Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Informatik 2
Programmiersprachen und Verifikation

Analyzing Arithmetic Prolog Programs by Symbolic Execution

Alexander Dominik Weinert

Masterarbeit
im Studiengang Informatik

Gutacher: Prof. Dr. Jürgen Giesl
Prof. Gerhard Lakemeyer, Ph.D.
Betreuer: Dipl.-Inf. Thomas Ströder

Abstract

Automated analysis of programs has been of interest for a long time now. One particular property of interest is termination behavior, that is, whether or not some program is guaranteed to terminate after a finite amount of steps.

In this work we develop an automated termination analysis for a fragment of the programming language PROLOG. This fragment consists of the purely logical fragment of PROLOG and the cut, as well as of arithmetic comparisons and evaluations. Even though there exist approaches that claim to analyze programs in PROLOG, most of them are constrained to a small subset of the language, so-called definite programs. These approaches do not take the more complicated features of the PROLOG language, such as the cut, arithmetic comparisons or arithmetic evaluations, into account.

To develop this analysis, we develop an abstract semantics that allows us to reason over sets of executions of a PROLOG program written in this fragment. The objects of this semantics are abstract program graphs that describe all possible evaluations of a given query. This abstract graph-based semantics is based on existing work, which we extend to take arithmetic comparisons and evaluations into account.

We then construct an integer transition system from the abstract program graph, the termination of which implies termination of the given program. The resulting transition system can then be analyzed for termination using existing methods.

We have implemented our approach in the termination prover APROVE. Using this implementation, we have conducted experiments using a broad set of benchmarks and showed that our approach offers an improvement over existing methods, both in terms of runtime and in terms of power.

Zusammenfassung

Die automatisierte Analyse von Programmen wird seit langem erforscht. Von besonderem Interesse ist dabei die Terminierungsanalyse, also die Frage, ob ein gegebenes Programm nach endlich vielen Schritten terminiert.

In dieser Arbeit entwickeln wir eine automatisierte Terminierungsanalyse für ein Fragment der Programmiersprache PROLOG. Dieses Fragment besteht aus dem rein logischen Fragment PROLOGs, dem Cut, sowie arithmetischen Vergleichen und Auswertungen. Obwohl es Ansätze gibt, die für sich beanspruchen, PROLOG zu analysieren, sind die meisten dieser Ansätze auf ein kleines Fragment von PROLOG beschränkt, nämlich auf sogenannte definite Programme. Diese Ansätze lassen dabei die Auswirkungen komplizierterer Sprachkonstrukte außer Acht, wie zum Beispiel die Auswirkungen des Cuts, arithmetischer Vergleiche und arithmetischer Auswertungen.

Wir entwickeln eine abstrakte Semantik, die uns erlaubt, Mengen von Ausführungen zu analysieren. Die Objekte dieser Semantik sind abstrakte Programmgraphen, die alle möglichen Evaluationen einer Anfrage an das Programm darstellen. Diese abstrakte, graphbasierte Semantik basiert auf einem existierenden Ansatz, den wir erweitern, um arithmetische Vergleiche und Auswertungen zu berücksichtigen.

Daraufhin konstruieren wir ein Integer Transitionssystem auf Grundlage des abstrakten Programmgraphen, dessen Terminierung die Terminierung des ursprünglichen Programms impliziert. Die Terminierung des Transitionssystems kann dann mit bekannten Methoden analysiert werden.

Wir haben unseren Ansatz in dem Terminierungsanalysetool APROVE implementiert. Mithilfe dieser Implementierung haben wir Experimente auf Grundlage einer großen Menge von Beispielen durchgeführt, die zeigen, dass unser Ansatz bessere Ergebnisse als bestehende Methoden liefert. Diese Verbesserungen zeigen sich sowohl in einer niedrigeren Laufzeit als auch in einer größeren Mächtigkeit.

Acknowledgments

First and foremost, I would like to thank my supervisors, Prof. Jürgen Giesl and Prof. Gerhard Lakemeyer for allowing me to write this thesis under their supervision. This goes as well for Thomas Ströder, without whose fast responses to emails, countless short and long talks as well as tireless proofreading I would not have been able to complete this thesis.

I also have to thank all the members of the APROVE team, both permanent and student workers, who always had the right answers when I needed them, both during long group discussions and to quick questions at a moment's notice.

The comparison between the approach presented in this thesis and others would not have been possible without the help of several people around the world and their willingness to dig deep into backups in order to extract copies of decades-old tools. Namely, these are, in alphabetical order, Samir Genaim, Enno Ohlebusch, Peter Schneider-Kamp, Danny De Schreye, Alexander Sczyrba, Alexander Serebrenik, René Thiemann, and Jan Krüger.

I also want to thank my friends who have supported me during not only the months it took me to write this thesis, but all throughout my studies.

Finally, I would like to thank my parents and my girlfriend, who have supported me tirelessly all throughout my studies and far earlier than that.

Alexander Weinert

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe sowie Zitate kenntlich gemacht habe.

Alexander Weinert

Contents

1. Introduction **1**
- 1.1. Related Work . . . 2
- 1.2. Contributions . . . 3
- 1.3. Mathematical Preliminaries . . . 4

2. Prolog Fragment **5**
- 2.1. Introduction to Logic Programming . . . 5
- 2.2. Syntax and Informal Semantics of PROLOG Fragment . . . 8
- 2.3. Differences between PROLOG Fragment and ISO-PROLOG . . . 13

3. Concrete Semantics **17**
- 3.1. Programs . . . 17
- 3.2. States and Evaluation . . . 19
- 3.3. Equivalence to ISO Semantics . . . 29

4. Abstract Semantics **31**
- 4.1. The Abstract Domain . . . 31
  - 4.1.1. Arithmetic States . . . 31
  - 4.1.2. Abstract Program States . . . 33
- 4.2. Structure of Abstract Evaluation . . . 36
- 4.3. Evaluation of Logic Programs . . . 37
- 4.4. Evaluation of Arithmetic Logic Programs . . . 42
  - 4.4.1. Safe Evaluation of Arithmetic Expressions . . . 42
  - 4.4.2. Evaluation of Arithmetic Comparison . . . 47
  - 4.4.3. Evaluation of Arithmetic Assignment . . . 54
- 4.5. Properties of the Abstract Evaluation Relation . . . 60
  - 4.5.1. Determinacy . . . 61
  - 4.5.2. Soundness . . . 64

5. Termination Analysis **71**
- 5.1. Termination Graphs . . . 72
  - 5.1.1. Abstract Semantics . . . 72
  - 5.1.2. Instantiation and Generalization . . . 76
  - 5.1.3. Splitting and Parallelization . . . 81
  - 5.1.4. Construction of Termination Graphs . . . 88
- 5.2. Transformation of Termination Graphs into Integer Transition Systems . . 91
  - 5.2.1. Integer Transition Systems . . . 92

5.2.2. Reduction of Termination Graphs to ITSs 96
5.2.3. Soundness of the Reduction 104
5.2.4. Termination Analysis of ITSs 105
5.3. Practical Implementation 106

6. Evaluation **109**
6.1. Compared Approaches 109
6.1.1. Strategies Implemented in AProVE 110
6.1.2. Other Tools for Termination 110
6.2. Types of Examples 112
6.3. Discussion of Results 113
6.3.1. Performance on Logic Examples with Cut 113
6.3.2. Performance on Arithmetic Examples 114
6.4. Combining the Approaches 117
6.5. Advantages and Limitations 117

7. Conclusion **119**
7.1. Future Work 120

A. Supplementary Proofs **123**

Bibliography **133**

1. Introduction

The automated analysis of programs has been of interest since the very beginnings of computer science. This interest goes back to [Tur36], which is widely regarded to be one of the first works of computer science. In this work, the author does not only define the Turing machine as a model of computation, but also discusses the possibility of automated analyses. However, he also shows that even the most basic analysis is impossible in general. He furthermore shows that the question whether or not a given program eventually terminates is impossible to answer automatically and correctly for all programs. This problem is commonly known as the halting problem. The corresponding analysis is known as termination analysis.

Despite this early proof, many sound, but necessarily incomplete analyses have been developed. One of the most popular methods for such an analysis is symbolic execution. This technique relies on the fact that computations are usually comprised of a sequence of states. The general idea of symbolic execution is to develop a finite representation of an infinite set of states. If these so-called abstract states can then be evaluated using an abstract semantics, it is possible to reason about an infinite set of concrete states. Even though this approach leads to an overapproximation of the behavior of the program, lots of valuable analyses have been carried out using this method.

In this work we develop an abstract semantics and a termination analysis for programs written in a fragment of the programming language Prolog. For this we use techniques from abstract interpretation. This poses an interesting problem, since the behavior of Prolog programs is not defined as a linear sequence of states, but via a search through a tree. Each program together with some input defines a search tree, which is then traversed at runtime.

We begin with an introduction to the fragment of Prolog that we consider in Section 2 *(*Prolog *Fragment)*. In that section we also describe the standard semantics of this fragment informally and give a pointer to their complete formal definition in an ISO standard.

In order to be able to apply standard methods from abstract analysis to Prolog, we then present a state-based semantics for our fragment, which is shown to be equivalent to the standard semantics. This work, which was originally introduced in [SESK$^+$12] and is merely reproduced in this thesis, is shown in Section 3 *(Concrete Semantics)*.

We then go on to develop an abstract semantics based on this concrete semantics in Section 4 *(Abstract Semantics)*. This abstract semantics extends work previously published in [GSSK$^+$12] and defines both abstract states and rules for evaluating these states. We extend the definition of an abstract state shown in [GSSK$^+$12] and define new rules that allow us to evaluate built-in features of Prolog that were not previously considered. Furthermore, we show that this abstract semantics is a sound overapproximation of the concrete semantics described in Section 3.

In Section 5 *(Termination Analysis)* we develop a termination analysis on the basis of the abstract semantics given in Section 4. For this we first construct a finite termination graph from a given program. This construction was previously implicitly described in [GSSK$^+$12] and part of the abstract semantics presented there. We then construct an integer transition system from the termination graph. We show that termination of the integer transition system implies termination of the original program.

We have implemented our termination analysis in the termination prover APROVE. Using this implementation, we have evaluated our approach to termination analysis in Section 6 *(Evaluation)*. Our termination analysis significantly outperforms existing approaches to termination analysis of arithmetic programs both in terms of power and in terms of runtime. It also is, to the best of our knowledge, the first termination analysis for PROLOG that handles both arithmetic comparisons and evaluations as well as the cut directly.

The thesis closes with a conclusion and summary of our findings in Section 7 *(Conclusion)*. In that section, we also give an outlook on further work that can build on this thesis.

1.1. Related Work

Termination analysis as well as the analysis of PROLOG and symbolic execution in general have a long history. Even though termination analysis is impossible as a decision procedure, there have been both theoretical methods and practical implementations that tackle this problem. In this section we give a brief overview over these techniques and provide pointers to literature on related topics.

Our work builds directly on previous work towards the termination analysis of PROLOG, namely work published in [GSSK$^+$12], which presented an abstract semantics for PROLOG, and [SESK$^+$12], in which the state-based concrete semantics of PROLOG were defined.

[CC77] is often cited as the seminal work on abstract interpretation, which is a framework for program analysis that is strongly connected to symbolic evaluation. It introduces this method and provides examples of analyses that can be formulated in terms of abstract interpretation. A more current overview of abstract interpretation given by the same authors can be found in [CC14].

Termination analyses of PROLOG programs differ along two major axes: Which language constructs they support and whether they follow a direct or a transformational approach to showing termination. Many works are only concerned with logic programming. This is a subset of actual PROLOG, in which only the logic capabilities are used. Thus, logic programs do not use more complex features of PROLOG such as the cut or arithmetic predicates.

The termination of logic programs is investigated, among others, in [NDS07, OCM00, SKGST09, SSKG11, Sma04, LMS03, BCG$^+$07, MB05, SDS03]. The nontermination of such programs is treated in [PM06]. These approaches do not support arithmetic predicates or the cut.

Non-termination analysis for arithmetic programs was presented in [VDS11]. Other methods for analyzing the termination of logic programs with cut are shown in [SKGN10, SSKG11, GSSK$^+$12].

A concrete semantics for Prolog including the cut has most recently been presented in [KK14]. The authors state that this semantics is amenable to abstract interpretation. To the best of our knowledge, however, no abstract semantics has been defined using this concrete semantics.

[SLH14] presents a more general analysis, in which the resource usage of a logic program is analyzed using abstract interpretation. Termination analysis is a special case of such an analysis, in which the only question is whether or not the program consumes a finite amount of the resource time. However, this resource analysis only operates on logic programs and does not take the cut into account. Arithmetic predicates are recognized, but not treated specially.

The state of the art in termination analysis is presented yearly at the International Termination Competition[1]. This competition featured categories for logic programming with and without cut as well as full Prolog until 2013.

To the best of our knowledge, there exists no termination analysis that handles a fragment of Prolog that includes both arithmetic comparisons and evaluations as well as the cut.

We use procedures that decide the termination problem for integer transition systems as a backend. The techniques used in the implementation are described in [FGP$^+$09], which builds on [PR04].

1.2. Contributions

In this section we provide a list of our contributions both on a theoretical and a practical level. We reference the relevant parts of this thesis for the theoretical contributions and name our practical contributions. Our contributions are as follows:

Abstract States We extended the definition of abstract states originally provided in [GSSK$^+$12] in order to carry information about the arithmetic state of variables. This extension is detailed in Section 4.1.2 *(Abstract Program States)*.

Separation of Semantics and Analysis We separated an abstract semantics for Prolog from the termination analysis using this semantics. This separation is evident in the division between Section 4 *(Abstract Semantics)* and Section 5 *(Termination Analysis)*. The combination of semantics and termination analysis was originally published in [GSSK$^+$12].

Extension of Semantics We extended the semantic rules used for the abstract evaluation of states in [GSSK$^+$12] with rules for handling arithmetic comparisons and evaluations. These additional rules are defined in Section 4.4.2 *(Evaluation of Arithmetic Comparison)* and Section 4.4.3 *(Evaluation of Arithmetic Assignment)*.

[1]`http://termination-portal.org/wiki/Termination_Competition`

Termination Graphs We extended the construction of termination graphs that was originally given in [GSSK+12]. We added handling of arithmetic comparisons and evaluations to this construction. The complete construction is detailed in Section 5.1.4 *(Construction of Termination Graphs).*

Integer Transition Systems We developed a new construction of integer transition systems from termination graphs. This construction is sound with regards to termination. The complete construction can be found in Section 5.2.2 *(Reduction of Termination Graphs to ITSs).*

Soundness Proofs We showed the soundness of both the abstract semantics and the termination analysis. The former is shown in Lemma 4.8. The latter is argued in Theorem 5.1.

Implementation of Transformation to Termination Graphs We implemented the aforementioned extension of the construction of termination graphs from programs in the termination prover APROVE. We reused parts of the implementation of the earlier termination analysis from [GSSK+12]. This contribution amounts to approximately 1250 lines of code added or changed in APROVE.

Implementation of Transformation to Integer Transition Systems In addition to the implementation of the transformation to termination graphs, we implemented the construction of integer transition systems from termination graphs in the termination prover APROVE. This transformation was implemented from scratch, using the existing termination analyses of APROVE as a backend. This contribution amounts to approximately 1550 lines of code added to APROVE.

Arithmetic Benchmarks We added 162 numerical benchmarks to the PROLOG benchmark suite of APROVE, 156 of which were added to the termination problem database[2]. A breakdown of the kinds of the examples is given in Section 6.2 *(Types of Examples).*

Experimental evaluation We evaluated our approach in comparison to four other methods for showing (non-)termination of PROLOG programs. The results of this evaluation are presented in Section 6.3 *(Discussion of Results).*

1.3. Mathematical Preliminaries

This work assumes that the reader has a general understanding of graduate-level mathematics and its standard notation. In Section 5 we furthermore assume the reader to know basic concepts and terminology of graph theory.

Throughout this work we regularly lift functions to sets and sequences without further mention. This lifting occurs pointwise, by applying the function to all members of the set or sequence, while retaining the order of elements in the case of the sequence.

Finally, we use the notation $f\big|_S$ to restrict the domain of the function f to the set S. The function $f\big|_S$ returns $f(s)$ for all $s \in S$ and is undefined on all elements not in S.

[2] `http://termination-portal.org/wiki/TPDB`

2. Prolog Fragment

The goal of this chapter is to introduce the programming language that we are going to consider in the remainder of this thesis. This language is a fragment of the full-featured declarative logic programming language Prolog.

We first give a brief introduction to logic programming in general in Section 2.1. Afterwards, we discuss the considered fragment of Prolog in Section 2.2 and give an informal explanation of its semantics. In Section 2.3, we highlight the differences between the fragment of Prolog that we consider and the complete language that is specified in [ISO95].

The goal of this section is merely to give a very brief introduction to logic programming and a fragment of Prolog. For a more thorough tutorial on Prolog, please refer to [SS86], which is widely regarded as the seminal tutorial for this language.

2.1. Introduction to Logic Programming

In this section we give a brief intuitive introduction to logic programming in general. For a formal model of programs written in this language, please refer to Section 3.1 *(Programs)*.

Logic programming is a programming paradigm that is vastly different from more well-known paradigms, such as imperative, functional, or object-oriented programming. The basic building blocks of an imperative program, for example, are statements that describe the computation to be carried out by the program. A program consists of a sequence of such statements that are translated directly to executable code.

Logic programming, in contrast, is declarative in nature. Instead of specifying the individual steps that produce output for a given input, the program contains rules that describe the output in relation to the input. The program is then executed by some runtime that infers the output using the rules present in the program.

As a leading example, we are going to write a program that allows the user to pose queries about the relationships in some family. This program is going to contain only the most basic facts about the structure of a specific family as well as general rules that allow to deduce more complicated relationships between members of the family.

The basic building block of logic programming is a **term**. Any term is either a **variable**, or it consists of a **function symbol** and a sequence of arguments. Every function symbol requires a fixed number of arguments, which we call its **arity**. The arguments are again terms.

We write a term consisting of the function symbol `f` and the arguments $\mathtt{t_1}$ through $\mathtt{t_n}$ as $\mathtt{f(t_1,...,t_n)}$. If `f` has arity 0, then we omit the brackets and simply write `f`.

In order to differentiate between function symbols of arity 0 and variables, we write function symbols with a lowercase letter at the beginning, whereas variables start with an uppercase letter from now on.

Example 2.1 (Terms)**.** We pick `father`, `female`, and `alice` as function symbols, where we let `father` have an arity of 2, we let `female` have an arity of 1, and we let `alice` have an arity of 0. Using this signature, both `father(female(alice),alice)` and `father(female(father(X,alice)),Y)` are terms, with `X` and `Y` being variables in the latter case. The expression `father(alice)`, however, is not a term, as `father` has an arity of 2, but only a single argument, namely `alice`. For the same reason, `father(X)` is not a term.

The intuitive meaning of these function symbols is that we, for example, would write `father(X, alice)` to denote that `X` is the father of `alice`. We could also write `female(alice)` to denote that `alice` is female. ▲

If a term contains no variables, we call it **ground**. We also call a function symbol of arity 0 a **constant**. Instead of saying "`f` is a function symbol with arity n", we write "the function symbol `f/n`." In logic programming, terms are used to define **predicates**. Since a formally precise distinction between terms and predicates would not serve the goal of this thesis, we do not distinguish strongly between the two for the remainder of this thesis.

Terms are the single data structure supported by logic programs. A logic program assigns a truth-value to them. The "meaning" of an imperative program, in contrast, is essentially defined as a transformation of the input values.

Any ground term is either true or false, depending on the program that is used to interpret it. Whether a term is true or false with respect to some program is defined in that program through **rules** and **facts**.

A logic program consists of a sequence of rules and facts. These rules and facts define which terms are true and which are not. An execution of a logic program starts with a term that may or may not contain variables. If the term does not contain variables, the goal of the execution is to infer whether or not the term holds true. If, on the other hand, the term does contain variables, then the goal of the execution is to find assignments of ground terms to the variables such that the resulting term holds true.

Facts are nothing but terms. Rules are a pair of a term and a non-empty sequence of terms, which we call the **head** and the **body** of the rule, respectively.

Example 2.2 (Rules and facts)**.** We pick the set of function symbols as in Example 2.1 and extend it with the constant `bob`. Both the term `father(bob, alice).` and the term `father(female(father(X,alice)),Y).` are facts. The pair of terms ((`father(X, Y)`), (`female(X), female(Z)`)) is a rule. The pair (`female(X)`, ϵ), however, is not a rule, since the body is empty. ▲

For a term `t`, we write `t.` in order to emphasize `t`'s role as a fact. We write a rule with the head `t` and the body $\mathtt{t_1}$ through $\mathtt{t_n}$ as $\mathtt{t\ :-\ t_1,\ldots,t_n.}$ We also call a (possibly empty) sequence of terms a **goal**.

A rule can be interpreted as an implication from right to left. The sequence of terms in the body is interpreted as a conjunction of the terms. Thus, the intuitive meaning of a rule is that the term at the head holds true if all of the terms in the body hold true. The variables in the head of a rule are universally quantified, whereas the variables that occur only in the body of a rule are existentially quantified.

Example 2.3 (Meaning of a rule)**.** For this example, we extend the set of function symbols from Example 2.2 with `grandfather/2`.

The rule `grandfather(X, Y) :- father(X, Z), father(Z, Y).` has the following meaning in plain English:

> For all terms `X` and `Y`, the following holds true: If there exists a term `Z`, such that both `father(X, Z)` and `father(Z, Y)` hold true, then `grandfather(X, Y)` holds true.

The rule `father(X, Y) :- female(X),female(Z).`, on the other hand, has the following meaning in plain English:

> For all terms `X` and `Y`, the following holds true: If there exists a term `Z`, such that both `female(X)` and `female(Z)` hold true, then `father(X, Y)` holds true.

Note that in both examples `X` and `Y` are universally quantified, whereas `Z` is existentially quantified.

We see that, using such a rule, we can define the truth-value of terms both according to their intuitive meaning as well as in contradiction thereto. The truth-value of terms depends solely on the rules given in the program, but not on any natural interpretation. ▲

An execution of a logic program starts with a term, which is called the **query**. The behavior of the program differs slightly depending on whether or not the query is ground. If it is ground, then the program tries to infer whether or not the query holds true using the rules and facts given in the program. If the query contains variables, however, the runtime tries to find assignments of ground terms to variables such that the query is true.

