ICLP 2009* (Berlin, Heidelberg, 2009), ICLP 09: International Conference on Logic Programming, Springer-Verlag, pp. 220–234.

[59] WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. SWI-Prolog. *CoRR abs/1011.5332* (2010).

[60] YARDENI, E., AND SHAPIRO, E. A type system for logic program. *Journal of Logic Programming 10*, 2 (jan 1991), 125–153.

Biography

Dean Voets was born on the 14th of September 1985 in Lier, Belgium. He grew up in Ranst and graduated from Sint-Gabriel College Boechout in 2003. He studied Computer Science at the Katholieke Universiteit Leuven and finished the degree of Master in Computer Science in 2007. His master´s thesis handled the topic of termination analysis of Constraint Handling Rules and was supervised by Professor Danny De Schreye.

In October 2007, Dean started working as a Ph.D. student at the Department of Computer Science of the Katholieke Universiteit Leuven. He joined the Analysis subgroup of DTAI (Declarative Languages and Artificial Intelligence) to work on Termination Analysis of Declarative Languages. His work was supervised by Professor Danny De Schreye. Dean was funded by the Fund for Scientific Research (FWO) in Flanders (Belgium).

List of publications

Publications in Journals

- Yi-dong Shen, Danny De Schreye, Dean Voets. Termination prediction for general logic programs. Theory and Practice of Logic Programming, volume 9 (6), pages 751-760.
- Dean Voets, Danny De Schreye. Non-termination analysis of logic programs with integer arithmetics. Theory and Practice of Logic Programming volume 11 (4-5), pages 521-536.

Publications at International Conferences

- Dean Voets, Paolo Pilozzi, Danny De Schreye. A new approach to termination analysis of CHR. 18th International Symposium on Logic-Based Program Synthesis and Transformation. Valencia, Spain. July 17-18, 2008. Pre-proceedings, pages 28-42.
- Dean Voets, Danny De Schreye. A New Approach to Non-termination Analysis of Logic Programs. 25th International Conference on Logic Programming. Pasadena, California, USA. July 14-17, 2009. Proceedings, pages 220-234.
- Dean Voets, Danny De Schreye. Non-termination analysis of logic programs using types. 20th international conference on Logic-based program synthesis and transformation. Hagenberg, Austria. July 23-25, 2010. Pre-Proceedings, pages 234-248.
- Dean Voets, Danny De Schreye. Non-termination analysis of logic programs using types. 20th international conference on Logic-based

program synthesis and transformation. Hagenberg, Austria. July 23-25, 2010. Proceedings, pages 133-148.

Publications at International Workshops

- Dean Voets, Paolo Pilozzi, Danny De Schreye. A new approach to termination analysis of Constraint Handling Rules. 9th International Workshop on Termination. Paris, France. 29 June 2007. Proceedings, pages 26-29.
- Dean Voets, Paolo Pilozzi, Danny De Schreye. A new approach to termination analysis of Constraint Handling Rules. 4th International Workshop on Constraint Handling Rules. Porto, Portugal. 8 September 2007. Proceedings, pages 77-89.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DTAI
Celestijnenlaan 200A box 2402
B-3001 Heverlee
voets.dean@gmail.com
http://www.cs.kuleuven.be