The choice of the algorithm that is used for this search is one of the major points in which programming languages using the logic paradigm differ. For the next example, we assume that there are sound and complete algorithms to find the truth-value of terms and assignments to variables such that a term is true.

Example 2.4 (A logic program)**.** Consider Program 2.1. This program uses the function symbols from Example 2.2 as well as the additional function symbols `male/1`, `ancestor/2`, and the constants `claire` and `diane`.

This program contains the facts that `alice`, `claire` and `diane` are female and that `bob` is male. It furthermore contains the facts that `bob` is `alice`'s father and the fact that `diane` is `claire`'s mother. The rules of this program state that there are two possible

Program 2.1 Family relationships

```
female(alice).                        male(bob).
female(claire).                       female(diane).
father(bob, alice).                   mother(claire, bob).
mother(diane, claire).
ancestor(X, Y) :- mother(X, Y).
ancestor(X, Y) :- ancestor(X, Z), mother(Z, Y).
```

ways for `X` to be an ancestor of `Y`. Either it holds true that `X` is the mother of `Y`, or there must exist some `Z` such that `X` is an ancestor of `Z` and `Z` is the mother of `Y`. Thus, we define the ancestor relation as containing only the maternal ancestors in this example.

Assume we pose the query `male(bob)` to the program. It would be able to infer that this term is true, since `male(bob)` is stated as a fact in the program. It would, however, not be able to infer that `ancestor(claire, alice)` is true. This term is neither stated as a fact in the program, nor is it derivable via the given rules for the function symbol `ancestor`. The program would, in contrast, be able to infer that `ancestor(diane, bob)` holds true.

If we posed the query `ancestor(X, bob)` to the program, it would try to find assignments of terms to `X` such that the query holds true. There are two such assignments to `X`, namely `X` $\mapsto$ `claire` and `X` $\mapsto$ `diane`. ▲

Logic programming is only one of many paradigms that a programming language can follow. As such, it is better to think of logic programming as a principle, rather than as a concrete programming language. In particular, the logic programming paradigm states nothing about the algorithm that is used to infer the truth value of terms from a program. It also does not support arithmetic comparisons or evaluations. Function symbols such as `</2`, `=/2` or `+/2` are undefined as well as constants such as `17` or `-3.8`. Finally it is not possible for the program to take any influence on the inference that is performed on it in pure logic programming.

In this work we consider a fragment of the logic programming language Prolog. The complete Prolog language defines an algorithm for the inference of the truth values of terms and it contains built-in predicates. In the next section we discuss this fragment of Prolog.

2.2. Syntax and Informal Semantics of Prolog Fragment

In this section we discuss the fragment of Prolog that we consider for the remainder of the thesis. First we explain how Prolog infers satisfying assignments of terms to variables and the truth value of variable-free terms. We then continue to explain those built-in predicates of Prolog that we include in our fragment, namely the cut and predicates for arithmetic comparison and assignment.

We give an intuitive explanation of the semantics of this fragment according to [ISO95]. For a more precise definition of this semantics, please refer to that work or to [DEDC96].

We also provide a formal semantics of this fragment in Section 3 *(Concrete Semantics)*, which is equivalent to the "official" semantics from [ISO95].

In order to describe the meaning of a PROLOG program, we first need to describe the notion of unification. Unification is a partial operation that takes two terms and returns a substitution of variables by terms. The idea is that two terms unify if the variables in these two terms can be replaced in such a way that both terms are the same with respect to the replacement. We call the result of the unification a **unifier**.

Example 2.5 (Unification)**.** Consider the two terms `male(bob)` and `male(X)`. These terms unify with the unifier ($\mathtt{X} \mapsto \mathtt{bob}$). This is the only unifier that unifies these two terms. The two terms `father(X, alice)` and `father(bob, Y)` unify as well with the unifier ($\mathtt{X} \mapsto \mathtt{bob}, \mathtt{Y} \mapsto \mathtt{alice}$). The terms `female(X)` and `male(Y)` do not unify, however, due to the differing function symbols `female/1` and `male/1`. ▲

A precise definition of unifiers can be found in Section 3.2. If two terms $\mathtt{t}_1$ and $\mathtt{t}_2$ unify with some substitution σ, we say that $\mathtt{t}_1$ and $\mathtt{t}_2$ **unify with the unifier** σ. In that case it holds that $\sigma(\mathtt{t}_1)$ is the same as $\sigma(\mathtt{t}_2)$.

Now assume that we have two terms $\mathtt{t}_1$ and $\mathtt{t}_2$ and two unifiers σ_1 and σ_2, which are not equal. We call σ_1 **more general** than σ_2 if we can find another non-trivial substitution γ, such that $\gamma(\sigma_1(\mathtt{t}_1))$ is the same as $\sigma_2(\mathtt{t}_1)$. The **most general unifier** of two terms is a unifier for which no other, more general unifier exists.

Example 2.6 (Most general unifier)**.** The terms `mother(X, alice)` and `mother(Y, alice)` unify, for example, with the unifiers ($\mathtt{X} \mapsto \mathtt{Z}, \mathtt{Y} \mapsto \mathtt{Z}$) and ($\mathtt{X} \mapsto \mathtt{female(Z)}, \mathtt{Y} \mapsto \mathtt{female(Z)}$). The former substitution is the most general unifier of the two terms. ▲

For simplicity, we always assume that the unifier always uses fresh variables that occur in neither of the two terms that are unified. This simplifies reasoning about the unification. It does not reduce the generality of unification, as it can be achieved by a simple renaming of the unified term.

PROLOG follows the paradigm of logic programming in that programs written in this language consist of facts and rules. Both facts and rules are written as they are in the intuitive introduction to logic programming in the preceding section. In order to evaluate a query, PROLOG explores a tree via depth-first search.

The root of this tree consists of the query given at the start of the program. The children of this node are all the possible ways to show that the query holds true. Since the truth of the query is defined via the facts and rules, the node has one child for each fact that starts with the same function symbol as the query as well as one child for each rule whose head starts with the same function symbol as the query does. These child nodes are then evaluated depth-first similar to how the root node was evaluated.

Every node of this tree is a goal that PROLOG tries to show to be true. The children of a given node are defined depending on the function symbol at the head of the first term of the goal. This function symbol may either be a built-in one, which we will cover later in this section, or it may be a user-defined one, such as `male/1` in Example 2.4.

If the function symbol is user-defined, then all facts and rules that start with the function symbol are collected. In the case that a fact unifies with the first term of a goal with the unifier σ, then this term is removed and σ is applied to the remaining goal. If, however, the head of a rule unifies with the first predicate of a goal with the unifier σ, this predicate is replaced by the body of the rule after application of σ. The substitution σ is applied to the remainder of the goal as well.

If the function symbol is built-in, it is evaluated to true or false depending on the rules corresponding to this predicate. If it evaluates to true, then the term is removed from the goal and the evaluation continues with the remainder of the goal. Otherwise, the current branch is abandoned. We will explain the rules for these function symbols later in this section.

Example 2.7 (Inference in PROLOG)**.** Consider Program 2.1 and the query `ancestor(X, bob)`. This query induces the search tree shown in Figure 2.1. The nodes are numbered in the order that they are explored in.

There are two ways that PROLOG can infer the truth of the query. The first way is by traversing states 1, 2, and 3 in this order. `X` is replaced with `claire` along the transition from 2 to 3. The other way is by traversing states 1, 4, 5, and 7. Along this path, `X` is instantiated to `diane` at the transition from state 5 to 7. Thus, PROLOG returns the two answers `X = claire` and `X = diane`, just as described in Example 2.4.

State 6 has no successors, since there is no fact or head of a rule which unifies with `mother(bob, bob)`. This tree is infinite since PROLOG always tries to use both `ancestor`-rules to find new answers to the query. Since the second rule is defined recursively, PROLOG applies this rule infinitely often and the inference algorithm never terminates. ▲

PROLOG also provides pre-defined arithmetic predicates and even arithmetic functions. The presence of functions is a slight deviation from logic programming, but makes the resulting language easier to use. An arithmetic comparison takes the form $\mathtt{e}_1 \bowtie \mathtt{e}_2$, where $\bowtie$ is one of `=:=`, `=\=`, `<`, `>`, `=<`, and `>=`, while $\mathtt{e}_1$ and $\mathtt{e}_2$ denote arithmetic expressions. Arithmetic expressions may include variables and integer literals as well as the unary and binary arithmetic operators `abs`, `sign`, `+`, `-`, `*`, `//`, `**`, `mod`, and `rem`. When the inference algorithm encounters such a comparison, it tries to evaluate the expression on both sides of the comparison and compares them according to the specified comparison operator. There are three possible outcomes of this comparison.

Evaluation Error It may be the case that the evaluation of the expression on either side of the comparison results in an error. This is possible, for example, via a division by zero, or if a sub-expression is an uninstantiated variable or user-defined term instead of a built-in function. In this case, PROLOG raises one of several exceptions [DEDC96, Section 6.2.1]. Since our fragment of PROLOG does not support exception handling, this leads to undefined behavior. In our formalization, we simply abort the inference in this case.

Success If both expressions successfully evaluate to integers, then the resulting integers are compared according to the comparison specified by the operator. If this comparison

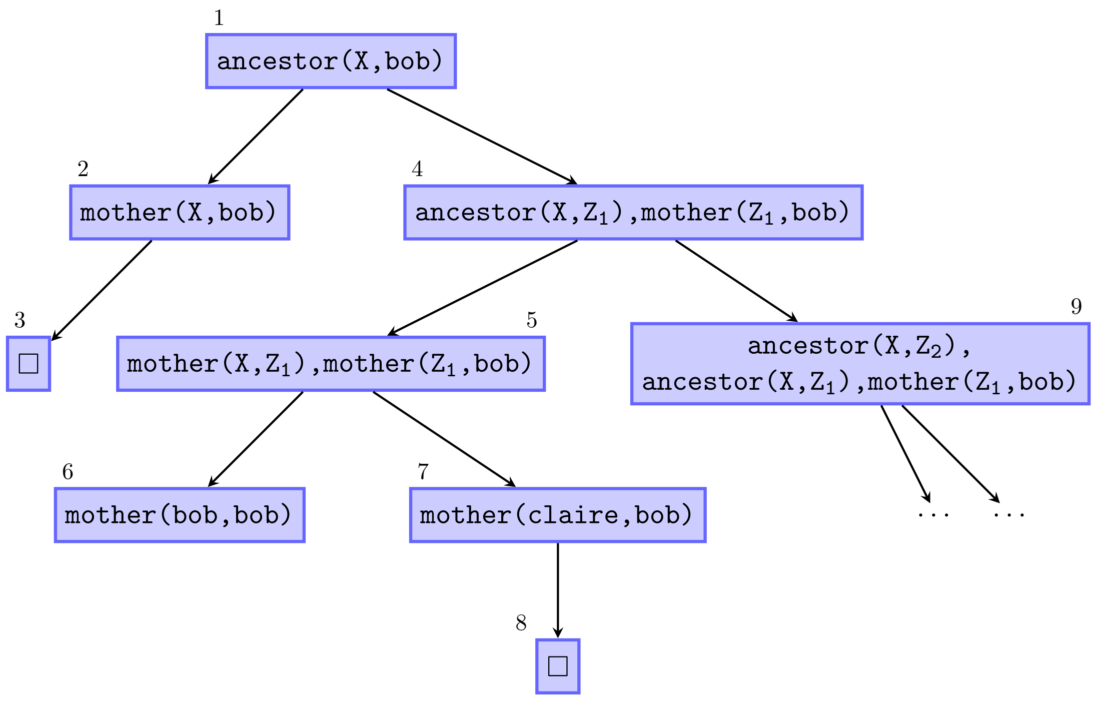

1
ancestor(X,bob)
2
mother(X,bob)
4
ancestor(X,Z1),mother(Z1,bob)
3
□
5
mother(X,Z1),mother(Z1,bob)
9
ancestor(X,Z2), ancestor(X,Z1),mother(Z1,bob)
6
mother(bob,bob)
7
mother(claire,bob)
...
...
8
□

Figure 2.1.: The search tree induced by `ancestor(`X`, bob)` on Program 2.1

evaluates to true, then the whole term is treated like a fact, removed from the current goal and evaluation continues with the remaining terms.

Failure If there is no error during the evaluation of the expressions on either side of the comparison and the comparison between the resulting integers evaluates to false, then the comparison is treated like a term for which no rules or facts are defined. The inference algorithm discards the branch and continues the search in the next one.

There is also the pre-defined function symbol `is`, which is used to unify integers with terms. Terms using the `is`-symbol are of the form t `is e`. Similar to the evaluation of expressions for arithmetic comparison, the expression `e` is evaluated to an integer. If this evaluation results in an error, the behavior is the same as during the evaluation of an expression for an comparison, that is, the inference is aborted and the program terminates. If the evaluation succeeds and the expression evaluates to some integer n, it is checked whether t and n unify. If this is the case, then the substitution $(t \mapsto n)$ is applied to the remaining terms and the inference continues with the remaining goal. Otherwise, the search abandons the current branch of the search tree and continues in the next one.

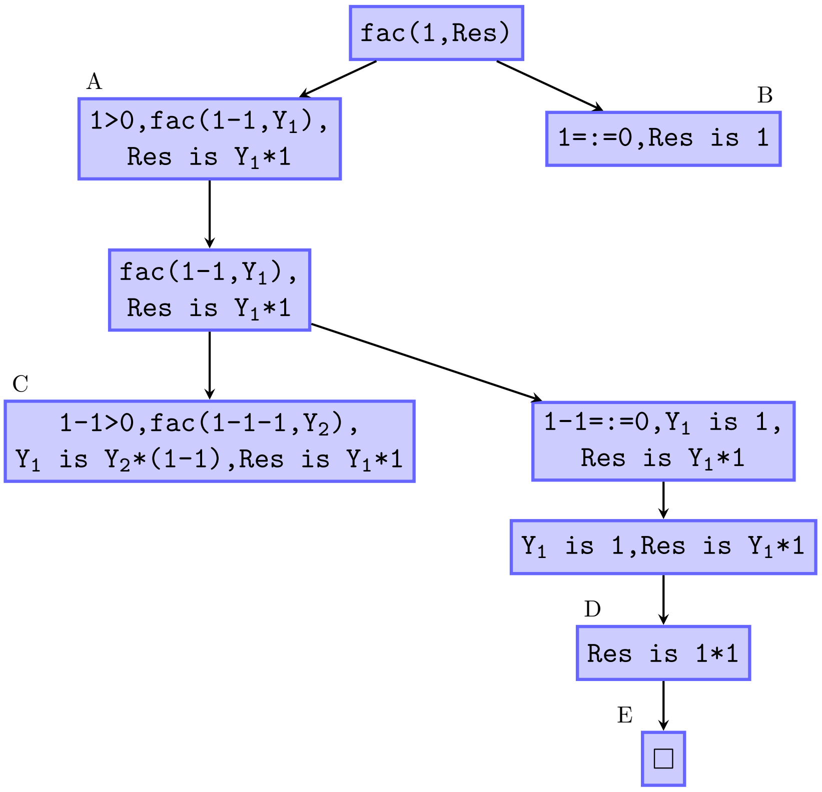
Example 2.8 (Arithmetic in PROLOG)**.** Program 2.2 defines the computation of the factorial of a number. The function symbol `fac/2` is defined such that the term `fac(X,`

Y) holds true if Y is the factorial of X. We show the search tree for the query fac(1, Res) in Figure 2.2.

Neither node B nor node C have any children, since both comparisons 1=:=0 and 1-1>0 are obviously false. The only succeeding instantiation that is found is the one found on the path to node E. Along this path, Res is instantiated to the result of 1*1 along the transition from D to E. Thus, Prolog reports that the single result is Res = 1, as was expected. ▲

Program 2.2 Computation of the factorial

```
fac(X, Y) :- X > 0, fac(X - 1, Y1), Y is Y1 * X.
fac(X, Y) :- X =:= 0, Y is 1.
```



fac(1,Res)
A
1>0,fac(1-1,Y₁), Res is Y₁*1
B
1=:=0,Res is 1
fac(1-1,Y₁), Res is Y₁*1
C
1-1>0,fac(1-1-1,Y₂), Y₁ is Y₂*(1-1),Res is Y₁*1
1-1=:=0,Y₁ is 1, Res is Y₁*1
Y₁ is 1,Res is Y₁*1
D
Res is 1*1
E
□

Figure 2.2.: The search tree induced by fac(1, X) on Program 2.2

In the previous example, after exploring the path leading to node E, Prolog backtracks through the tree and tries to show that the goal in node B holds true. However, since there are children of node A, the programmer knows that the evaluation of the first term of the goal in node A must have succeeded. Hence, we must have shown that 1 > 0 holds true. Thus, the programmer knows that 1 =:= 0 cannot hold true. Prolog does not have that knowledge and thus has to spend time to show that 1 =:= 0 does not hold. This time is negligible in this case, but may impact performance significantly in more complicated programs.

In order to address this, PROLOG has a feature called the cut, which is written as `!`. This symbol can occur in the place of a term in the right-hand side of a rule. When the interpreter reaches the cut, it prunes the current search tree. It does so by cutting off all alternative choices it had for showing the truth of the last term for which it tried out rules. Reaching the cut does not, however, cut off all unexplored branches, but only the most recently produced ones.

Example 2.9 (The cut)**.** Program 2.3 shows a program that computes the factorial of a number using the cut. Consider the first rule. After PROLOG succeeds to show that the fact `X > 0` holds true for some instantiation of `X`, it reaches the cut and will not try to backtrack to show that `X =:= 0` holds true. The search tree induced by the query `fac(1, Res)` on Program 2.3 is shown in Figure 2.3. The branch that is not searched due to the evaluation of the cut is drawn gray.

Program 2.3 Computation of the factorial with cut

```
fac(X, Y) :- X > 0, !, fac(X - 1, Y1), Y is Y1 * X.
fac(X, Y) :- X =:= 0, Y is 1.
```

Program 2.4 shows that the cut only removes the most recently produced unexplored branches. The search tree induced by the query `f(X)` on this program is shown in figure 2.4. The branches not explored due to reaching the cut are drawn in gray. Evaluating the cut in node D only cuts off node E, since the last evaluation of a user-defined function symbol at this point happened during evaluation of node B. There was another choice of rules when evaluating node A. However, node C, which has not yet been explored upon reaching the cut, is not cut off. This shows that only the last choice of rules is cut off as a result of evaluating the cut.

Program 2.4 Cutting branches on the same level

```
f(X) :- g(a).                       f(X) :- h(b).
g(X) :- !, h(a).                    g(X) :- h(b).
h(b).
```

▲

While we only give an informal explanation of the behavior of the cut at this point, we define it formally later on in Section 3.2.

2.3. Differences between Prolog Fragment and ISO-Prolog

In the previous section we have discussed the programming language that we are going to consider for the remainder of this work. This language is a fragment of the full featured PROLOG language, which is also called ISO-PROLOG. Its official definition can be found

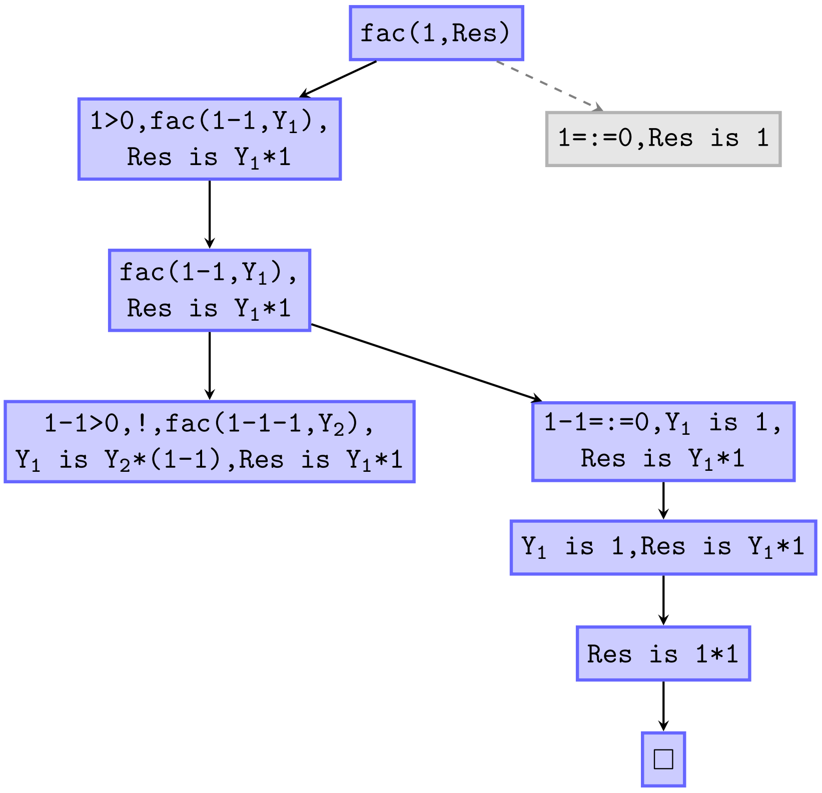

fac(1,Res)
1>0,fac(1-1,Y1), Res is Y1*1
1=:=0,Res is 1
fac(1-1,Y1), Res is Y1*1
1-1>0,!,fac(1-1-1,Y2), Y1 is Y2*(1-1),Res is Y1*1
1-1=:=0,Y1 is 1, Res is Y1*1
Y1 is 1,Res is Y1*1
Res is 1*1
□

Figure 2.3.: The search tree induced by `fac(1,` X`)` on Program 2.3

in [ISO95]. Since we only consider a fragment of the full language, there are some features of ISO-Prolog that are missing from our fragment. In this section we give a brief overview over the differences between our fragment and the complete language.

One major feature of ISO-Prolog that our fragment does not support is the handling of exceptions. In the full language, it is possible to raise and handle exceptions using the built-in predicates `throw/1` and `catch/3`, which behave similarly to the keywords of the same name from imperative languages. In addition to exceptions thrown by the user using the `throw/1`-predicate, an exception may also be raised due to an error during the search for answer substitutions. In particular, this may happen if there is an error during the evaluation of an arithmetic expression, such as division by zero or if a variable occurs as a subexpression. In this case, one of several exceptions is raised by the runtime system, which can only be caught using the `catch/3`-predicate. Since this predicate does not exist in our fragment, we are unable to handle such errors. The exact behavior of ISO-Prolog during the evaluation of arithmetic expressions is defined in [DEDC96, Section 6.2.1].

Another omission from our fragment is the presence of real numbers and manipulations of the underlying representation of numbers. Prolog features full support for both integer and floating point numbers as defined in [ISO94]. In addition to the arithmetic functions present in our fragment, there are also pre-defined operators that work on

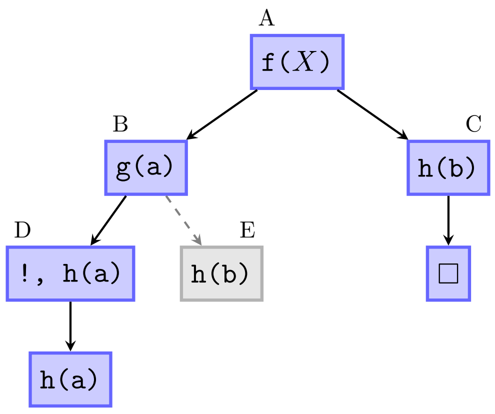


Figure 2.4.: The search tree induced by `f(`X`)` on Program 2.4

floating point numbers, such as floating point division, the natural logarithm or the calculation of the square root. There are also functions that operate on the internal representation of the numbers, such as bitwise conjunction or disjunction and bitwise complementation. All of these features are omitted from our fragment.

Furthermore, there is a set of predicates that modifies the search through the tree during runtime. The most often used one of these is the `\+/1`-predicate, which succeeds if its single argument fails. Some other predicates allow more intricate manipulation of the program at runtime, such as the addition and removal of rules and facts during the inference. These predicates are also not supported by our fragment. It is, however, possible to rewrite the `\+/1`-predicate using the cut. Since our fragment contains the cut, this predicate can be removed from the program in a preprocessing step.

In order to be able to react to user input, Prolog also features a number of predicates that perform input and output, both from and to the user in an interactive way as well as from and to files. Since our goal in this thesis is to develop a semantics that allows for automated and static analysis of the behavior of a program, we also omit these predicates from our fragment.

Finally, there exist some constructs that are mainly included as syntactic sugar, such as the predicates `->/2`, `->;/3`, or `;/2`, which represent if-then, if-then-else and boolean disjunction, respectively. Even though it may be possible in some cases to remove this syntactic sugar through a preprocessing step, this is not the focus of this thesis. We thus assume that programs are written without these predicates.

3. Concrete Semantics

In the previous chapter, we gave an overview over the fragment of Prolog that we are going to consider for the remainder of this thesis as well as an informal explanation of the semantics of this fragment. The goal of this chapter is to formalize these semantics.

We start with a formalization of programs in Section 3.1. In Section 3.2 we give an overview over a state-based formal semantics for our fragment. This semantics was first presented in [SESK^{+}12]. We present the rules that are relevant for the remainder of this work and point to that work for a more thorough explanation. Finally, we formally state the equivalence of this semantics with the ISO-semantics of Prolog in Section 3.3 and refer to [SESK^{+}11] for complete proofs.

3.1. Programs

In Section 2.2 *(Syntax and Informal Semantics of* Prolog *Fragment)*, we gave an informal explanation of the syntax of programs written in our fragment of the language. However, in order to be able to define a formal semantics, we first need a more rigorous formal representation of a program. In this section we define such a representation

As mentioned previously, the most basic unit of a Prolog program is a term. The intuition behind the notion thereof was explained in Section 2.1 *(Introduction to Logic Programming)*. We now proceed to formally define terms. The following definitions are based on the definitions in [Gie11, Section 2.1].

Definition 3.1 (Signature, arity, constant)**.** For all $i \in \mathbb{N}_0$ let Σ_i be some set of symbols. We call $\Sigma := \cup_{i \geq 0} \Sigma_i$ a **signature** if all Σ_i are pairwise disjoint and Σ is finite and nonempty. We call the elements of Σ_i **i-ary** function symbols. Furthermore, we call the members of Σ_0 **constants**. ■

As previously stated in our informal explanation of terms, a term is either a variable, or it consists of a function symbol and as many arguments as the arity of the function symbol requires.

Definition 3.2 (Variable, term, subterm)**.** Let Σ be a signature and let $\mathcal{V}$ be a non-empty and countably infinite set of **variables**, such that Σ and $\mathcal{V}$ are disjoint. The set $\mathit{Terms}_{\Sigma,\mathcal{V}}$ is defined as the smallest set for which all of the following conditions hold:

- $\mathcal{V} \subseteq \mathit{Terms}_{\Sigma,\mathcal{V}}$
- $\Sigma_0 \subseteq \mathit{Terms}_{\Sigma,\mathcal{V}}$
- For all $n \in \mathbb{N}$ the statement

$$f \in \Sigma_n \textbf{ and } t_1, \ldots, t_n \in \mathit{Terms}_{\Sigma,\mathcal{V}} \textbf{ implies } f(t_1, \ldots, t_n) \in \mathit{Terms}_{\Sigma,\mathcal{V}}$$

holds true

We call the elements of $\mathit{Terms}_{\Sigma,\mathcal{V}}$ **terms**. For a given term t, we denote the set of variables that appear in t as $\mathcal{V}(t)$.

A term t' is called a **subterm** of another term t if either $t' = t$ holds true or if t is of the form $f(t_1, \ldots, t_n)$ and there is some $i \in [1, n]$ such that t' is a subterm of t_i. ■

In our intuitive explanation, we have made a distinction between rules and facts. Rules consisted of a term called the head and a nonempty sequence of terms called the body, whereas facts consisted of a single term. In order to have a uniform representation of facts and rules, we only consider rules in the remainder of this work. A fact is written as a rule with an empty body. This removes the formal representation further from the actual syntax of a PROLOG program, but makes it easier to treat rules and facts uniformly, as we will see in Section 3 *(Concrete Semantics)*.

Definition 3.3 (Goal, rule)**.** Let Σ be a signature and let $\mathcal{V}$ be a set of variables. We define:

$$\mathit{Goals}_{\Sigma,\mathcal{V}} := \left\{t_1, \ldots, t_n \mid n \in \mathbb{N}_0, t_i \in \mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}}\right\}$$

The elements of $\mathit{Goals}_{\Sigma,\mathcal{V}}$ are called **goals**. We write the empty goal as $\square$.

We also define the set:

$$\mathit{Rules}_{\Sigma,\mathcal{V}} := \mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}} \times \mathit{Goals}_{\Sigma,\mathcal{V}}$$

The elements of $\mathit{Rules}_{\Sigma,\mathcal{V}}$ are called **rules**. If (h, b) is a rule, we also write h :- b. ■

Since a program is nothing but a finite sequence of rules, we can now define a program. We have to consider the built-in predicates that occur in our fragment of PROLOG, which must not be redefined by the user. Hence, they may only occur in the body of a rule, but not in its head. Furthermore, we have to allow the presence of the cut on the right-hand side of any rule.

Definition 3.4 (Signature with built-ins, program)**.** Let $\Sigma = \cup_{i\geq 0}\Sigma_i$ be a signature and let $\mathcal{V}$ be a set of variables. We define the **signature with built-ins** Σ_{BI} as

$$\begin{aligned}\Sigma_{BI} := &(\Sigma_0 \uplus \{!\} \uplus \mathbb{Z})\\ &(\Sigma_1 \uplus \{\mathit{abs}, \mathit{sign}, -\})\\ &(\Sigma_2 \uplus \{+, -, *, //, **, \mathit{mod}, \mathit{rem}\} \uplus \{=:=, =\backslash=, <, >, =<, >=\} \uplus \{\mathit{is}\})\end{aligned}$$

We define $\mathit{Programs}_{\Sigma,\mathcal{V}}$ as

$$\mathit{Programs}_{\Sigma,\mathcal{V}} := \{r_1, \ldots, r_n \mid n \in \mathbb{N}_0, r_i \in \mathit{Rules}_{\Sigma_{BI},\mathcal{V}}, r_i \text{ is valid}\},$$

where, in this context, for a rule h :- $t_1, \ldots, t_n$ to be valid, it has to fulfill all of the following conditions:

- h is an element of $\mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}}$
- For all t_i, if ! is a subterm of t_i, then $t_i = !$

- For all $t_i = f(t'_1, \dots, t'_m)$, none of $\{=:=, =\backslash=, <, >, =<, >=, \mathit{is}\}$ occur in t'_1 through t'_m
- For all $t_i = f(\dots)$, f is not one of $\{\mathit{abs}, \mathit{sign}, -, +, *, //, **, \mathit{mod}, \mathit{rem}\}$, nor is it a member of $\mathbb{N}$.

An element of $\mathit{Programs}_{\Sigma,\mathcal{V}}$ is called a **program**. ■

The built-in predicates can only appear in the body of the rule. The cut may only appear on its own, not as an argument to any other predicate. Furthermore, an arithmetic comparison may not occur inside the argument of a predicate, but only at the topmost "level" of a term. Similarly, arithmetic operators and integer literals may occur only inside the arguments to a term, but not as a term on their own.

The latter three conditions for the validity of a rule are not formally necessary, but they serve to exclude those formal descriptions that do not correspond to actual Prolog programs. A program that contains any rules that are not valid according to this definition will be rejected by the interpreter. They do not, however, influence the formal semantics of the program.

Example 3.1 (Formalization of Program 2.3)**.** We construct the formalization P of Program 2.3, which we presented on page 13. The signature Σ of Program 2.3 consists only of the single symbol *fac*, which is a member of Σ_2. The set of variables $\mathcal{V}$ is some superset of $\{X, Y, Y_1\}$.

The program P contains only two rules, which we call r_1 and r_2. They are defined as

$$r_1 := \mathit{fac}(X, Y) \text{ :- } >(X, 0), !, \mathit{fac}(-(X, 1), Y_1), \mathit{is}(Y, *(Y_1, X))$$
$$r_2 := \mathit{fac}(X, Y) \text{ :- } =:=(X, 0), \mathit{is}(Y, 1)$$

We can easily see that both of these rules are valid according to Definition 3.4. Thus, $P = r_1, r_2$ and $P \in \mathit{Programs}_{\Sigma,\mathcal{V}}$. ▲

Every program that is written in the fragment that we discussed in Section 2.2 *(Syntax and Informal Semantics of* Prolog *Fragment)* has exactly one formal definition up to superfluous elements of the signature and the set of variables. We thus assume in the remainder of this thesis that a program is given to us in this formal model instead of plain source code. Furthermore, we often omit Σ and $\mathcal{V}$, as they are easy to infer from the rules of the program.

3.2. States and Evaluation

We have defined the behavior of a program informally in Section 2.2 *(Syntax and Informal Semantics of* Prolog *Fragment)*. However, in order to argue about the behavior in a formally rigorous way, we need a formal semantics. The semantics of Prolog are officially defined in [ISO95]. These semantics are defined via a depth-first search through a tree.

While intuitive, this semantics is not very amenable to abstract interpretation, as most methods of abstract interpretation assume that the behavior of programs is defined via a linear sequence of states. Thus, we present a state-based operational semantics for programs, which was first introduced in [SESK$^+$12]. This semantics defines the evaluation of a query on a program in terms of states. It consists of a definition of states for PROLOG as well as semantic rules that define how states are evaluated.

These semantic rules are defined for the full extent of ISO-PROLOG. We are, however, only going to present the semantic rules that are relevant for our fragment.

The major insight of this state-based semantics is that the program together with the inference algorithm can be treated like an imperative program. Since the algorithm implements a depth-first search through the tree induced by the query, all that must be stored is the current state of the search.

This state is usually stored as a stack of nodes that have not yet been visited and a set of nodes that already have been visited. However, we do not need to store the nodes that were already visited, since the structure of a tree prevents us from reaching them again.

Hence, we only need to keep track of the nodes that still need to be searched. Due to this reasoning, our states consist of nothing but the stack of the depth-first search, which is a stack of goals that need to be shown to hold true. We use the symbol $|$ as a separator for these goals.

One more thing to keep in mind is the cut. In order to be able to evaluate the cut correctly, we need to keep track of the terms that result from a single choice of rules. To do this, we use the symbol ? together with an index in order to denote the end of the "scope" of cuts. Since we have to distinguish between cuts in different scopes, we index both the cut symbols and the end-of-scope markers with a natural number.

Additionally, we must be able to label goals with a rule that we are going to apply for the evaluation of user-defined predicates. We give a more detailed explanation of this behavior later in this section. Furthermore, we need an additional error state to denote an error during arithmetic evaluation as discussed in the previous chapter.

Definition 3.5 (Concrete state)**.** Let P be some program and let Σ and $\mathcal{V}$ be its signature and set of variables, respectively. We define the **goal signature** Σ' of P as

$$\Sigma_G := (\Sigma_{BI} \setminus \{!\}) \cup \{!_m, ?_m \mid m \in \mathbb{N}_0\},$$

where all $!_m$ and $?_m$ have arity 0. We define the set

$$\begin{aligned}&\mathit{ConcreteStates}_{P,\mathcal{V}} := \\ &\quad\{\langle \mathit{Err}\rangle\} \cup \{\langle g_1 \mid \cdots \mid g_n\rangle \mid n \in \mathbb{N}_0, g_i \in \mathit{Goals}_{\Sigma_G,\mathcal{V}} \times (\mathit{Rules}_{\Sigma_G,\mathcal{V}} \cup \{\epsilon\}), g_i \text{ is valid}\},\end{aligned}$$

where, in order for a goal $g_i = t_1, \ldots, t_{n_i}$ to be valid, it has to fulfill all of the following conditions:

- If $?_k$ is a subterm of some t_j, then $n_i = 1$ and $t_j = ?_\mathrm{k}$
- If $!_k = t_j$ in g_i for some natural numbers k and j, then there must exist some $m \in [i+1; n]$ such that $g_m = ?_\mathrm{k}$

An element of $\mathit{ConcreteStates}_{P,\mathcal{V}}$ is called a **concrete state** of P. We write an element (g, r) of $\mathit{Goals}_{\Sigma_G,\mathcal{V}} \times (\mathit{Rules}_{\Sigma_G,\mathcal{V}} \cup \{\epsilon\})$ as g^r. If $r = \epsilon$, we simply write g.

■

The original definition of the program state in [SESK+12] also stores the answer substitution that is being built in the state. In the remainder of this thesis, however, we do not require this substitution. Thus, we omit it in the definition of the state.

Example 3.2 (Concrete state)**.** Reconsider Program 2.1 and Figure 2.1 on pages 8 and 11, respectively. When the depth-first search is just about to enter state 6, there are three unexplored nodes on its stack, namely the nodes 6, 7, and 9. The nodes 6 and 7 result from the same case distinction, namely the one in node 5. There are two other case distinctions before that one, namely those in state 4 and 1. Thus, there are three scope markers in the current state, $?_1, ?_2$, and $?_3$, denoting the case distinctions in states 1, 4, and 5, respectively. Hence, the current state of the inference is:

$$\begin{aligned}\mathfrak{s} = \langle\; & \mathit{mother}(\mathit{bob}, \mathit{bob}) \mid \mathit{mother}(\mathit{claire}, \mathit{bob}) \mid ?_3 \mid \\ & \qquad \mathit{ancestor}(X, Z_2), \mathit{ancestor}(X, Z_1), \mathit{mother}(Z_1, \mathit{bob}) \mid ?_2 \mid ?_1\rangle\end{aligned}$$

We write $\mathfrak{s}$ to denote concrete states. ▲

Using this definition, we can now define the initial state of an evaluation. At the start of the inference, there is only a single node that is yet to be searched, namely the node consisting only of the starting term.

Definition 3.6 (Initial state)**.** Let P be some program over the signature Σ and let $\mathcal{V}$ be some set of variables. Let $t \in \mathit{Terms}_{\Sigma_{BI}}$ be some term. The **initial state** $\mathfrak{s}^t_{init}$ of P is defined as:

$$\mathfrak{s}^t_{init} := \langle t \rangle$$

■

Our goal for the remainder of this section is to define the execution relation $\rightarrow$ over pairs of concrete states. The idea behind this relation is that $\mathfrak{s} \rightarrow \mathfrak{s}'$ holds true for two states $\mathfrak{s}$ and $\mathfrak{s}'$ if $\mathfrak{s}$ evaluates to $\mathfrak{s}'$ with a single step of the inference algorithm.

The only possible steps are either a continuation of the depth-first search into the leftmost unexplored node or the abandonment of the leftmost branch to continue with the second one from the left. We define this relation using the standard notation for semantic rules, where we write

$$\frac{\mathfrak{s}}{\mathfrak{s}'}$$

to denote that state $\mathfrak{s}$ evaluates to $\mathfrak{s}'$. In these rules, t and T denote terms and sequences of terms, whereas g and G denote goals and sequences of goals.

All semantic rules operate by evaluating the first term of the first goal of a state. The aim is to emulate depth-first search, the behavior of which depends on the first term of the leftmost goal of the tree.

Concrete Evaluation Rule 3.1 Success Rule

$$\text{Success}\frac{\langle \Box \mid G\rangle}{\langle G\rangle}$$

The easiest case occurs if the first goal of a state is the empty goal. In this case the tree-based inference succeeded in the leftmost branch and we can simply continue the inference with the next goal. This behavior is captured in Concrete Evaluation Rule 3.1.

The other major mechanism that we need to emulate is the evaluation of terms. In our intuitive explanation in Section 2.2 we already explained the process of unification of two terms. We now give a precise and formal definition of this notion.

Definition 3.7 (Unification, most general unifier)**.** Let t_1 and t_2 be terms over some signature Σ and some set of variables $\mathcal{V}$. We call a function $\sigma : \mathcal{V} \to \mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}}$ a **unifier** of t_1 and t_2 if

$$\sigma(t_1) = \sigma(t_2)$$

holds true. If there exists no unifier for some pair of terms t_1 and t_2, we say that t_1 and t_2 **do not unify**.

If for some pair of terms t_1 and t_2 there exist unifiers σ_1 and σ_2, we call σ_1 **more general** than σ_2 if there is some other substitution $\delta : \mathcal{V} \to \mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}}$, which is not the identity function, such that

$$\delta(\sigma_1(t_1)) = \sigma_2(t_1)$$

holds true.

For two terms t_1 and t_2 we call the unifier σ for which no more general unifier exists the **most general unifier** of t_1 and t_2, which we denote by $\mathit{mgu}(t_1, t_2) = \sigma$. If t_1 and t_2 do not unify, we write $\mathit{mgu}(t_1, t_2) = \bot$. ■

In the remainder of this thesis, we also write $t\sigma$ to denote $\sigma(t)$. We also write $T\sigma$, $g\sigma$, and $G\sigma$ to denote the pointwise application of σ to all elements of the sequence of terms T, the goal g, and the sequence of goals G, respectively.

For each pair of terms t_1 and t_2, there exists at most one most general unifier up to variable renaming. Hence, the function mgu is well-defined. It is furthermore efficiently computable, as shown in [DEDC96, Chapter 3].

Using the most general unifier, we now define rules for the evaluation of user-defined predicates. While such predicates are evaluated in a single step in the original inference algorithm, we split the evaluation in the state-based semantics. Recall that in order to evaluate a term, we first collect all the rules that are applicable to it. We then check whether or not the head of the rule unifies with the term.

In a first step, we copy the term to be evaluated multiple times and label each copy with the rule that we want to apply. For this we introduce Concrete Evaluation Rule 3.2. In this rule, we also rename all the variables occurring in the rules in order to safeguard against inadvertent aliasing of variables. We also use the helper function $\mathit{Slice}_P(t)$, which returns a sequence of those rules in P whose head starts with the same function symbol as t. Finally, we add a scope end marker in order to make it possible to evaluate cuts correctly. These markers will later be used in Concrete Evaluation Rule 3.5.

Concrete Evaluation Rule 3.2 Case Rule

$$\text{Case}\frac{\langle (t,T) \mid G\rangle}{\left\langle (t,T)^{r'_1} \mid \ldots \mid (t,T)^{r'_n} \mid ?_m \mid G\right\rangle}$$

Where: $\mathit{Slice}_P(t) = r_1, \ldots, r_n$
$r'_i := r_i\,[\mathcal{V} \mapsto \mathcal{V}_m]\,[! \mapsto !_m]$
$m \in \mathbb{N}$ does not occur as an index in t, T or G
$t = f(\ldots)$ and $f \notin \{=:=, =\backslash=, <, >, =<, >=, \mathit{is}\}$

In the second step, we check for each rule individually if its head unifies with the current term. If this is the case, we replace the term with the body of the rule and continue the inference using Concrete Evaluation Rule 3.3. If, however, the head of the rule and the term do not unify, we immediately abandon that branch and continue with the next unexplored branch. For this, we introduce Concrete Evaluation Rule 3.4.

Concrete Evaluation Rule 3.3 Evaluation Rule

$$\text{Eval}\frac{\left\langle (t,T)^{h\,:\text{-}\,B} \mid G\right\rangle}{\langle (B\sigma, T\sigma) \mid G\rangle}$$

Where: $\sigma = \mathit{mgu}(t,h) \neq \bot$

Concrete Evaluation Rule 3.4 Backtracking Rule

$$\text{Backtrack}\frac{\left\langle (t,T)^{h\,:\text{-}\,B} \mid G\right\rangle}{\langle G\rangle}$$

Where: $\mathit{mgu}(t,h) = \bot$

It now remains to define rules to evaluate built-in predicates. We have to deal with two kinds of these predicates, namely the cut and the arithmetic predicates. First, we define rules for handling the cut. In order to evaluate this predicate, we use the end-of-scope markers that we introduce into the state when using Concrete Evaluation Rule 3.2. We simply remove all goals except for the current one up to, but not including the corresponding marker. This is implemented in Concrete Evaluation Rule 3.5.

We keep the current end-of-scope marker, since there may be more cuts contained in T. This strategy will eventually lead to a case where the current goal is nothing but this end marker. Since it does not have any effect on the computation, but exists only for

Concrete Evaluation Rule 3.5 Cut Rule

$$\text{Cut}\frac{\langle (!_n, T) \mid G \mid ?_n \mid G' \rangle}{\langle T \mid ?_n \mid G' \rangle}$$

the purpose of bookkeeping, we can simply remove it from the state and continue with the next goal. We formalize this behavior in Concrete Evaluation Rule 3.6.

Concrete Evaluation Rule 3.6 Failure Rule

$$\text{Failure}\frac{\langle ?_n \mid G \rangle}{\langle G \rangle}$$

The remaining semantic rules handle arithmetic comparisons and assignments. We start with the handling of arithmetic comparisons. As explained informally in Section 2.2, in order to perform an arithmetic comparison, we first evaluate both arithmetic expressions on the left- and right-hand side. If either evaluation fails, we transition to an error state and the inference terminates. Should the evaluations of the expressions on both sides succeed, however, we simply perform the arithmetic comparison of the two resulting natural numbers.

In the case that this comparison succeeds, we continue with the next term of the current goal. Should the comparison fail, we abandon the current goal and continue with the next one. We formalize the evaluation of arithmetic expressions and comparisons in the functions $eval_E$ and $eval_C$.

Definition 3.8 (Evaluation of Arithmetic Expressions, Comparisons)**.** We define the helper functions $P2M_E^{un}$ and $P2M_E^{bin}$ to translate from Prolog notation to mathematical notation as follows:

f	$-$	abs	$sign$
$P2M_E^{un}(f,n)$	$-1 \cdot n$	$sign(n) \cdot n$	$sign(n)$

f	$+$	$-$	$*$
$P2M_E^{bin}(f,n_1,n_2)$	$n_1 + n_2$	$n_1 - n_2$	$n_1 \cdot n_2$

f	$//$	$**$
$P2M_E^{bin}(f,n_1,n_2)$	$sign(n_1/n_2) \cdot \lfloor \lvert n_1/n_2 \rvert \rfloor$	$n_1^{n_2}$

f	mod	rem
$P2M_E^{bin}(f,n_1,n_2)$	$n_1 - (\lfloor n_1/n_2 \rfloor) \cdot n_2$	$n_1 - (sign(n_1/n_2) \cdot \lfloor \lvert n_1/n_2 \rvert \rfloor) \cdot n_2$

where $sign(n) := 1$ if $n \geq 0$ and $sign(n) := -1$ otherwise.

Let t be some term. We define the **evaluation function for expressions** of the type $eval_E : Terms \to \mathbb{Z} \cup \{\bot\}$ as follows:

$$\begin{array}{ll}
eval_E(t) = t & \text{if } t \in \mathbb{Z} \\
eval_E(op(t)) = P2M_E^{un}(f, n) & \text{if } n = eval_E(t) \neq \bot \\
 & \text{and } op \in \{abs, sign, -\} \\
eval_E(op(t_1, t_2)) = P2M_E^{bin}(f, n_1, n_2) & \text{if } n_1 = eval_E(t_1) \neq \bot, n_2 = eval_E(t_2) \neq \bot, \\
 & \text{and } op \in \{+, -, *, **\} \\
eval_E(op(t_1, t_2)) = P2M_E^{bin}(f, n_1, n_2) & \text{if } n_1 = eval_E(t_1) \neq \bot, n_2 = eval_E(t_2) \neq \bot, \\
 & n_2 \neq 0, \text{ and } op \in \{//, mod, rem\} \\
eval_E(t) = \bot & \text{otherwise}
\end{array}$$

We define the conversion function for comparison operators $P2M_C$ with the following value table:

$\bowtie$	=:=	=\\=	<	>	=<	>=
$P2M_C(\bowtie)$	$=$	$\neq$	$<$	$>$	$\leq$	$\geq$

Using this helper function, we define the **evaluation function for comparisons** of the type $eval_C : Terms \to \{true, false, \bot\}$ as follows:

$$\begin{array}{ll}
eval_C(\bowtie(t_1, t_2)) \equiv (n_1\ P2M_C(\bowtie)\ n_2) & \text{if } n_1 = eval_E(t_1) \neq \bot, n_2 = eval_E(t_2) \neq \bot, \\
 & \text{and } \bowtie \in \{=:=, =\backslash=, <, >, =<, >=\} \\
eval_C(t) = \bot & \text{otherwise}
\end{array}$$

■

These evaluation functions have the properties that one would assume from such functions. In order to show the soundness of our abstract semantics later on, we need two such properties, which we state here.

Lemma 3.1. *Let t be some term. If $eval_C(t) \neq \bot$, then there exists no subterm t' of t such that $eval_C(t') = \bot$.* ■

Proof. This can easily be seen using structural induction over the structure of t. We omit the full proof for the sake of readability at this point. □

Lemma 3.2. *Let t be some term. If $eval_C(t) \neq \bot$, then there exists no subterm t' of t, such that t' is a program variable.* ■

Proof. This statement follows directly from Lemma 3.1. Since $eval_C(\texttt{X})$ is undefined for all program variables X, it would contradict that lemma if there were a term that contained a program variable, but that evaluated to something other than $\bot$. □

Using these definitions we can now concisely define rules for the three possible results of an arithmetic comparison. The error case, in which the evaluation of the expression on either side fails, is formalized in Concrete Evaluation Rule 3.7. The case in which the evaluation of the comparison succeeds is formalized in Concrete Evaluation Rule 3.8. If

Concrete Evaluation Rule 3.7 Arithmetic Comparison Rule (Error)

$$\text{ArithCompErr}\frac{\langle (t, T) \mid G\rangle}{\langle \mathit{Err}\rangle}$$

Where: $t = \bowtie(t_1, t_2)$, for $\bowtie\, \in \{=:=, =\backslash=, <, >, =<, >=\}$
$\mathit{eval}_C(t) = \bot$

Concrete Evaluation Rule 3.8 Arithmetic Comparison Rule (Success)

$$\text{ArithCompSucc}\frac{\langle (t, T) \mid G\rangle}{\langle T \mid G\rangle}$$

Where: $t = \bowtie(t_1, t_2)$, for $\bowtie\, \in \{=:=, =\backslash=, <, >, =<, >=\}$
$\mathit{eval}_C(t) \equiv \mathit{true}$

the comparison fails, we apply Concrete Evaluation Rule 3.9 and continue with the next goal.

The final built-in function symbol that we need to handle is the *is*-symbol. This symbol expects a term on its left-hand side and an arithmetic expression on its right-hand side. If the evaluation of the expression fails, we transition to the same error state as before using Concrete Evaluation Rule 3.10.

If the evaluation of the expression succeeds and the result unifies with the term on the left-hand side, then the resulting substitution is applied to the remainder of the terms and evaluation continues with the remaining terms. This behavior is formalized in Concrete Evaluation Rule 3.11.

If the result of the evaluation of the right-hand side does not unify with the left-hand side of the function symbol, then this is treated like a normal unification error and the evaluation continues with the remaining goals. We apply rule Concrete Evaluation Rule 3.12 in this case.

Using these rules, we are able to emulate the inference algorithm of Prolog. We present the application of these rules in the following example.

Example 3.3 (State-based evaluation of Program 2.3 on $\mathit{fac}(1, \mathit{Res})$)**.** Consider Program 2.3 on page 13, which we call P in the remainder of this example, and the term $q := \mathit{fac}(1, \mathit{Res})$. We have shown the evaluation of this term using the original inference

Concrete Evaluation Rule 3.9 Arithmetic Comparison Rule (Failure)

$$\text{ArithCompFail}\frac{\langle (t, T) \mid G\rangle}{\langle G\rangle}$$

Where: $t = \bowtie(t_1, t_2)$, for $\bowtie\, \in \{=:=, =\backslash=, <, >, =<, >=\}$
$\mathit{eval}_C(t) \equiv \mathit{false}$

Concrete Evaluation Rule 3.10 Arithmetic Evaluation Rule (Error)

$$\text{IsErr}\,\frac{\langle (is(t_1,t_2),T) \mid G\rangle}{\langle Err\rangle}$$

Where: $eval_E(t_2) = \bot$

Concrete Evaluation Rule 3.11 Arithmetic Evaluation Rule (Success)

$$\text{IsSucc}\,\frac{\langle (is(t_1,t_2),T) \mid G\rangle}{\langle T\sigma \mid G\rangle}$$

Where: $eval_E(t_2) \neq \bot$
$\sigma = mgu(t_1, eval_E(t_2)) \neq \bot$

algorithm in Example 2.9 on page 13. In this example we evaluate this term on P using the state-based semantics.

According to Definition 3.6, the initial state of this computation is:

$$\mathfrak{s}^q_{init} = \langle fac(1, Res)\rangle$$

Since the first and only goal has a term at its head that starts with a user-defined function symbol and since the goal is not labeled with any clause, we apply Concrete Evaluation Rule 3.2 and evaluate $\mathfrak{s}^q_{init}$ to

$$\mathfrak{s}_1 := \Big\langle fac(1, Res)^{fac(X_1,Y_1)\ \text{:-}\ >(X_1,0),!_1,fac(-(X_1,1),Y1_1),is(Y_1,*(Y1_1,X_1))} \mid \\ fac(1, Res)^{fac(X_1,Y_1)\ \text{:-}\ =:=(X_1,0),is(Y_1,1)} \mid ?_1\Big\rangle$$

Now the first term of the first goal is a labeled term, which we can evaluate using either Concrete Evaluation Rule 3.3 or Concrete Evaluation Rule 3.4. Since $fac(X_1, Y_1)$ and $fac(1, Res)$ unify with the most general unifier $\sigma_1 : X_1 \mapsto 1, Y_1 \mapsto Res_1, Res \mapsto Res_1$, we can apply Concrete Evaluation Rule 3.3 and receive the state:

$$\mathfrak{s}_2 := \langle >(1,0), !_1, fac(-(1,1), Y1_1), is(Res_1, *(Y1_1, 1)) \mid \\ fac(1, Res)^{fac(X_1,Y_1)\ \text{:-}\ =:=(X_1,0),is(Y_1,1)} \mid ?_1\Big\rangle$$

Concrete Evaluation Rule 3.12 Arithmetic Evaluation Rule (Failure)

$$\text{IsFail}\,\frac{\langle (is(t_1,t_2),T) \mid G\rangle}{\langle G\rangle}$$

Where: $eval_E(t_2) \neq \bot$
$mgu(t_1, eval_E(t_2)) = \bot$

At this point the first term of the first goal of s_2 is a built-in predicate, namely the larger-than predicate. Since $eval_C(>(1,0)) = eval_E(1) > eval_E(0) = 1 > 0 \equiv true$ holds, we can use Concrete Evaluation Rule 3.8 and evaluate $\mathfrak{s}_2$ to:

$$\mathfrak{s}_3 := \Big\langle !_1, fac(-(1,1), Y1_1), is(Res_1, *(Y1_1, 1)) \mid fac(1, Res)^{fac(X_1,Y_1) \text{ :- } =:=(X_1,0), is(Y_1,1)} \mid ?_1 \Big\rangle$$

We now have to evaluate the cut. The only goal between the current one and the end-of-scope marker $?_1$ is $fac(1, Res)^{fac(X_1,Y_1) \text{ :- } =:=(X_1,0), is(Y_1,1)}$, which is removed by evaluating cut. As mentioned before, we do not remove the end-of-scope marker, but keep it in order to handle further cuts. Using Concrete Evaluation Rule 3.5, we evaluate $\mathfrak{s}_3$ to:

$$\mathfrak{s}_4 := \langle fac(-(1,1), Y1_1), is(Res_1, *(Y1_1, 1)) \mid ?_1 \rangle$$

The next step consists of the evaluation of a user-defined predicate again. For this, we use Concrete Evaluation Rule 3.2 and pick the number 2 as the unused index to create two goals. These are labeled with their respective rules. Thus, we receive the state:

$$\begin{aligned}\mathfrak{s}_5 := \Big\langle & fac(-(1,1), Y1_1),\\ & is(Res_1, *(Y1_1, 1))^{fac(X_2,Y_2) \text{ :- } >(X_2,0), !_2, fac(-(X_2,1), Y1_2), is(Y_2, *(Y1_2, X_2))} \mid\\ & fac(-(1,1), Y1_1),\ is(Res_1, *(Y1_1, 1))^{fac(X_2,Y_2) \text{ :- } =:=(X_2,0), is(Y_2,1)} \mid ?_2 \mid ?_1 \Big\rangle\end{aligned}$$

We see that $fac(X_2, Y_2)$ and $fac(-(1,1), Y1_1)$ unify with the unifier

$$\sigma_2 : X_2 \mapsto -(1,1), Y_2 \mapsto Y_3, Y1_1 \mapsto Y_3,$$

which we apply to the body of the rule and the remaining terms. We apply rule Concrete Evaluation Rule 3.3 to receive:

$$\begin{aligned}\mathfrak{s}_6 := \Big\langle & >(-(1,1), 0),\\ & !_2, fac(-(-(1,1),1), Y1_2), is(Y_3, *(Y1_2, -(1,1))), is(Res_1, *(Y_3, 1)) \mid\\ & fac(-(1,1), Y1_1),\ is(Res_1, *(Y1_1, 1))^{fac(X_2,Y_2) \text{ :- } =:=(X_2,0), is(Y_2,1)} \mid ?_2 \mid ?_1 \Big\rangle\end{aligned}$$

Now, since $1 - 1 > 0$ is obviously equivalent to *false*, we apply Concrete Evaluation Rule 3.9 and abandon the current goal to continue with the remaining ones. This leads to the state:

$$\mathfrak{s}_7 := \Big\langle fac(-(1,1), Y1_1), is(Res_1, *(Y1_1, 1))^{fac(X_2,Y_2) \text{ :- } =:=(X_2,0), is(Y_2,1)} \mid ?_2 \mid ?_1 \Big\rangle$$

At this point, we check whether the chosen rule is applicable to the first term of the first goal. We see that $fac(X_2, Y_2)$ and $fac(-(1,1), Y1_1)$ unify with the same most general unifier σ_2 as before. Again, we use Concrete Evaluation Rule 3.3 and Concrete Evaluation Rule 3.8 and receive:

$$\mathfrak{s}_8 := \langle =:=(-(1,1), 0), is(Y_3, 1), is(Res_1, *(Y_3, 1)) \mid ?_2 \mid ?_1 \rangle$$

and:

$$\mathfrak{s}_9 := \langle is(Y1_1, 1), is(Res_1, *(Y1_1, 1)) \mid ?_2 \mid ?_1 \rangle$$

In the next step, we evaluate the *is*-predicate for the first time. Since $Y1_1$ and 1 unify with $\sigma_3 : Y1_1 \mapsto 1$, we can apply Concrete Evaluation Rule 3.11 and arrive at:

$$\mathfrak{s}_{10} := \langle is(Res_1, *(1, 1)) \mid ?_2 \mid ?_1 \rangle$$

We apply the same rule again and unify Res_1 with the result of evaluating $*(1, 1)$, that is, with the unifier $\sigma_4 : Res_1 \mapsto 1$. Since this was the last term in this goal, we end up with the empty goal in:

$$\mathfrak{s}_{10} := \langle \square \mid ?_2 \mid ?_1 \rangle$$

We can now apply Concrete Evaluation Rule 3.1 and, subsequently, Concrete Evaluation Rule 3.6 twice and receive the sequence of states:

$$\mathfrak{s}_{11} := \langle ?_2 \mid ?_1 \rangle$$

$$\mathfrak{s}_{12} := \langle ?_1 \rangle$$

$$\mathfrak{s}_{13} := \langle \epsilon \rangle$$

At this point we can apply no more rules and thus, the computation has finished. ▲

Since the concrete evaluation rules mirror the inference algorithm, which is deterministic, they are deterministic as well.

Lemma 3.3 (Deterministic concrete semantics)**.** *Let $\mathfrak{s}$ be a concrete state. If there exists a concrete evaluation rule r that is applicable to $\mathfrak{s}$, then no other rule $r' \neq r$ is applicable to $\mathfrak{s}$.* ■

Proof. For the complete proof, please refer to [SESK+11]. □

3.3. Equivalence to ISO Semantics

The semantics presented in the previous section are equivalent to those of ISO-Prolog for the properties that we are interested in, which are termination-behavior, complexity, and the reachability of program errors. These properties are shown in [SESK+11, Theorem 3]. We merely state these properties separately here and refer to the original publication for the proofs.

In order to properly formulate these properties, we use the state transition relation $\to$ and its transitive closure $\to^*$. We also use the notion of a final state, which is a state from which the execution cannot continue.

Definition 3.9 (Concrete state transition relation, final state)**.** Let $\mathfrak{s}$ and $\mathfrak{s}'$ be concrete states. We write $\mathfrak{s} \to \mathfrak{s}'$ if there exists a concrete evaluation rule whose application transforms $\mathfrak{s}$ into $\mathfrak{s}'$. We furthermore write $\mathfrak{s} \to^* \mathfrak{s}'$ if there is a sequence of concrete states $\mathfrak{s}_1, \ldots, \mathfrak{s}_n$ with $n \geq 1$, such that $\mathfrak{s} = \mathfrak{s}_1$, $\mathfrak{s}' = \mathfrak{s}_n$ and $\mathfrak{s}_i \to \mathfrak{s}_{i+1}$ for all $i \in [1; n-1]$. If $n \gneq 1$, we write $\mathfrak{s} \to^+ \mathfrak{s}'$.

We say that $\mathfrak{s}$ is a **final state** if there exists no state $\mathfrak{s}'$ such that $\mathfrak{s} \to \mathfrak{s}'$ holds true. ■

Lemma 3.4 (Equivalence of state-based semantics and ISO-semantics)**.** *Let P be a program and let q be a term. For any state $\mathfrak{s}$ it holds true that $\mathfrak{s}^q_{init} \to^* \mathfrak{s}$ if and only if the execution of q according to the ISO-semantics leads to a tree corresponding to $\mathfrak{s}$.* ■

Lemma 3.5 (Equivalence of state-based semantics and ISO-semantics (termination))**.** *Let P be a program and let q be a term. There exists a final state $\mathfrak{s}$ such that $\mathfrak{s}^q_{init} \to^* \mathfrak{s}$ if and only if the execution of q according to the ISO-semantics terminates.* ■

Lemma 3.6 (Equivalence of state-based semantics and ISO-semantics (complexity))**.** *Let P be a program and let $\mathfrak{s}$ be a final state of the program with $\mathfrak{s}^q_{init} \to^* \mathfrak{s}$. Furthermore, let $\mathfrak{s}_1, \ldots, \mathfrak{s}_l$ be the intermediate states that occur during the evaluation of $\mathfrak{s}^q_{init}$ to $\mathfrak{s}$. Finally, let k be the length of the execution of q according to the ISO-semantics. Then $l \in \Theta(k)$ holds true.* ■

Proof. The proofs for Lemma 3.4, Lemma 3.5, and Lemma 3.6 can be found in [SESK$^+$11, Appendix C]. □

4. Abstract Semantics

The goal of this section is to develop an abstract semantics on the basis of the concrete state-based semantics from the previous section. This abstract semantics is based on work presented in [GSSK$^+$12].

In Section 4.1 we define the domain that we use for our abstract semantics. Section 4.2 contains an overview over the general idea of the abstract interpretation. We present rules that allow us to evaluate these abstract states in Section 4.3 and Section 4.4. The former section contains evaluation rules previously published in [GSSK$^+$12], whereas we develop new rules in the latter section. In Section 4.5 we prove that this abstract semantics is both deterministic and sound.

4.1. The Abstract Domain

In this section we define the abstract states that we are going to evaluate with our semantics. We base our definition on the definition of abstract states in [GSSK$^+$12]. Since our focus lies on the abstract evaluation of arithmetic predicates, we first define so-called arithmetic states in Section 4.1.1. We then define abstract states in Section 4.1.2, which use arithmetic states as one of their components.

Our goal in this section is to define a representation of an infinite set of concrete states. For this, we use two kinds of variables, namely program- and term variables. All variables that occurred so far in this thesis are program variables, since they occur in programs. Term variables, on the other hand, occur in positions where we do not know the subterm precisely.

This notion will be defined formally in Section 4.1.2. For now, it suffices to note that we use disjoint sets of program variables $\mathcal{V}$ and sets of term variables $\mathcal{T}$, where the term variables are those that are of interest for our abstraction.

4.1.1. Arithmetic States

We use term variables to denote positions for which we do not know the term precisely. However, we want to be able to retain some precision when evaluating arithmetic comparisons and evaluations. To this end we use arithmetic states, which store knowledge about the result of the arithmetic expressions that the variables represent.

The general idea for these states is to hold a set of assignments of numeric values to variables. We will see how we use these states to improve the precision of our abstraction in Section 4.1.2.

Definition 4.1 (Arithmetic State)**.** Let $\mathcal{T}$ be a set of variables. We define the set of all assignments of integers to these variables as

$$\mathit{ArithAssignments}_{\mathcal{T}} := \{f \mid f : \mathcal{T} \to \mathbb{Z}\}$$

We call any subset $\mathcal{A}$ of *ArithAssignments* an **arithmetic state**. ■

Example 4.1 (Arithmetic state)**.** We pick the set of variables $\mathcal{T}$ such that $\{X, Y\} \subseteq \mathcal{T}$ holds true. The set

$$\mathcal{A} := \{\sigma \mid \sigma(X) > 0, \sigma(Y) = \sigma(X) - 1\}$$

is an arithmetic state that contains infinitely many assignments. ▲

Since arithmetic states do not need to be finite and may be very large, we need a concise notation for them. We choose to denote an arithmetic state by a set of relations. The arithmetic state corresponding to a set of relations is the set of all those assignments that satisfy all the relations in the set.

Definition 4.2 (Set notation of arithmetic states)**.** Let R be a finite set of relations over the integers and over a set of variables $\mathcal{T}$. We define the **arithmetic state associated with** R as

$$\mathcal{A}_R := \{\sigma \mid \forall r \in R.\ r\sigma \text{ is a tautology}\}$$

■

Example 4.2 (Set notation of arithmetic states)**.** We pick $\mathcal{T}$ and $\mathcal{A}$ as in Example 4.1. If we furthermore pick

$$R := \{X > 0, Y = X - 1\},$$

then we have $\mathcal{A} = \mathcal{A}_R$. ▲

Quite often we also need the statement that one arithmetic state is a subset of another. In a lot of cases, the latter state is defined by a single relation. We use the symbol $\models$ for this.

Definition 4.3 (Modeling of relations)**.** Let $\mathcal{T}$ be some set of variables, let $\mathcal{A}$ be some arithmetic state and let r be some relation over the integers and $\mathcal{T}$. We say that $\mathcal{A}$ **models** r if and only if $\mathcal{A} \subseteq \mathcal{A}_{\{r\}}$ holds true. If this statement does not hold true, we say that $\mathcal{A}$ **does not model** r. We write $\mathcal{A} \models r$ and $\mathcal{A} \not\models r$, respectively. ■

This notation allows us to concisely denote that some relation is "implied" by an arithmetic state. It does, however, have the counterintuitive property that there exist relations for which neither the relation nor its negation are modeled by the arithmetic state.

Example 4.3 (Modeling of relations)**.** We pick $\mathcal{T}$ and $\mathcal{A}$ to be the set of variables and the arithmetic state from Example 4.1, respectively. Then both $\mathcal{A} \models X > 0$ and $\mathcal{A} \not\models X \leq 0$ holds true. Furthermore, both of $\mathcal{A} \not\models Y \leq 0$ and $\mathcal{A} \not\models Y > 0$ hold true. ▲

In the following sections, we often use the representations of relations as PROLOG terms and as mathematical objects. In order to simplify reading of this thesis, we do not differentiate between these two notations. We also use the operator $\neg$ on relations in order to negate the relation. This is well-defined as the set of relations $\{=, \neq, <, >, \leq, \geq\}$ is closed under negation.

Example 4.4 (Relations as terms and mathematical objects, negation)**.** The relation $X = Y$ can also be written as the term $=:=(X, Y)$. We write $\neg(X = Y)$ and $\neg(=:=(X, Y))$ to denote the same thing, namely the relation $X \neq Y$ or the term $=/=(X, Y)$. ▲

One useful observation to make is the fact that the set-notation behaves similarly to traditional sets. More precisely, the following lemma holds true, which allows us to keep unnecessary updates to arithmetic states to a minimum.

Lemma 4.1. *Let $\mathcal{A}$ be an arithmetic state and let r be a relation. Then the statement*

$$\mathcal{A} \models r \textbf{ implies } \mathcal{A} \cap \mathcal{A}_{\{r\}} = \mathcal{A}$$

holds true. ■

Proof. We first take the set-notation of $\mathcal{A}_{\{r\}}$ apart and see that $\mathcal{A} \cap \mathcal{A}_{\{r\}}$ is the same as $\mathcal{A} \cap \{\sigma \mid r\sigma \text{ is a tautology}\}$. We proceed to show that $\mathcal{A}$ is a subset of $\{\sigma \mid r\sigma \text{ is a tautology}\}$. For this, let $\sigma \in \mathcal{A}$. Since $\mathcal{A} \models r$ holds true by assumption, we know that for all $\sigma \in \mathcal{A}$, it holds true that $r\sigma$ is a tautology. Thus $\sigma \in \mathcal{A}_{\{r\}}$ holds true. Hence, we have $\mathcal{A}_{\{r\}} \subseteq \mathcal{A}$. Thus, both $\mathcal{A} \cap \{\sigma \mid r\sigma \text{ is a tautology}\} = \mathcal{A}$ and $\mathcal{A} \cap \mathcal{A}_{\{r\}} = \mathcal{A}$ hold true. □

Using this notion of arithmetic states, we now define abstract program states.

4.1.2. Abstract Program States

After having defined arithmetic states in the previous section, we now use them as a component of abstract states. The aim of this section is to define abstract states, where each abstract state represents a potentially infinite set of concrete states.

An abstract state consists mainly of two components. The first component is a sequence of goals, which is very similar to a concrete state. In contrast to the concrete state, whose goals only contained a single kind of variables, this sequence may also contain term variables. These term variables may be replaced by terms containing only program variables to receive a concrete state that is represented by the abstract state. Since this idea on its own would result in a very coarse abstraction, we constrain the admissible substitutions using the second part of the abstract state, the so-called knowledge base, which we discuss in the latter part of this section.

We call the kind of variables used so far "program variables" in order to distinguish them from term variables. To aid readability, we write program variables as $\mathtt{X}$ and term variables as X in the remainder of this thesis. Similarly to program variables, we denote the term variables that occur in a term t, a sequence of terms T, a goal g, or a sequence of goals G as $\mathcal{T}(t)$, $\mathcal{T}(T)$, $\mathcal{T}(g)$, and $\mathcal{T}(G)$, respectively.

Example 4.5 (Program variables and term variables)**.** Consider the sequence of goals

$$\langle \mathit{fac}(-(X,1),\texttt{Y1}), \mathit{is}(\texttt{Y},*(\texttt{Y1},X))\rangle\,,$$

which contains the program variables `Y` and `Y1` as well as the term variable X. This sequence of goals represents, for example, the concrete state

$$\langle \mathit{fac}(-(1,1),\texttt{Y1}), \mathit{is}(\texttt{Y},*(\texttt{Y1},1))\rangle\,,$$

since both occurrences of the single term variable X have been replaced with the term 1.

It does not, however, represent

$$\langle \mathit{fac}(-(1,1),\texttt{Y1}), \mathit{is}(\texttt{Y},*(\texttt{Y1},X))\rangle\,,$$

since this sequence still contains the term variable X. It also does not represent

$$\langle \mathit{fac}(-(-(Z,1),1),\texttt{Y1}), \mathit{is}(\texttt{Y},*(\texttt{Y1},-(Z,1)))\rangle\,,$$

in which both occurrences of X have been replaced with the term $-(Z,1)$, but the new terms still contain the term variable Z. ▲

This notion of term variables would theoretically suffice to define abstract states. However, this abstraction would be too coarse for any nontrivial analysis. Thus, we keep track of additional information to constrain the possible instantiations of the term variables.

More precisely, we keep track of four major restrictions on the instantiations: groundness, non-unifiability, instantiation with arithmetic expressions and the arithmetic state.

The first major property of interest is whether or not a term may contain any variables. A term which does not contain any variables is called a ground term.

Definition 4.4 (Groundness)**.** A term t is called **ground** if $\mathcal{V}(t) \cup \mathcal{T}(t) = \emptyset$ holds true. ■

Our definition of an abstract state contains a set of those term variables which may only be instantiated to ground terms.

In addition to that, we also store pairs of terms that may not be instantiated to unifying terms. Furthermore, we keep track of those variables that may only be instantiated to arithmetic expressions whose evaluation does not lead to an error. Finally, we store the relations between the variables that we know to hold true in an arithmetic state. We combine all of these elements into an abstract state.

Definition 4.5 (Abstract state, ground set, nonunifying pairs, arithmetic variables)**.** Let Σ be some signature and let $\mathcal{V}$ and $\mathcal{T}$ be two disjoint, countable and infinite sets of variables. Furthermore, let P be some program over Σ and $\mathcal{V}$. We define the set of abstract states:

$$\begin{aligned}
&\mathit{AbstractStates}_{\Sigma,\mathcal{V},\mathcal{T}} := \\
&\qquad \mathit{ConcreteStates}_{P,\mathcal{V}\cup\mathcal{T}} \times 2^{\mathcal{T}} \times 2^{\left(\mathit{Terms}_{\Sigma\setminus(\mathcal{V}\cup\mathcal{T}),\mathcal{V}\cup\mathcal{T}} \times \mathit{Terms}_{\Sigma\setminus(\mathcal{V}\cup\mathcal{T}),\mathcal{V}\cup\mathcal{T}}\right)} \times \\
&\qquad\qquad 2^{\mathit{Terms}_{\Sigma\setminus(\mathcal{V}\cup\mathcal{T}),\mathcal{V}\cup\mathcal{T}}} \times 2^{\mathit{ArithAssignments}_{\mathcal{T}}}
\end{aligned}$$

We call an element $s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ of $\mathit{AbstractStates}_{\Sigma,\mathcal{V},\mathcal{T}}$ an **abstract state of P over $\mathcal{T}$**. We call $S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}$ the **state form**, **ground set**, **nonunifying pairs**, **arithmetic variables** and the **arithmetic state of** s, respectively. We also call $\mathcal{G}, \mathcal{U}, \mathcal{E}$, and $\mathcal{A}$ the **knowledge base** of s. ■

The purpose of an abstract state is to represent an infinite set of concrete states. For this, the state form of an abstract state contains term variables, which may be replaced by terms without term variables to receive a concrete state. The replacements that may be applied to the state form are constrained by the knowledge base of the abstract state.

We now formalize these constraints using the notion of conforming substitutions. The idea is that such a substitution is conforming to an abstract state if it replaces all term variables from the state form and respects the restrictions imposed by the knowledge base of the state.

Definition 4.6 (Conforming substitutions, representation of concrete states, concretizations of abstract states)**.** Let Σ be a signature and let $\mathcal{V}$ and $\mathcal{T}$ be sets of program- and term variables, respectively. Furthermore, let P be a program over Σ and $\mathcal{V}$. Let $s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state of P over $\mathcal{T}$ and let $\sigma : \mathcal{T} \to \mathit{Terms}_{\Sigma \setminus (\mathcal{V} \cup \mathcal{T}), \mathcal{V}}$ be a substitution. We say that σ **conforms** to s if all of the following conditions hold:

- $\mathcal{T}(S\sigma) = \emptyset$
- For all X in $\mathcal{G}$, $\mathcal{V}(X\sigma) = \emptyset$
- For all pairs of terms t_1, t_2 in $\mathcal{U}$, $\mathit{mgu}(t_1\sigma, t_2\sigma) = \bot$
- For all X in $\mathcal{E}$, $\mathit{eval}_E(X\sigma) \neq \bot$
- $(\mathit{eval}_E \circ \sigma)\big|_{\mathcal{T}} \in \mathcal{A}$

We say that σ conforms to $(\mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ to denote that the latter four conditions hold true.

If σ conforms to s, we say that s **represents** the concrete state $S\sigma$. Finally, we define:

$$\mathit{Conc}(s) := \{S\sigma \mid s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}), \sigma \text{ conforms to } s\}$$

We call the concrete states in $\mathit{Conc}(s)$ the **concretizations** of s. ■

Example 4.6 (Abstract state and concretizations)**.** Consider the abstract state

$$s = (\langle \mathit{fac}(-(X,1), \mathtt{Y1}), \mathit{is}(\mathtt{Y}, *(\mathtt{Y1}, X))\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X>0\}})$$

This abstract state has the state form $\langle \mathit{fac}(-(X,1), \mathtt{Y1}), \mathit{is}(\mathtt{Y}, *(\mathtt{Y1}, X))\rangle$, its ground set contains the single term variable X, it has the arithmetic variable X and the arithmetic state $\mathcal{A}_{\{X>0\}}$. It contains no nonunifying pairs.

We define the substitution $\sigma : X \mapsto 1$. It replaces the single arithmetic variable X by 1. Hence $X\sigma$ is ground and $\mathit{eval}_E(1) = 1 \neq \bot$ holds true. Also, the single relation $X > 0$ in $\mathcal{A}$ is a tautology after application of σ:

$$\sigma(X > 0) = 1 > 0 \equiv \mathit{true}$$

Hence, σ conforms to s. Thus,

$$\mathfrak{s} := \langle \mathit{fac}(-(X,1), \mathtt{Y1}), \mathit{is}(\mathtt{Y}, *(\mathtt{Y1}, X))\rangle\, \sigma = \langle \mathit{fac}(-(1,1), \mathtt{Y1}), \mathit{is}(\mathtt{Y}, *(\mathtt{Y1}, 1))\rangle$$

is represented by s and $\mathfrak{s} \in \mathit{Conc}(s)$ holds true. ▲

4.2. Structure of Abstract Evaluation

Our goal in the following two sections is to define a semantics that allows us to directly evaluate abstract states. To this end, we are going to define rules that provide a sound abstraction of the concrete execution relation.

The main difference between the concrete and the abstract evaluation rules is the number of successors a single state has. The concrete semantics is linear in the sense that each concrete state has at most one successor state. This is due to the deterministic behavior of the ISO-Prolog semantics. In the abstract state, however, this is not possible.

Example 4.7. Consider the abstract state

$$s := (\langle {>}(X,0) \mid {=:=}(X,0) \mid ?_1 \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

that contains the single term variable X. Two concrete states that are represented by this abstract state are

$$\mathfrak{s}_1 = \langle {>}(1,0) \mid {=:=}(1,0) \mid ?_1 \rangle$$
$$\mathfrak{s}_2 = \langle {>}(-1,0) \mid {=:=}(-1,0) \mid ?_1 \rangle$$

Their respective concrete successor states $\mathfrak{s}'_1$ and $\mathfrak{s}'_2$ are

$$\mathfrak{s}'_1 = \langle \Box \mid {=:=}(1,0) \mid ?_1 \rangle$$
$$\mathfrak{s}'_2 = \langle {=:=}(-1,0) \mid ?_1 \rangle$$

However, there exists no single abstract state s' such that both s'_1 and s'_2 are members of $Conc(s')$. ▲

This example shows that we need to be able to have multiple successors of a single abstract state. For this, we use rules of the form

$$\frac{s}{s_1 \quad \ldots \quad s_n},$$

for some $n \in \mathbb{N}$, where s and s_1 through s_n are abstract states. The states s_1 through s_n are the successor states of s.

Similarly to the concrete rules, we define these rules such that their application is guided by the syntactical form of the first term of the first goal of the abstract state that is evaluated.

The abstract inference starts, similarly to the concrete one, with an initial state. This starting state is very simple to construct, as we have no knowledge about any properties of the variables so far.

The only major difference to the construction of a concrete starting state from a term is that we now allow term variables in the initial term. This enables us to specify starting states that represent an infinite number of concrete states. We call such an initial term with term variables a query.

Definition 4.7 (Query, initial state of abstract evaluation)**.** Let Σ be a signature, let $\mathcal{V}$ and $\mathcal{T}$ be disjoint sets of variables and let q be a term over Σ and $\mathcal{V} \cup \mathcal{T}$. We call q the **query**. We define the **initial state s^q_{init} of the abstract evaluation of** q as

$$s^q_{init} := (\langle q\,[(\mathcal{V} \cup \mathcal{T}) \mapsto (\mathcal{V} \cup \mathcal{T})_0]\rangle\,, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset),$$

where $(\mathcal{V} \cup \mathcal{T}) \mapsto (\mathcal{V} \cup \mathcal{T})_0$ denotes the function mapping each member X of $\mathcal{V} \cup \mathcal{T}$ to X_0. ■

We define rules that allow us to abstractly evaluate logic programs with cut in the following section. Afterwards, we consider arithmetic programs in Section 4.4.

4.3. Evaluation of Logic Programs

In this section we are going to define abstract evaluation rules that handle the evaluation of user-defined terms and the cut. These rules are taken from [GSSK$^+$12], where they were introduced as part of a termination analysis. We only reproduce those rules that contribute to the abstract semantics in this section. Those rules that were used specifically for termination analysis are presented in Section 5.1 *(Termination Graphs)*. Our own contribution is presented in Section 4.4, where we define new rules for the evaluation of arithmetic comparisons and the *is*-symbol.

Similarly to the concrete case, the simplest case is that in which the first goal of the abstract state is the empty goal. In that case, all concretizations of the abstract state also have the empty goal as their first goal. Since the only possible rule to apply in that case is Concrete Evaluation Rule 3.1, which removes the first goal from the state, we can do the same in the abstract state. This is formalized in Abstract Evaluation Rule 4.1. Since the removal of the empty goal does not have any influence on the knowledge about the variables in the state, we do not change the knowledge base of the abstract state.

Abstract Evaluation Rule 4.1 Success Rule

$$\text{Success}\,\frac{(\langle \square \mid G\rangle\,, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle G\rangle\,, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}$$

The next three rules are used to evaluate user-defined function symbols. We use the same idea for evaluation of these predicates as we did in the concrete case. This means we first use Abstract Evaluation Rule 4.2 to copy the goals and label each copy with the rule that we are going to try to apply. For this we use the same helper-function $\mathit{Slice}_P(t)$ as we did in the concrete case, which yields an ordered sequence of the rules in the program P that have the same function symbol at their head as t has.

The idea for the evaluation of such a labeled goal is to check whether or not the first term of the goal unifies with the head of the rule that the goal is labeled with. If this is the case, we apply the rule and continue with the goal. Otherwise, we abandon the goal

Abstract Evaluation Rule 4.2 Case Rule

$$\text{Case}\frac{(\langle (t,T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{\left(\left\langle (t,T)^{c'_1} \mid \ldots \mid (t,T)^{c'_n} \mid ?_\mathrm{m} \mid G\right\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}\right)}$$

Where: $\mathit{Slice}_P(t) = c_1, \ldots, c_n$
$c'_i := c_i\,[(\mathcal{V} \cup \mathcal{T}) \mapsto (\mathcal{V} \cup \mathcal{T})_m]\,[! \mapsto !_m]$
$m \in \mathbb{N}$ does not occur as an index in t, T, or G
$t = f(\ldots)$ and $f \notin \{=:=, =\backslash=, <, >, =<, >=, \mathit{is}\}$

and continue with the next one. This is the same procedure that we used in the concrete case. However, the question of whether or not two terms unify does not necessarily have a simple answer of *true* or *false* in the presence of term variables.

Example 4.8 (Unification with term variables)**.** Consider the two terms $t_1 = f(X)$ and $t_2 = f(g)$, where X is a term variable, and the concretization $\sigma_1 : X \mapsto g$. Then we have $t_1\sigma_1 = f(g)$, which obviously unifies with t_2 with the identity-function as the most general unifier.

Now consider the concretization $\sigma_2 : X \mapsto h$. We then have $t_1\sigma_2 = f(h)$, which does not unify with t_2.

Both σ_1 and σ_2 conform to the knowledge base $(\emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$. Thus, there exist some concretizations of the abstract state

$$s := (\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

that unify with t_2 and some that do not. ▲

Due to this, we need to create two successor states if we are not sure whether or not the first term of a labeled goal unifies with the head of the rule. This construction is formalized in Abstract Evaluation Rule 4.3.

It may be possible for us to infer that no concretization γ exists that allows $t\gamma$ and h to unify. This is the case, for example, if the pair (t, h) is a member of the set of nonunifying pairs of terms. In that case, we only need to produce a single successor state, namely that in which the two terms do not unify and the goal is abandoned. To do so, we use Abstract Evaluation Rule 4.4. Note that we do not add the pair (t, h) to $\mathcal{U}$ in this case, as this information is already stored in the abstract state.

Using the previously defined abstract evaluation rules, we can abstractly evaluate pure logic programs. We now reproduce rules from [GSSK+12] that allow us to evaluate the cut in abstract states.

The adaptation of the cut- and failure rules from the concrete semantics is very straightforward. Since neither ! nor ? can appear as subterms, but only as terms on their own, they can never be represented by a term variable. Hence, all cuts and end-of-scope markers that appear in any concretization of an abstract state also appear in the abstract state itself. Thus, we can simply evaluate them similarly to the concrete case. We do so in Abstract Evaluation Rule 4.5 and Abstract Evaluation Rule 4.6.

Abstract Evaluation Rule 4.3 Evaluation Rule

$$\text{Eval}\frac{\left(\left\langle (t,T)^{h \,:\!-\, B} \mid G\right\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}\right)}{(\langle (B\sigma, T\sigma) \mid G'\rangle, \mathcal{G}', \mathcal{U}', \mathcal{E}', \mathcal{A}') \quad (\langle G\rangle, \mathcal{G}, \mathcal{U}'', \mathcal{E}, \mathcal{A})}$$

Where: $\sigma := mgu(t,h) \neq \bot$
$G' := G$ after pointwise application of $(t_1, \ldots, t_n) \mapsto \left(t_1\sigma\big|_{\mathcal{G}}, \ldots, t_n\sigma\big|_{\mathcal{G}}\right)$
$\mathcal{G}' := \mathcal{T}(Range(\sigma|_{\mathcal{G}}))$
$\mathcal{U}' := \mathcal{U}\sigma\big|_{\mathcal{G}}$
$\mathcal{E}' := \mathcal{E}\sigma\big|_{\mathcal{G}}$
$\mathcal{A}' := \mathcal{A}\sigma\big|_{\mathcal{G}}$
$\mathcal{U}'' := \mathcal{U} \cup \{(t,h)\}$
$\exists\gamma$ conforming to $(\langle (t,T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ with $mgu(t\gamma, h) \neq \bot$

Abstract Evaluation Rule 4.4 Backtracking Rule

$$\text{Backtrack}\frac{\left(\left\langle (t,T)^{h \,:\!-\, B} \mid G\right\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}\right)}{(\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}$$

Where: $\nexists\gamma$ conforming to $(\langle (t,T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ with $mgu(t\gamma, h) \neq \bot$

We now use these abstract semantics to represent infinitely many possible executions of Program 2.1. This program is reprinted as Program 4.1 for the sake of readability.

Example 4.9 (Abstract Evaluation of Program 4.1 on *ancestor*(X,Y))**.** Consider Program 4.1, which we call P, and the query $q := ancestor(X,Y)$, where X and Y are term variables. We abstractly evaluate q on P. The initial fragment of the infinite tree of abstract states is shown in Figure 4.1.

According to Definition 4.7, the initial state of this inference is

$$s^q_{init} := (\langle ancestor(X,Y)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset),$$

which is present in the graph as node A. Since the function symbol *ancestor* is user-defined, the only applicable rule is Abstract Evaluation Rule 4.2. We apply this rule and receive state B. This application is the same as in the concrete case.

Abstract Evaluation Rule 4.5 Cut Rule

$$\text{Cut}\frac{(\langle (!_m, T) \mid G \mid ?_m \mid G'\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle T \mid ?_m \mid G'\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}$$

Where: G does not contain $?_m$

Abstract Evaluation Rule 4.6 Failure Rule

$$\text{Failure}\frac{(\langle ?_m \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}$$

Program 4.1 Family relationships (reprint of Program 2.1 on page 8)

```
female(alice).                      male(bob).
female(claire).                     female(diane).
father(bob, alice).                 mother(claire, bob).
mother(diane, claire).
ancestor(X, Y) :- mother(X, Y).
ancestor(X, Y) :- ancestor(X, Z), mother(Z, Y).
```

When we try to evaluate node B, we see that $\mathit{ancestor}(X, Y)$ and $\mathit{ancestor}(\mathtt{X}, \mathtt{Y})$ unify in any case. However, there does not exist a rule that produces only a successor state in which the unification succeeds. Hence, we have to produce one successor state for both the case in which the unification succeeds and the one in which it does not. We apply Abstract Evaluation Rule 4.3 and receive the two successor states C and D.

We do not show the remaining evaluation of node C, as we would just apply Abstract Evaluation Rule 4.2 again here. Eventually, we would finish evaluating the first goal of this node, and continue with the second one. Here, we would again have one abstract successor state in which $\mathit{ancestor}(X, Y)$ and $\mathit{ancestor}(\mathtt{X}, \mathtt{Y})$ unify, which would be very similar to state A. Thus, there would be an infinite sequence of nodes. This sequence corresponds to the infinite evaluation of the query as shown in Figure 2.1 on page 11.

While node C represents the case in which the unification succeeded, node D represents the case in which the unification failed. Note that the knowledge base of node D has changed in comparison to that of node B, as we added the nonunifying pair $(\mathit{ancestor}(X, Y), \mathit{ancestor}(\mathtt{X}, \mathtt{Y}))$. Since we now have this knowledge in our knowledge base, we do not need to perform a case analysis for the evaluation of node D. Instead, we only need to consider the case in which $\mathit{ancestor}(X, Y)$ and $\mathit{ancestor}(\mathtt{X}, \mathtt{Y})$ do not unify, which allows us to abandon the current goal and continue with the only remaining one, namely $?_1$.

Note that we reached an unsatisfiable knowledge base in node D, meaning that the abstract state

$$(\left\langle \mathit{ancestor}(X, Y)^{\mathit{ancestor}(\mathtt{X},\mathtt{Y})\ \texttt{:-}\ \mathit{ancestor}(\mathtt{X},\mathtt{Z}), \mathit{ancestor}(\mathtt{Z},\mathtt{Y})} \mid ?_1 \right\rangle, \\ \emptyset, \{(\mathit{ancestor}(X, Y), \mathit{ancestor}(\mathtt{X}, \mathtt{Y}))\}, \emptyset, \mathcal{A}_\emptyset)$$

does not represent any concrete states. However, the rules of the abstract allow for further evaluation of nodes D and E.

We can evaluate node E using Abstract Evaluation Rule 4.6, at which point we end up with an empty state form. At this point the evaluation is finished. ▲

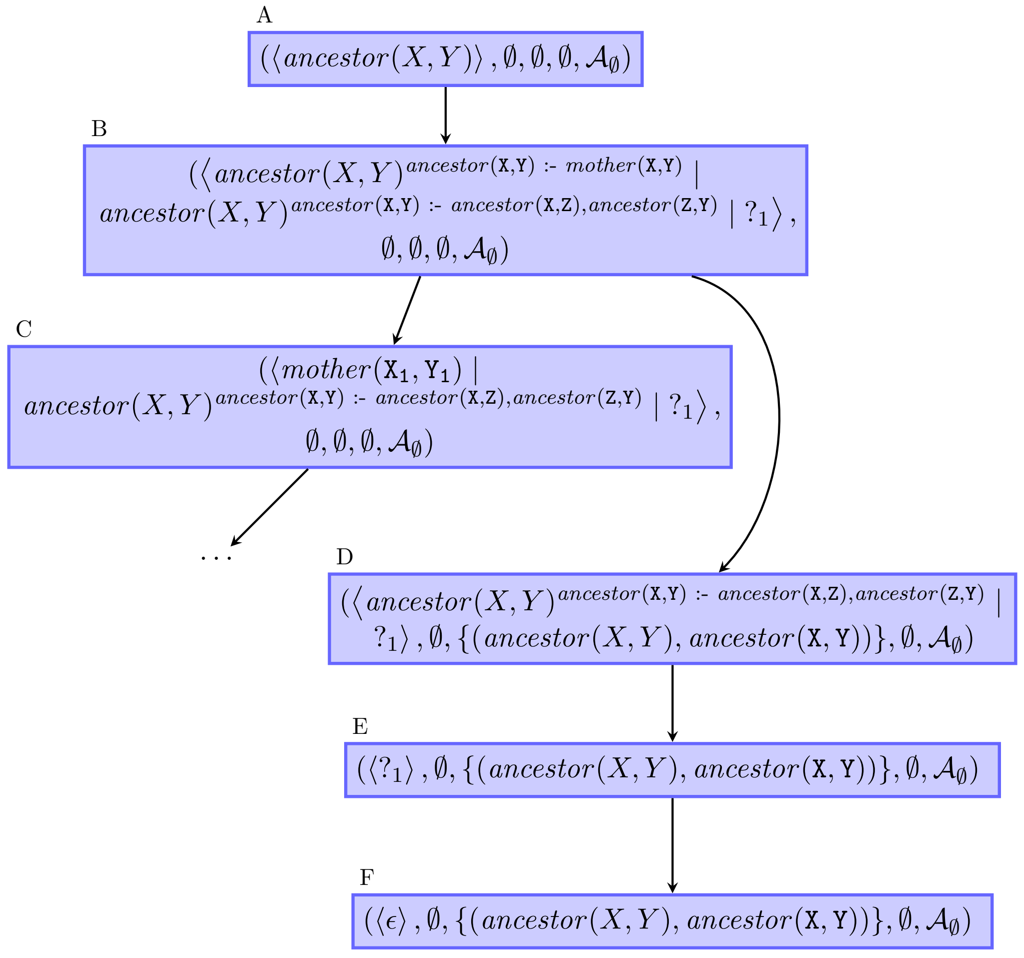

A
$(\langle ancestor(X,Y)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
B
$(\langle ancestor(X,Y)^{ancestor(\mathtt{X,Y})\ \text{:-}\ mother(\mathtt{X,Y})} \mid ancestor(X,Y)^{ancestor(\mathtt{X,Y})\ \text{:-}\ ancestor(\mathtt{X,Z}),ancestor(\mathtt{Z,Y})} \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
C
$(\langle mother(\mathtt{X_1}, \mathtt{Y_1}) \mid ancestor(X,Y)^{ancestor(\mathtt{X,Y})\ \text{:-}\ ancestor(\mathtt{X,Z}),ancestor(\mathtt{Z,Y})} \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
D
$(\langle ancestor(X,Y)^{ancestor(\mathtt{X,Y})\ \text{:-}\ ancestor(\mathtt{X,Z}),ancestor(\mathtt{Z,Y})} \mid ?_1\rangle, \emptyset, \{(ancestor(X,Y), ancestor(\mathtt{X},\mathtt{Y}))\}, \emptyset, \mathcal{A}_\emptyset)$
E
$(\langle ?_1\rangle, \emptyset, \{(ancestor(X,Y), ancestor(\mathtt{X},\mathtt{Y}))\}, \emptyset, \mathcal{A}_\emptyset)$
F
$(\langle \epsilon\rangle, \emptyset, \{(ancestor(X,Y), ancestor(\mathtt{X},\mathtt{Y}))\}, \emptyset, \mathcal{A}_\emptyset)$

Figure 4.1.: Abstract evaluation of $ancestor(X, Y)$ on Program 4.1

In the previous example we have seen that we can indeed model the infinite evaluation of the query that we have shown in Example 2.7 with our abstract semantics. More importantly, we are able to model any concrete evaluation using this abstract semantics. This property of an abstract semantics is usually known as soundness, meaning that the abstract evaluation does not "lose" any states. More specifically, our abstract semantics has the property that every rule produces abstract successor states that "cover" all the possible concrete successor states of the original abstract state.

Definition 4.8 (Soundness of abstract interpretation)**.** Let

$$r := \frac{s}{s_1 \quad \dots \quad s_n}$$

be an abstract evaluation rule. We say that r is **sound** iff the following property holds:

For all $\mathfrak{s} \in Conc(s), \mathfrak{s}' \in ConcreteStates$ it holds true that
$\mathfrak{s} \to \mathfrak{s}'$ implies that there exists some $i \in [1; n]$ such that $\mathfrak{s}' \in Conc(s_i)$

■

The soundness of these rules has been shown in [GSSK+12]. We merely state this fact here and point to that work for the full proof.

Lemma 4.2 (Soundness of abstract interpretation (Logic))**.** *The abstract evaluation rules 4.1 through 4.6 are sound.* ■

Proof. The full proof of this can be found in [GSSK+12, Appendix A]. □

4.4. Evaluation of Arithmetic Logic Programs

In the previous section, we have reproduced prior work published in [GSSK+12]. The abstract evaluation rules in that work can be used to abstractly evaluate logic programs with cuts. The authors do, however, not provide rules to handle the built-in arithmetic predicates and functions of PROLOG used for integer arithmetic. We define such rules in this section.

Since there is the possibility for every arithmetic evaluation and comparison to lead to an error state, we first develop a sound heuristic to check for the possibility of such an error in Section 4.4.1. In Section 4.4.2 we then define rules for the evaluation of arithmetic comparisons. Finally, we define rules to handle the abstract evaluation of the *is*-predicate in Section 4.4.3.

4.4.1. Safe Evaluation of Arithmetic Expressions

The evaluation of an arithmetic expression can introduce implementation-defined behavior into the inference. In our semantics, we treat this behavior as an error. For any term without term variables, it is easy to decide whether or not the evaluation of the term results in an error by checking whether or not $eval_E(t) = \bot$ holds true. In the presence of term variables, however, this check is not so simple anymore and does not even lead to a simple result of *true* or *false*.

Example 4.10 (Unsafe evaluation with term variables)**.** Consider the abstract state

$$s := (\langle //(3, X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset),$$

where X is a term variable. The conforming substitution $\sigma_1 := X \mapsto 1$ concretizes $//(3, X)$ to an expression whose evaluation results in the natural number 3. Another substitution, for example $\sigma_2 := X \mapsto f$ concretizes $//(3, X)$ to a term whose evaluation results in an error, as f is not part of the predefined arithmetic functions. The final substitution $\sigma_3 := X \mapsto 0$ also leads to the evaluation of $//(3, X)\sigma_3$ resulting in an error, as division by zero is not defined. ▲

We thus see that, depending on the position and the context of the term variables, we may be certain that the evaluation of some term leads to an error, or we may be certain that it does not. In order to decide this precisely, we require a set of term variables $\mathcal{E}$ that we know not to be instantiated to any terms whose evaluation results in an error, as well as an arithmetic state $\mathcal{A}$ that allows us to detect potential divisions by zero. These two components are part of the knowledge base of abstract states, so we can easily use them during abstract evaluation. We define the helper function $\mathit{safe}_{\mathcal{E},\mathcal{A}} : \mathit{Terms} \to \{\mathit{true}, \mathit{false}, \mathit{maybe}\}$ to check whether or not a term may evaluate to an error.

This safety heuristic is applied recursively to the operands of built-in arithmetic functions. We use the operator $\otimes$ to combine the results of the heuristic on the arguments to the functions.

Definition 4.9 (Safety heuristic)**.** Let $\otimes$ be a binary operator on $\{\mathit{true}, \mathit{false}, \mathit{maybe}\}$, defined as follows:

$$\begin{array}{ccc}
\mathit{true} \otimes \mathit{true} = \mathit{true} & \mathit{true} \otimes \mathit{maybe} = \mathit{maybe} & \mathit{true} \otimes \mathit{false} = \mathit{false} \\
\mathit{maybe} \otimes \mathit{true} = \mathit{maybe} & \mathit{maybe} \otimes \mathit{maybe} = \mathit{maybe} & \mathit{maybe} \otimes \mathit{false} = \mathit{false} \\
\mathit{false} \otimes \mathit{true} = \mathit{false} & \mathit{false} \otimes \mathit{maybe} = \mathit{false} & \mathit{false} \otimes \mathit{false} = \mathit{false}
\end{array}$$

Let $\mathcal{E}$ be a set of variables and let $\mathcal{A}$ be an arithmetic state. We define $\mathit{safe}_{\mathcal{E},\mathcal{A}} : \mathit{Terms} \to \{\mathit{true}, \mathit{false}, \mathit{maybe}\}$ inductively over the structure of a term. The function is defined on atoms as follows:

$$\begin{array}{ll}
\mathit{safe}_{\mathcal{E},\mathcal{A}}(n) := \mathit{true} & \text{if } n \in \mathbb{Z} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathtt{X}) := \mathit{false} & \text{if } \mathtt{X} \text{ is a program variable} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(X) := \mathit{true} & \text{if } X \text{ is a term variable and } X \in \mathcal{E} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(X) := \mathit{maybe} & \text{if } X \text{ is a term variable and } X \notin \mathcal{E}
\end{array}$$

It is defined for compound terms as follows:

$$\begin{array}{ll}
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathit{op}(t')) := \mathit{safe}_{\mathcal{E},\mathcal{A}}(t') & \text{if } \mathit{op} \in \{\mathit{abs}, \mathit{sign}, -\} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathit{op}(t'_1, t'_2)) := \mathit{safe}_{\mathcal{E},\mathcal{A}}(t'_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t'_2) & \text{if } \mathit{op} \in \{+, -, *, **\} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathit{op}(t'_1, t'_2)) := \mathit{safe}_{\mathcal{E},\mathcal{A}}(t'_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t'_2) & \begin{array}{l}\text{if } \mathit{op} \in \{//, \mathit{mod}, \mathit{rem}\} \\ \text{and } \mathcal{A} \models t'_2 \neq 0\end{array} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathit{op}(t'_1, t'_2)) := \mathit{false} & \begin{array}{l}\text{if } \mathit{op} \in \{//, \mathit{mod}, \mathit{rem}\} \\ \text{and } \mathcal{A} \models t'_2 = 0\end{array} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(\mathit{op}(t'_1, t'_2)) := \mathit{maybe} & \begin{array}{l}\text{if } \mathit{op} \in \{//, \mathit{mod}, \mathit{rem}\} \\ \text{and } \mathcal{A} \not\models t'_2 = 0 \\ \text{and } \mathcal{A} \not\models t'_2 \neq 0\end{array} \\
\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) := \mathit{false} & \text{otherwise}
\end{array}$$

■

This function yields *true* for any term that will definitely not lead to an error during evaluation and *false* for any term that will inevitably lead to an error during evaluation. It returns *maybe* for all other terms. We test this function on our previous example.

Example 4.11 (Application of $\mathit{safe}_{\mathcal{E},\mathcal{A}}$ to Example 4.10)**.** Let $t := //(3, X)$, where X is a term variable. We first consider $\mathcal{E}_1 := \{X\}$ and the arithmetic state $\mathcal{A}_1 := \mathcal{A}_{\{X \neq 0\}}$. In this case we have

$$\mathit{safe}_{\mathcal{E}_1,\mathcal{A}_1}(t) = \mathit{safe}_{\mathcal{E}_1,\mathcal{A}_1}(3) \otimes \mathit{safe}_{\mathcal{E}_1,\mathcal{A}_1}(X) = \mathit{true} \otimes \mathit{true} = \mathit{true},$$

since $\mathcal{A}_1 \models X \neq 0$ and $X \in \mathcal{E}_1$.

We now pick $\mathcal{E}_2 = \emptyset$. The function then yields

$$\mathit{safe}_{\mathcal{E}_2,\mathcal{A}_1}(t) = \mathit{safe}_{\mathcal{E}_2,\mathcal{A}_1}(3) \otimes \mathit{safe}_{\mathcal{E}_2,\mathcal{A}_1}(X) = \mathit{true} \otimes \mathit{maybe} = \mathit{maybe}$$

As a final example we choose $\mathcal{A}_2 := \mathcal{A}_{\{X=0\}}$. In this case $\mathit{safe}_{\mathcal{E}_2,\mathcal{A}_2}(t)$ evaluates to

$$\mathit{safe}_{\mathcal{E}_2,\mathcal{A}_2}(t) = \mathit{false},$$

since division by zero leads to an error. ▲

We see that the function works as intended on this example. In fact, we show that this function works as intended on all terms in the following lemma.

Lemma 4.3 (Soundness of safety heuristic)**.** *Let $s = (\langle(t, T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. Then the following two implications hold true:*

$$\begin{aligned}&\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true} \textbf{ implies that}\\ &\quad\textbf{for all concretizations } \gamma \textbf{ conforming to } s \textbf{ it holds true that } \mathit{eval}_E(t\gamma) \neq \bot\end{aligned} \tag{4.1}$$

$$\begin{aligned}&\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{false} \textbf{ implies that}\\ &\quad\textbf{for all concretizations } \gamma \textbf{ conforming to } s \textbf{ it holds true that } \mathit{eval}_E(t\gamma) = \bot\end{aligned} \tag{4.2}$$

■

Proof. We show both statements separately and begin with statement 4.1.

Proof of statement 4.1

Let $s = (\langle(t, T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be some abstract state and let γ be some concretization conforming to s. Assume $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true}$. We show $\mathit{eval}_E(t\gamma) \neq \bot$ by structural induction over t.

Induction base: t has no arguments Since we assumed that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true}$ holds true, there are two cases to consider: The term t may be an integer, or it may be a term variable. The cases that t is a program variable or some user-defined constant would contradict our assumption.

Case 1: $t = n$ is an integer In this case, $eval_E(t) = n$ holds true. Since n is a natural number, $eval_E(t) \neq \bot$, whence the statement holds true.

Case 2: $t = X$ is a term variable In this case, due to our assumption, $X \in \mathcal{E}$ holds true. Since γ conforms to s, we know by the definition of conforming substitutions that $eval_T(X\gamma) \neq \bot$ holds true.

Induction step: t has arguments Let $t = op(t_1, \ldots, t_n)$ and assume that $safe_{\mathcal{E},\mathcal{A}}(t_i) = true$ implies $eval_E(t_i\gamma) \neq \bot$ for all $1 \leq i \leq n$. Since we have $safe_{\mathcal{E},\mathcal{A}}(t) = true$ by assumption, we know that $n \leq 2$, otherwise this assumption would be false. We distinguish two cases.

Case 1: $t = op(t')$ Since our assumption was that $safe_{\mathcal{E},\mathcal{A}}(t) = true$ holds true, we know that $op \in \{abs, sign, -\}$ and that $safe_{\mathcal{E},\mathcal{A}}(t') = true$. We apply the induction hypothesis and have $eval_E(t'\gamma) = n \neq \bot$. We can then see that $eval_E(t\gamma) = op(n)$, which is well-defined in all three possible cases for op. Hence, we have $eval_E(t\gamma) \neq \bot$.

Case 2: $t = op(t'_1, t'_2)$ Due to our assumption we have $safe_{\mathcal{E},\mathcal{A}}(t'_1) = safe_{\mathcal{E},\mathcal{A}}(t'_2) = true$ and we know that either $op \in \{+, -, *, **\}$ holds true, or $op \in \{//, mod, rem\}$ and $\mathcal{A} \models t'_2 \neq 0$ do. In both cases, we can evaluate $eval_E(t\gamma)$ to $eval_E(t'_1\gamma)\ op\ eval_E(t'_2\gamma)$, apply the induction hypothesis to $eval_E(t'_1\gamma) = n_1$ and $eval_E(t'_2\gamma) = n_2$ and receive $eval_E(t\gamma) = n_1\ op\ n_2$.

In the former case, we see that this expression is well defined for all $n_1, n_2 \in \mathbb{Z}$ and all $op \in \{+, -, *, **\}$. In the latter case, we again use that γ conforms to s. Since $\mathcal{A} \models t'_2 \neq 0$, we see that $t'_2\gamma \neq 0$ is a tautology, due to the definition of conforming substitutions. Hence, $eval_E(t'_1\gamma)\ op\ eval_E(t'_2\gamma)$ is well defined in this case as well.

We have thus shown that $safe_{\mathcal{E},\mathcal{A}}(t) = true$ implies $eval_E(t\gamma) \neq \bot$ for all conforming concretizations γ. Hence, statement 4.1 holds true.

Proof of statement 4.2

Let $s = (\langle (t, T) \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be some abstract state and let γ be some concretization conforming to s. Assume $safe_{\mathcal{E},\mathcal{A}}(t) = false$. We show $eval_E(t\gamma) = \bot$ by structural induction over t.

Induction base: t has no arguments In this case, since $safe_{\mathcal{E},\mathcal{A}}(t) = false$, we must have $t = \mathtt{X}$, where $\mathtt{X}$ is a program variable. We can then directly see that $eval_E(t\gamma) = \bot$, since $t\gamma = t$, as γ does not substitute program variables.

Induction step: t has arguments For the induction step we have to distinguish five cases, corresponding to the five possible ways that $safe_{\mathcal{E},\mathcal{A}}(t)$ may yield *false*.

Case 1: $t = op(t')$ and $op \in \{abs, sign, -\}$ In this case we have, by definition, $safe_{\mathcal{E},\mathcal{A}}(t) = safe_{\mathcal{E},\mathcal{A}}(t')$. Hence, by assumption, we have $safe_{\mathcal{E},\mathcal{A}}(t') = false$. We can apply the induction hypothesis to t' and receive $eval_E(t'\gamma) = \bot$. Thus, we receive $eval_E(t\gamma) = \bot$.

Case 2: $t = op(t'_1, t'_2)$ and $op \in \{+, -, *, **\}$ In this case we have $safe_{\mathcal{E},\mathcal{A}}(t) = safe_{\mathcal{E},\mathcal{A}}(t'_1) \otimes safe_{\mathcal{E},\mathcal{A}}(t'_2) = false$. Due to the definition of $\otimes$, we know that either $safe_{\mathcal{E},\mathcal{A}}(t'_1) = false$ or $safe_{\mathcal{E},\mathcal{A}}(t'_2) = false$. W.l.o.g. assume $safe_{\mathcal{E},\mathcal{A}}(t'_1) = false$. We can then apply the induction hypothesis to t'_1 and receive $eval_E(t'_1\gamma) = \bot$ and hence $eval_E(t\gamma) = \bot$.

Case 3: $t = op(t'_1, t'_2)$, $op \in \{//, op, rem\}$ and $\mathcal{A} \models t'_2 \neq 0$ This case is handled the same way as case 2. Hence, we omit the proof to avoid unnecessary repetition.

Case 4: $t = op(t'_1, t'_2)$, $op \in \{//, op, rem\}$ and $\mathcal{A} \models t'_2 = 0$ Since $\mathcal{A} \models t'_2 = 0$ we know that $eval_E(t'_2\gamma) = 0$ holds true. Hence, we have $eval_E(op(t'_1, t'_2)\gamma) = \bot$ due to the definition of $eval_E$.

Case 5: otherwise Since t does not start with the symbol of any built-in function, we immediately have $eval_E(t\gamma) = \bot$ due to the definition of $eval_E$.

We have thus shown that $safe_{\mathcal{E},\mathcal{A}}(t) = false$ implies $eval_E(t\gamma) = \bot$ for all conforming concretizations γ. Hence, statement 4.2 holds true.

Conclusion

We have shown that both statement 4.1 and 4.2 hold true. Hence, lemma 4.3 holds true. □

We make one more observation that allows us to keep unnecessary updates to a state during evaluation to a minimum. This observation states that, if we are certain that a term evaluates without an error, then we also know that all its term variables can only be instantiated with arithmetic expressions that do not evaluate to an error.

Lemma 4.4. *Let t be a term, let $\mathcal{E}$ be a set of variables and let $\mathcal{A}$ be an arithmetic state. Then the statement*

$$safe_{\mathcal{E},\mathcal{A}}(t) = true \textbf{ implies } \mathcal{T}(t) \subseteq \mathcal{E}$$

holds true. ■

Proof. We show this statement by structural induction over the term t. Assume that $safe_{\mathcal{E},\mathcal{A}}(t) = true$ holds true

Induction base: t has no arguments Since we assumed that $safe_{\mathcal{E},\mathcal{A}}(t) = true$ holds true, there are two cases to consider: The term t is either an integer or a term variable. If t is a program variable or some user-defined constant, we would have a direct contradiction to our assumption.

Case 1: $t = n$ is an integer In this case, $\mathcal{T}(t) = \emptyset$ holds true and hence $\mathcal{T}(t) \subseteq \mathcal{V}$ does as well, whence the statement holds true.

Case 2: $t = X$ is a term variable In this case, due to our assumption, $X \in \mathcal{E}$ holds true. Hence, we have $\mathcal{T}(t) = \{X\} \subseteq \mathcal{E}$ and the statement to be shown holds true.

Induction step: t has arguments Let $t = f(t'_1, \dots, t'_n)$ and assume that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_i) = \mathit{true}$ implies $\mathcal{T}(t_i) \subseteq \mathcal{E}$ for all $i \in [1;n]$. Furthermore assume that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true}$ holds true. It is easy to see from the definition of $\mathit{safe}_{\mathcal{E},\mathcal{A}}$ that due to $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true}$, $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_i) = \mathit{true}$ must also hold true for all $i \in [1;n]$. Hence, we can apply the induction hypothesis and receive $\mathcal{T}(t_i) \subseteq \mathcal{E}$ for all $i \in [1;n]$. Since $\mathcal{T}(t) = \cup_{i=1}^{n} \mathcal{T}(t_i)$, we see that $\mathcal{T}(t) \subseteq \mathcal{E}$ holds true.

Conclusion We have shown that the statement

$$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t) = \mathit{true} \textbf{ implies } \mathcal{T}(t) \subseteq \mathcal{E}$$

holds true by structural induction over the structure of t. □

Using this heuristic for safe evaluation of terms we can now concisely define abstract evaluation rules that allow us to evaluate arithmetic comparisons and assignments. We do so in the following section.

4.4.2. Evaluation of Arithmetic Comparison

In this section we define abstract evaluation rules for abstract states that have the form $(\langle(\bowtie(t_1, t_2), T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$, where $\bowtie$ is one of =:=, =\=, <, >, =< and >=. We treat the *is*-predicate in the following section.

There are three possible outcomes of the evaluation of a concretization of such a state. It may be the case that the evaluation leads to an error, for example due to a division by zero. This corresponds to Concrete Evaluation Rule 3.7. The other two possible cases are those in which the evaluation of terms succeeds and the comparison between the integers succeeds or fails, respectively. These cases were treated in the concrete case with Concrete Evaluation Rule 3.8 and Concrete Evaluation Rule 3.9.

In the abstract case it may be possible that we cannot tell precisely which of these three cases occurs, as we see in the following example.

Example 4.12 (Arithmetic Comparison in an Abstract State)**.** Consider the abstract state

$$s := (\langle >(X, 0), f \mid g\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

and the three concretizations

$$\sigma_1 := X \mapsto //(1, 0) \qquad \sigma_2 := X \mapsto 1 \qquad \sigma_3 := X \mapsto -1$$

all of which conform to s. Hence, the three concrete states

$$\begin{aligned}\mathfrak{s}_1 &:= \langle {>}(X,0), f \mid g\rangle \sigma_1 = \langle {>}(//(1,0),0), f \mid g\rangle\\ \mathfrak{s}_2 &:= \langle {>}(X,0), f \mid g\rangle \sigma_2 = \langle {>}(1,0), f \mid g\rangle\\ \mathfrak{s}_3 &:= \langle {>}(X,0), f \mid g\rangle \sigma_3 = \langle {>}(-1,0), f \mid g\rangle\end{aligned}$$

are all concretizations of $\mathfrak{s}$. We see that we have to use Concrete Evaluation Rule 3.7 in order to evaluate $\mathfrak{s}_1$, which results in the concrete state

$$\mathfrak{s}_1' = \langle \mathit{Err}\rangle\,.$$

Furthermore, we can only evaluate $\mathfrak{s}_2$ using Concrete Evaluation Rule 3.8, yielding the successor state

$$\mathfrak{s}_2' = \langle f \mid g\rangle\,.$$

The only rule applicable to $\mathfrak{s}_3$ is Concrete Evaluation Rule 3.9, which results in the state

$$\mathfrak{s}_3' = \langle g\rangle\,.$$

Hence, the abstract successor states of s must cover all three possible outcomes of this comparison. ▲

The previous example has shown the worst possible case for our semantics, which is one in which we are unable to rule out any of the three outcomes of arithmetic comparison. In some cases, however, it is possible to rule out one or two of these cases, which reduces the number of states produced by our semantics. We first present the three rules for the cases in which we can remove two of the three possible outcomes and only need a single abstract successor state.

The easiest case is the one in which the evaluation of either of the two expressions inevitably fails. We use the previously defined function $\mathit{safe}_{\mathcal{E},\mathcal{A}}$ in order to characterize such states. Recall that $\mathit{safe}_{\mathcal{E},\mathcal{A}}$ takes the set of variables that are known to be substituted with expressions that evaluate without an error, which is stored in our abstract state in the set $\mathcal{E}$. It also takes the arithmetic state $\mathcal{A}$ of the abstract state as a parameter. This rule is formalized in Abstract Evaluation Rule 4.7.

Abstract Evaluation Rule 4.7 Arithmetic Comparison Rule (Error)

$$\text{ArithCompErr}\,\frac{(\langle(\bowtie(t_1,t_2),T)\mid G\rangle\,,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle \mathit{Err}\rangle\,,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\bowtie\, \in \{=:=, =\backslash=, <, >, =<, >=\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$

The other two cases in which we only produce a single successor state are very similar to each other. For both cases we first need to make sure that the evaluation of the expressions on either side does not result in an error. We do so by demanding that both $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1') \equiv \mathit{true}$ and $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2') \equiv \mathit{true}$ hold true.

If we can then infer that either $\mathcal{A} \models (t_1 \bowtie t_2)$ or $\mathcal{A} \models \neg\,(t_1 \bowtie t_2)$ holds true, we only need to produce a single successor state in which the comparison either succeeds or fails. These ideas are formalized in Abstract Evaluation Rule 4.8 and Abstract Evaluation Rule 4.9, respectively.

Note that we do not update the set $\mathcal{E}$, since this would only add redundant information, as both $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1)$ and $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2)$ evaluate to true already. This has been shown in Lemma 4.4.

Abstract Evaluation Rule 4.8 Arithmetic Comparison Rule (Success)

$$\text{ArithCompSucc}\,\frac{(\langle(\bowtie(t_1,t_2),T) \mid G\rangle\,,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle T \mid G\rangle\,,\mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$
$\bowtie\, \in \{\texttt{=:=}, \texttt{=\textbackslash=}, \texttt{<}, \texttt{>}, \texttt{=<}, \texttt{>=}\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$\mathcal{A} \models (t_1 \bowtie t_2)$

Abstract Evaluation Rule 4.9 Arithmetic Comparison Rule (Failure)

$$\text{ArithCompFail}\,\frac{(\langle(\bowtie(t_1,t_2),T) \mid G\rangle\,,\mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle G\rangle\,,\mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$
$\bowtie\, \in \{\texttt{=:=}, \texttt{=\textbackslash=}, \texttt{<}, \texttt{>}, \texttt{=<}, \texttt{>=}\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$\mathcal{A} \models \neg\,(t_1 \bowtie t_2)$

Having covered those cases in which a single successor state suffices to represent all concrete successor states, we now formalize those cases in which we need two abstract successor states. It may be the case that we are certain that the evaluation of both arguments of $\bowtie(t_1,t_2)$ succeeds without error, but we do not know the result of the comparison. In this case we do not need to produce an error state, but we receive two successor states, modeling success and failure of the comparison, respectively.

This rule is given as Abstract Evaluation Rule 4.10. Note that we update the arithmetic state of the successor state to contain the assumption that we made in either case. This allows us to keep our arithmetic inference precise, as this knowledge may be used for other arithmetic comparisons later on. We do not, however, update the set of error-free expressions $\mathcal{E}$, as we are already certain that the evaluation of t_1 and t_2 succeeds. We will show later that these additions to the knowledge base do not influence the soundness of our semantics and that they are nothing but a case analysis.

Another possible case is that we are uncertain whether or not the evaluation of the expressions on both sides will succeed, but we know the result of the comparison if it does. Example for such case would be the comparisons =:=(X,X) and =\=(X,X), where X

Abstract Evaluation Rule 4.10 Arithmetic Comparison Rule (Success, Failure)

$$\text{ArithCompSuccFail}\frac{(\langle(\bowtie(t_1,t_2),T)\mid G\rangle,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle T\mid G\rangle,\mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A}')\quad(\langle G\rangle,\mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A}'')}$$

Where:
$\mathcal{G}' := \mathcal{G}\cup\mathcal{T}(t_1)\cup\mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A}\cap\mathcal{A}_{\{t_1\bowtie t_2\}}$
$\mathcal{A}'' := \mathcal{A}\cap\mathcal{A}_{\{\neg(t_1\bowtie t_2)\}}$
$\bowtie\,\in\{=:=,=\backslash=,<,>,=<,>=\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1)\otimes\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2)\equiv\mathit{true}$
$\mathcal{A}\not\models(t_1\bowtie t_2)$ **and** $\mathcal{A}\not\models\neg(t_1\bowtie t_2)$

is a term variable. In order to cover these cases, we introduce Abstract Evaluation Rule 4.11 and Abstract Evaluation Rule 4.12, which formalize success and failure of such an unsafe comparison, respectively. We do not update the arithmetic states in these rules, since we are already able to infer the truth of the relevant comparison. We do, however, update the set of arithmetic variables in the corresponding case.

Abstract Evaluation Rule 4.11 Arithmetic Comparison Rule (Error, Success)

$$\text{ArithCompErrSucc}\frac{(\langle(\bowtie(t_1,t_2),T)\mid G\rangle,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle \mathit{Err}\rangle,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})\quad(\langle T\mid G\rangle,\mathcal{G}',\mathcal{U},\mathcal{E}',\mathcal{A})}$$

Where:
$\mathcal{G}' := \mathcal{G}\cup\mathcal{T}(t_1)\cup\mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E}\cup\mathcal{T}(t_1)\cup\mathcal{T}(t_2)$
$\bowtie\,\in\{=:=,=\backslash=,<,>,=<,>=\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1)\otimes\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2)\equiv\mathit{maybe}$
$\mathcal{A}\models(t_1\bowtie t_2)$

Abstract Evaluation Rule 4.12 Arithmetic Comparison Rule (Error, Failure)

$$\text{ArithCompErrFail}\frac{(\langle(\bowtie(t_1,t_2),T)\mid G\rangle,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle \mathit{Err}\rangle,\mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})\quad(\langle G\rangle,\mathcal{G}',\mathcal{U},\mathcal{E}',\mathcal{A})}$$

Where:
$\mathcal{G}' := \mathcal{G}\cup\mathcal{T}(t_1)\cup\mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E}\cup\mathcal{T}(t_1)\cup\mathcal{T}(t_2)$
$\bowtie\,\in\{=:=,=\backslash=,<,>,=<,>=\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1)\otimes\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2)\equiv\mathit{maybe}$
$\mathcal{A}\models\neg(t_1\bowtie t_2)$

The least precise case, that is, the one that produces the most successor states, is the case in which we neither know whether or not the evaluation of the compared expressions results in an error, nor what result the evaluation of the relation has. In this case, we have to produce the error state for the case that the evaluation of the expressions fails as well as one case for the success and the failure of the comparison each. For this case we introduce Abstract Evaluation Rule 4.13. In this rule we update both the arithmetic states of the resulting states as well as their arithmetic variables.

Abstract Evaluation Rule 4.13 Arithmetic Comparison (Error, Success, Failure)

$$\text{ArithCompErrSuccFail}\frac{(\langle(\bowtie(t_1,t_2),T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle Err\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \quad (\langle T \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}') \quad (\langle G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}'')}$$

Where:
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{t_1 \bowtie t_2\}}$
$\mathcal{A}'' := \mathcal{A} \cap \mathcal{A}_{\{\neg(t_1 \bowtie t_2)\}}$
$\bowtie \in \{=:=, =\backslash=, <, >, =<, >=\}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$
$\mathcal{A} \not\models (t_1 \bowtie t_2)$ **and** $\mathcal{A} \not\models \neg(t_1 \bowtie t_2)$

These rules cover all combinations of the three possible successor states of an arithmetic comparison. While Abstract Evaluation Rule 4.13 allows us to produce the widest range of successor states, it would yield abstract states without any concretizations if applied indiscriminately. The presence of the other rules allows us to only produce states that actually represent concrete states.

Example 4.13 (Abstract evaluation of arithmetic comparisons)**.** Consider Program 2.2 from page 12, which we reprint as Program 4.2 for the sake of readability. Also consider the query $q := \mathit{fac}(X, Y)$. We show the relevant parts of the tree of abstract states induced by this query on Program 4.2 in Figure 4.2.

Program 4.2 Computation of the factorial

```
fac(X, Y) :- X > 0, fac(X - 1, Y1), Y is Y1 * X.
fac(X, Y) :- X =:= 0, Y is 1.
```

As argued before, the inference starts in the abstract state

$$s^q_{\mathit{init}} := (\langle \mathit{fac}(X_0, Y_0)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

This state is represented by node A in the tree. We start the evaluation similar to the concrete case, that is, by application of Abstract Evaluation Rule 4.2 and Abstract Evaluation Rule 4.3. The intermediate state is not shown in the figure. We end up, among others, with node B, in which we now have to apply one of the rules for arithmetic comparisons.

Since $\mathit{safe}_{\emptyset,\emptyset}(X) = \mathit{maybe}$ holds true and since both $\emptyset \not\models X > 0$ and $\emptyset \not\models X \leq 0$ hold true, we apply Abstract Evaluation Rule 4.13, which yields three successor states for the

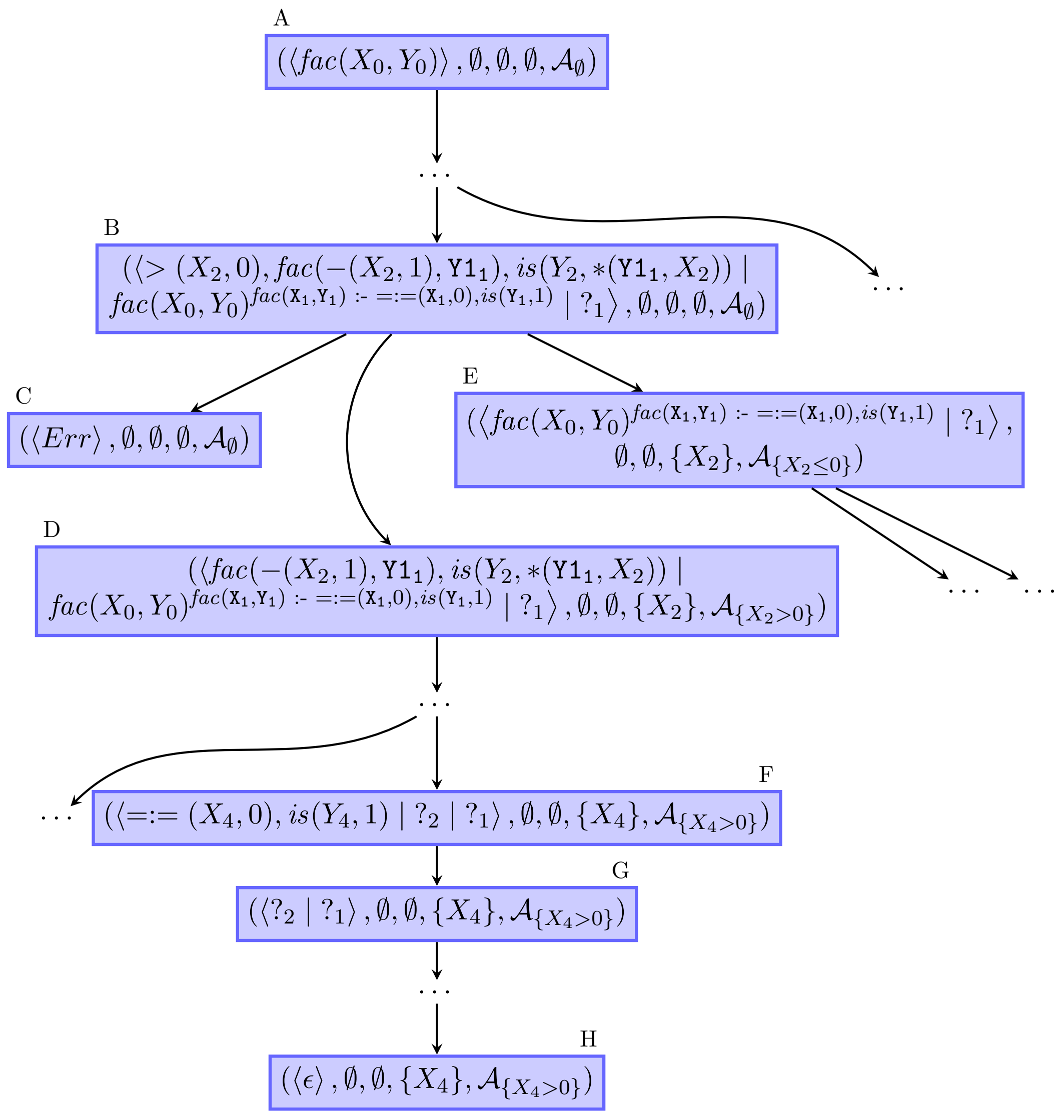

A
$(\langle fac(X_0, Y_0)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
B
$(\langle > (X_2, 0), fac(-(X_2, 1), \mathtt{Y1}_1), is(Y_2, *(\mathtt{Y1}_1, X_2)) \mid fac(X_0, Y_0)^{fac(\mathtt{X}_1,\mathtt{Y}_1) \text{ :- } =:=(\mathtt{X}_1,0), is(\mathtt{Y}_1,1)} \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
C
$(\langle Err\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
E
$(\langle fac(X_0, Y_0)^{fac(\mathtt{X}_1,\mathtt{Y}_1) \text{ :- } =:=(\mathtt{X}_1,0), is(\mathtt{Y}_1,1)} \mid ?_1\rangle, \emptyset, \emptyset, \{X_2\}, \mathcal{A}_{\{X_2 \leq 0\}})$
D
$(\langle fac(-(X_2, 1), \mathtt{Y1}_1), is(Y_2, *(\mathtt{Y1}_1, X_2)) \mid fac(X_0, Y_0)^{fac(\mathtt{X}_1,\mathtt{Y}_1) \text{ :- } =:=(\mathtt{X}_1,0), is(\mathtt{Y}_1,1)} \mid ?_1\rangle, \emptyset, \emptyset, \{X_2\}, \mathcal{A}_{\{X_2 > 0\}})$
F
$(\langle =:= (X_4, 0), is(Y_4, 1) \mid ?_2 \mid ?_1\rangle, \emptyset, \emptyset, \{X_4\}, \mathcal{A}_{\{X_4 > 0\}})$
G
$(\langle ?_2 \mid ?_1\rangle, \emptyset, \emptyset, \{X_4\}, \mathcal{A}_{\{X_4 > 0\}})$
H
$(\langle \epsilon\rangle, \emptyset, \emptyset, \{X_4\}, \mathcal{A}_{\{X_4 > 0\}})$

Figure 4.2.: Parts of abstract evaluation of $fac(X, Y)$ on Program 4.2

state represented by node B. The first state is the error state, which is represented by node C. This node has no more successors, just as a concrete error state would.

The second and third state are the states reached by success and failure of the comparison, respectively. They are represented by nodes D and E. Note that we updated the arithmetic variables in both states to contain the set $\mathcal{T}(X_2) \cup \mathcal{T}(0) = \{X_2\}$. We also updated the arithmetic states of both states to reflect whether the comparison $X_2 = 0$ succeeds or fails.

We consider the remaining evaluation of node D. After some intermediate steps, which are not shown in this figure, we reach node F. During the intermediate steps, the variable X_2 has been renamed to X_4. At this point we have to decide whether or not the evaluation of the arithmetic term represented by the term variable X_4 will succeed and whether or not the integer this term evaluates to is equal to zero.

Both of these pieces of information are stored in the knowledge base of node D. We see that both $\mathit{safe}_{\{X\},\mathcal{A}_{\{X>0\}}}(X) \equiv \mathit{true}$ and $\mathcal{A}_{\{X>0\}} \models (X \neq 0)$ hold true. Hence, we know that the evaluation of X will succeed and that the comparison of this term for equality with zero will fail. Thus, we have to apply Abstract Evaluation Rule 4.9 and receive only a single state that corresponds to the aforementioned facts. This state is represented by node G.

At this point, we only need to remove the end-of-scope markers $?_2$ and $?_1$. We can do so by applying Abstract Evaluation Rule 4.6 twice and receive the node labeled with H in Figure 4.2. This node does not have any successor states, so the inference terminates at this point. ▲

We have seen in the previous example that our choice of rules provides a reasonable abstraction of the concrete evaluation of the predicates for arithmetic comparison. In Section 4.5 we show that the abstraction provided by this set of rules is actually sound. The main idea for that proof is that whenever we produce multiple abstract successor states, our changes to the knowledge base only formalize the case analysis performed in that rule. We formalize and prove this idea in the following two lemmas.

Lemma 4.5. *Let*

$$s = (\langle \bowtie(t_1,t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$$

be an abstract state, where $\bowtie \in \{\texttt{=:=}, \texttt{=\textbackslash=}, \texttt{<}, \texttt{>}, \texttt{=<}, \texttt{>=}\}$, *and let* $\mathfrak{s}$ *be a member of* $\mathit{Conc}(s)$ *with the concretizing substitution* γ.

If Concrete Evaluation Rule 3.8 is applicable to $\mathfrak{s}$*, then* γ *also conforms to*

$$s' = \left(\langle \bowtie(t_1,t_2), T \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}'\right),$$

where $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$, $\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$ *and* $\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{t_1 \bowtie t_2\}}$. ■

Proof. It is clear that γ fulfills the first and the third condition of Definition 4.6, since both $S := \langle \bowtie(t_1,t_2), T \mid G\rangle$ and $\mathcal{U}$ are the same in both abstract states. Hence, the second, fourth and fifth condition remain to be shown.

We start by showing that for all $X \in \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$, it holds true that $\mathcal{V}(X\gamma)$ is empty. If X is a member of $\mathcal{G}$, this is clear, since γ conforms to s. If, however, X is a member of $\mathcal{T}(t_1)$ or $\mathcal{T}(t_2)$, Lemma 3.2 yields that $X\gamma$ does not contain any program variables. Hence, this statement holds true.

We now show that for all $X \in \mathcal{E} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$, it holds true that $\mathit{eval}_C(X\gamma) \neq \bot$. Again, if X is a member of $\mathcal{E}$, then we know that this holds true since γ conforms to s. Also, if X is a member of $\mathcal{T}(t_1) \cup \mathcal{T}(t_2)$, then Lemma 3.1 yields the truth of the statement. Thus, this statement holds true as well.

It remains to show that for all $\mathit{eval}_E \circ \gamma\big|_{\mathcal{T}}$ is a member of $\mathcal{A} \cap \mathcal{A}_{\{t_1 \bowtie t_2\}}$. We know that it is a member of $\mathcal{A}$ since γ already conformed to s. Furthermore, we know that $(t_1 \bowtie t_2)\mathit{eval}_E \circ \gamma\big|_{\mathcal{T}}$ is a tautology, since we were able to apply Concrete Evaluation

Rule 3.8 to $\mathfrak{s}$, which is only applicable if $eval_C(\bowtie(t_1, t_2))$ holds true. Hence, $eval_E \circ \gamma\big|_{\mathcal{T}}$ is a member of $\mathcal{A}_{\{t_1 \bowtie t_2\}}$ by Definition 4.2 as well. Thus, the statement holds true.

We have thus shown that γ fulfills all conditions of Definition 4.6. Thus, γ conforms to s' and Lemma 4.5 holds true. □

Lemma 4.6. *Let*

$$s = (\langle \bowtie(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$$

be an abstract state, where $\bowtie \in \{=:=, =\backslash=, <, >, =<, >=\}$*, and let* $\mathfrak{s}$ *be a member of* $Conc(s)$ *with the concretizing substitution* γ.

If Concrete Evaluation Rule 3.9 is applicable to $\mathfrak{s}$*, then* γ *also conforms to*

$$s' = \left(\langle \bowtie(t_1, t_2), T \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}'\right),$$

where $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$*,* $\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_1) \cup \mathcal{T}(t_2)$ *and* $\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{\neg(t_1 \bowtie t_2)\}}$. ■

Proof. This proof is nearly identical to the previous one, except for the replacement of $t_1 \bowtie t_2$ by $\neg(t_1 \bowtie t_2)$. We can copy the previous proof and receive that Lemma 4.6 holds true as well. □

4.4.3. Evaluation of Arithmetic Assignment

In the previous section, we have defined rules that handle abstract states of the form $(\langle(\bowtie(t_1, t_2), T) \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$, where $\bowtie$ is one of =:=, =\=, <, >, =<, and >=. We have also treated the cut in the section before that. The only built-in predicate that we still need to handle is the function symbol $is/2$. Its semantics have been formally defined in Section 3.2 *(States and Evaluation)*. The intuitive idea behind this predicate is to evaluate its right-hand side to some integer and then attempt unification between this integer and the left-hand side. If the evaluation of the right-hand side fails, an error is thrown, which, in our case, leads to the abortion of the inference.

Similar to the case of arithmetic comparisons, we have three possible outcomes of the evaluation of this predicate. Again, it may not be possible to decide precisely which of the three outcomes is going to happen, due to the presence of term variables.

Example 4.14 (*Is*-predicate in the presence of term variables)**.** Consider the abstract state

$$s := (\langle is(1, X), f \mid g\rangle, \emptyset, \emptyset, \emptyset, \emptyset)$$

and the three substitutions

$$\sigma_1 := X \mapsto //(1, 0) \qquad \sigma_2 := X \mapsto 1 \qquad \sigma_3 := X \mapsto 2$$

all of which conform to s. These substitutions concretize s to the concrete states

$$\begin{aligned}\mathfrak{s}_1 &:= \langle is(1, X), f \mid g\rangle\,\sigma_1 = \langle is(1, //(1, 0)), f \mid g\rangle\\ \mathfrak{s}_2 &:= \langle is(1, X), f \mid g\rangle\,\sigma_2 = \langle is(1, 1), f \mid g\rangle\\ \mathfrak{s}_3 &:= \langle is(1, X), f \mid g\rangle\,\sigma_3 = \langle is(1, 2), f \mid g\rangle\end{aligned}$$

We see that these three states, all of which are concretizations of s, have vastly differing concrete successor states. During the evaluation of $\mathfrak{s}_1$, the program tries to evaluate $//(1,0)$ which leads to the successor state

$$\mathfrak{s}'_1 = \langle \mathit{Err} \rangle$$

The evaluation of the right-hand side succeeds in the state s_2, as does the unification of the two terms 1 and 1. Thus, the evaluation of the current term succeeds and the resulting state is

$$\mathfrak{s}'_2 = \langle f \mid g \rangle$$

In the final state $\mathfrak{s}_3$, the evaluation of the right-hand expression succeeds as well, but the unification with the left-hand side does not, since the terms 1 and 2 do not unify. Hence, the goal is abandoned and we receive the successor state

$$\mathfrak{s}'_3 = \langle g \rangle$$

There is no single abstract state that represents all of these successor states. ▲

Again, this example displays the worst possible case with regards to the number of abstract states that we produce. In fact, there are several cases in which we can find a single abstract state that represents all possible concrete successor states. The simplest of these cases is the one in which the evaluation of the right-hand side of the term leads to an error. In this case we only need to produce an error state. This is formalized in Abstract Evaluation Rule 4.14.

Abstract Evaluation Rule 4.14 Arithmetic Assignment Rule (Error)

$$\text{ArithAssError}\frac{(\langle \mathit{is}(t_1, t_2), T \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle \mathit{Err} \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}$$

Where: $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$

Another simple case is the one in which we are certain that the unification of the result of the evaluation of the right-hand side with the left-hand side will fail. This is, for example, the case if the left-hand side is neither a variable nor an integer. Even if the right-hand side is evaluated successfully, the result of the evaluation can only unify with a variable or an integer. Thus, we only need to produce a state in which the unification fails in this case, as long as we are certain that the evaluation of the right-hand side succeeds. The formal rule for this is given as Abstract Evaluation Rule 4.15.

A similar rule for the case that we are not sure whether or not the evaluation of the right-hand side succeeds is given as Abstract Evaluation Rule 4.16. In this rule we update the set of arithmetic variables in the non-error state, just as we did in the similar case for arithmetic comparison in Abstract Evaluation Rule 4.12.

In the previous two rules we excluded three possibilities for t_1. For these rules to be applicable, t_1 may neither be a member of $\mathbb{Z}$, nor $\mathcal{V}$, nor $\mathcal{T}$. We now define rules to handle each of these three cases and start with the case that t_1 is a member of $\mathbb{Z}$.

Abstract Evaluation Rule 4.15 Arithmetic Assignment Rule (No Unification, Safe)

$$\text{ArithAssNoUnificationSafe}\frac{(\langle is(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$
$t_1 \notin \mathbb{Z} \cup \mathcal{V} \cup \mathcal{T}$

Abstract Evaluation Rule 4.16
Arithmetic Assignment Rule (No Unification, Unsafe)

$$\text{ArithAssNoUnificationUnsafe}\frac{(\langle is(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{Err \quad (\langle G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E}',\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_2)$
$safe_{\mathcal{E},\mathcal{A}}(t_2) = maybe$
$t_1 \notin \mathbb{Z} \cup \mathcal{V} \cup \mathcal{T}$

In this case there are three possible outcomes: Either the evaluation of the right-hand side fails, or it succeeds and we have to distinguish between succeeding and failing unification. If the right-hand term evaluates to the same integer as the one on the left-hand side, the unification succeeds and the evaluation continues with the same goal. Otherwise, the goal is abandoned and the evaluation continues with the next goal.

In the best case, we are certain that the evaluation of the right-hand side succeeds and we can infer whether or not it is the same integer as the left-hand side using the arithmetic state. If we can infer either of these facts, we only have to produce a single successor state. The case in which the integers on either side are the same is treated in Abstract Evaluation Rule 4.17. The case of a failing unification is handled in Abstract Evaluation Rule 4.18.

Abstract Evaluation Rule 4.17 Arithmetic Assignment Rule (Literal, Success, Safe)

$$\text{ArithAssLitSuccessSafe}\frac{(\langle is(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle T \mid G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \models (n = t_2)$

It may also be the case that we know that the evaluation of the right-hand side of the term succeeds, but we do not know whether or not it evaluates to the same integer as

Abstract Evaluation Rule 4.18 Arithmetic Assignment Rule (Literal, Failure, Safe)

$$\text{ArithAssLitFailSafe}\frac{(\langle \mathit{is}(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \models (n \neq t_2)$

the left-hand side. In this case we need to produce two states, one covering the case in which the unification succeeds, and one in which it fails. These states are created using Abstract Evaluation Rule 4.19. We update the resulting arithmetic states according to success or failure of the unification in a similar way to the corresponding case distinction in Abstract Evaluation Rule 4.10.

Abstract Evaluation Rule 4.19
Arithmetic Assignment Rule (Literal, Unknown, Safe)

$$\text{ArithAssLitUnknownSafe}\frac{(\langle \mathit{is}(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle T \mid G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A}') \quad (\langle G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E},\mathcal{A}'')}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{t_1=t_2\}}$
$\mathcal{A}'' := \mathcal{A} \cap \mathcal{A}_{\{t_1\neq t_2\}}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \not\models (n = t_2)$ **and** $\mathcal{A} \not\models (n \neq t_2)$

In all of the preceding rules we have assumed that we are certain that the evaluation of the right-hand side of the predicate succeeds. This was formalized using the condition that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$ holds true. We have covered the case that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$ holds true previously in Abstract Evaluation Rule 4.14. Hence, we need to consider the case that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$ holds true.

This assumption does not change the basic reasoning behind the given rules. The only major change we need to make is the production of an error state. In the non-error states we furthermore have to store the assumption that the evaluation of t_2 succeeded. We do this by adding the term variables of t_2 to the set of arithmetic variables, just like we did in the previous section when evaluating arithmetic comparisons.

The case in which we do not know whether or not the evaluation of the right-hand side succeeds, but we are certain that, if it succeeds, it evaluates to the same integer as the one on the left-hand side, is covered in Abstract Evaluation Rule 4.20. If we are certain that, if the right-hand side evaluates without an error, it evaluates to some integer other than the one on the left-hand side, we apply Abstract Evaluation Rule 4.21. Finally, the case in which we have the least information is formalized in Abstract Evaluation Rule 4.22. In this case we know neither whether the evaluation of the right-hand side succeeds, nor, if it does, to which integer it will evaluate.

Abstract Evaluation Rule 4.20

Arithmetic Assignment Rule (Literal, Success, Unsafe)

$$\text{ArithAssLitSuccessUnsafe}\frac{(\langle \mathit{is}(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle \mathit{Err}\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \quad (\langle T \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_2)$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \models (n = t_2)$

Abstract Evaluation Rule 4.21

Arithmetic Assignment Rule (Literal, Failure, Unsafe)

$$\text{ArithAssLitFailUnsafe}\frac{(\langle \mathit{is}(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle \mathit{Err}\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \quad (\langle G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A})}$$

Where: $\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_2)$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \models (n \neq t_2)$

At this point, two cases remain to be handled. It may still be the case that t_1 is either a program- or a term variable. These cases are denoted by $t_1 \in \mathcal{V}$ and $t_1 \in \mathcal{T}$, respectively. We treat the case $t_1 \in \mathcal{V}$ in the following rules.

If t_1 is a program variable, then we are certain that the unification succeeds, since a program variable unifies with every term. However, even if we are able to tell that the right-hand side of the *is*-predicate evaluates without an error, we may not be able to determine which integer that is. Due to these complications, we take a slight detour in order to evaluate abstract states of this form.

For this, we take a fresh term variable and replace all occurrences of the left-hand side of the *is*-predicate with this fresh variable. We furthermore store the equality between the new term variable and the result of the evaluation of the expression on the right-hand side in the arithmetic state. The fresh variable then represents the result of the evaluation of the right-hand side.

Example 4.15 (Abstract evaluation of $\mathit{is}(\mathtt{X}, Y)$)**.** Consider the abstract state

$$s := (\langle \mathit{is}(\mathtt{X}, Y), =:=(\mathtt{X}, Y)\rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_\emptyset)$$

Abstract Evaluation Rule 4.22

Arithmetic Assignment Rule (Literal, Unknown, Unsafe)

$$\text{ArithAssLitUnknownUnsafe}\frac{(\langle is(t_1,t_2),T \mid G\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A})}{(\langle Err\rangle, \mathcal{G},\mathcal{U},\mathcal{E},\mathcal{A}) \quad s_1' \quad s_2'}$$

Where:
$s_1' := (\langle T \mid G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E}',\mathcal{A}')$
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{t_1=t_2\}}$
$s_2' := (\langle G\rangle, \mathcal{G}',\mathcal{U},\mathcal{E}',\mathcal{A}'')$
$\mathcal{A}'' := \mathcal{A} \cap \mathcal{A}_{\{t_1\neq t_2\}}$
$safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv maybe$
$t_1 = n \in \mathbb{Z}$
$\mathcal{A} \not\models (n = t_2)$ **and** $\mathcal{A} \not\models (n \neq t_2)$

in which `X` is a program variable and Y is a term variable. The two concretizations

$$\sigma_1 := Y \mapsto 1 \qquad\qquad \sigma_2 := Y \mapsto 2$$

both conform to s. Hence, we cannot tell which number is bound to `X` in the remainder of the execution. Furthermore, we are unable to store the relation $\mathtt{X} = Y$ in the arithmetic state of a potential abstract successor state, since the arithmetic state can only store relations over term variables, but not over program variables.

Thus, we pick the fresh term variable Z, replace all occurrences of `X` in the remaining goal with Z and store the fact $Z = Y$ in the resulting abstract state. We also add Z to the set of ground variables as well as to the set of arithmetic variables. Hence, we end up with the resulting abstract state

$$s' = (\langle =:=(\mathtt{Z}, Y)\rangle, \{Y, Z\}, \emptyset, \{Y, Z\}, \mathcal{A}_{\{Z=Y\}})$$

which represents all concrete successor states of s. ▲

The intuition of this method is formalized in Abstract Evaluation Rule 4.23 for the case that we are certain that the right-hand side of the predicate evaluates to some integer. If we cannot tell whether or not this is the case, we apply Abstract Evaluation Rule 4.24 and produce an error state in addition to the succeeding successor state.

There remains the final case in which t_1 is a term variable, which is denoted by $t_1 \in \mathcal{T}$. Since t_1 denotes an arbitrary term in this case, which is only constrained by the knowledge base of its state, we have no way of telling whether t_1 represents a number, a program variable or some other term. Hence, we need to account for both possibilities of succeeding and failing unification. Thus, even if we are certain that t_2 is evaluated without errors, we have to produce two states.

In the case that the unification succeeds, we use the same method that we applied in Abstract Evaluation Rule 4.23 and Abstract Evaluation Rule 4.24. For the failing case,

Abstract Evaluation Rule 4.23

Arithmetic Assignment Rule (Program Variable, Safe)

$$\text{ArithAssProgvarSafe}\frac{(\langle \mathit{is}(t_1,t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle T[t_1 \mapsto Y] \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}, \mathcal{A}')}$$

Where: $Y \notin \mathcal{T}$ is a fresh term variable
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{Y=t_2\}}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$t_1 \in \mathcal{V}$

Abstract Evaluation Rule 4.24

Arithmetic Assignment Rule (Program Variable, Unsafe)

$$\text{ArithAssProgvarUnsafe}\frac{(\langle \mathit{is}(t_1,t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle \mathit{Err}\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \quad (\langle T[t_1 \mapsto Y] \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}')}$$

Where: $Y \notin \mathcal{T}$ is a fresh term variable
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \{Y\} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{Y=t_2\}}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$
$t_1 \in \mathcal{V}$

we simply store the relation $t_1 \neq t_2$ in the arithmetic state. This intuition is formalized in Abstract Evaluation Rule 4.26.

A similar rule is given as Abstract Evaluation Rule 4.26 for the case in which we are uncertain whether or not the evaluation of the right-hand side of the predicate succeeds.

The most common use case for the *is*-predicate is the evaluation of an expression and the assignment of the result to a program variable on the left-hand side. In this section we have provided abstract execution rules for all possible uses of this predicate. In combination with those rules defined in the previous section, these rules provide a sound and precise abstraction of the concrete behavior of PROLOG. They are also amenable to abstract interpretation, since the choice of rules is deterministic. We show these properties in the following section.

4.5. Properties of the Abstract Evaluation Relation

In the previous section, we have defined a set of 26 rules that define an abstraction of the concrete execution of a program defined in Section 3 *(Concrete Semantics)*. We have claimed that this abstraction is not only sound, but that it is also amenable for automated abstract analysis. In this section, we show both of these claims.

Abstract Evaluation Rule 4.25 Arithmetic Assignment Rule (Term Variable, Safe)

$$\text{ArithAssTermvarSafe}\frac{(\langle \mathit{is}(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle T[t_1 \mapsto Y] \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}, \mathcal{A}') \quad (\langle G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}, \mathcal{A}'')}$$

Where: $Y \notin \mathcal{T}$ is a fresh term variable
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{Y=t_2\}}$
$\mathcal{A}'' := \mathcal{A} \cap \mathcal{A}_{\{t_1 \neq t_2\}}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$t_1 \in \mathcal{T}$

Abstract Evaluation Rule 4.26
Arithmetic Assignment Rule (Term Variable, Unsafe)

$$\text{ArithAssTermvarUnsafe}\frac{(\langle \mathit{is}(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})}{(\langle \mathit{Err}\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \quad s_1' \quad s_2'}$$

Where: $s_1' := (\langle T[t_1 \mapsto Y] \mid G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}', \mathcal{A}')$
$Y \notin \mathcal{T}$ is a fresh term variable
$\mathcal{G}' := \mathcal{G} \cup \mathcal{T}(t_2)$
$\mathcal{E}' := \mathcal{E} \cup \{Y\} \cup \mathcal{T}(t_2)$
$\mathcal{A}' := \mathcal{A} \cap \mathcal{A}_{\{Y=t_2\}}$
$s_2' := (\langle G\rangle, \mathcal{G}', \mathcal{U}, \mathcal{E}'', \mathcal{A}'')$
$\mathcal{A}'' := \mathcal{A} \cap \mathcal{A}_{\{t_1 \neq t_2\}}$
$\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$
$t_1 \in \mathcal{T}$

We begin by showing that the choice of a rule to apply to any given abstract state is deterministic in Section 4.5.1. In Section 4.5.2, we show the soundness of this abstraction.

4.5.1. Determinacy

We are now going to show that each abstract state has at most a single rule that can be applied to it. First we formalize this notion.

Definition 4.10 (Deterministic abstract evaluation)**.** Let $R = r_1, \ldots, r_n$ be a set of abstract evaluation rules. We call R **deterministic** if the following holds:

for all $s \in \mathit{AbstractStates}$ **it holds that**
for all $i \in [1; n]$. $[r_i$ **is applicable to** $s]$ **implies**
$[$**for all** $j \in [1; n]$. $i \neq j$ **implies** $[r_j$ **is not applicable to** $s]]$

■

Lemma 4.7 (Determinacy of abstract rules)**.** *The set R consisting of the Abstract Evaluation Rules 4.1 through 4.26 is deterministic.* ■

Proof. Let R be the set of Abstract Evaluation Rules 4.1 through 4.26 and let $s = (\langle t, T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. Let i be some integer in the range $[1; 26]$ and assume that Abstract Evaluation Rule 4.i is applicable to s. We show that there is no integer j in the range $[1; 26]$ such that $i \neq j$ and such that Abstract Evaluation Rule 4.j is applicable to s by case distinction with respect to the value of i.

Case $i = 1$ In this case we have $t = \Box$ and $T = \epsilon$. Since no other rule allows this, no other rule is applicable to s.

Case $i = 2$ In this case we know that t is of the form $f(t_1, \ldots, t_n)$ for some $n \geq 0$, where f is a user-defined function symbol. No other rule handles unlabeled user-defined predicates, so no other rule is applicable to s.

Case $i = 3$ In this case we have $(t, T)^{h \,:\!-\, b}$, where there is no conforming substitution γ such that $t\gamma$ and h unify. The only other rule that is applicable to labeled goals, namely rule 4.4 is only applicable if there exists such a substitution. Hence, no other rule is applicable to s.

Case $i = 4$ The same reasoning as in the previous case applies. Thus, no other rule is applicable to s.

Case $i \in \{5, 6\}$ In this case we have either $t = !_{\mathrm{m}}$ or $t = ?_{\mathrm{m}}$. Since no other rule handles these constructs, no other rule is applicable to s in either case.

Case $i = 7$ We have $t = \bowtie(t_1, t_2)$, where $\bowtie \in \{=:=, =\backslash=, <, >, =<, >=\}$ and $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$. Since t is of the form $\bowtie(t_1, t_2)$, the only candidates for other applicable rules are rules 4.8 through 4.13. However, all of these rules demand that either $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$ or $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_1) \otimes \mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$ holds true, both of which are obviously false in this case. Thus, no other rule is applicable to s.

Case $i \in \{8, 9, 10, 11, 12, 13\}$ If either of these rules is applicable, the respective other rules are the only other candidates for applicability due to the syntactic structure of $t = \bowtie(t_1, t_2)$ are the other rules treated in this case. We list the restrictions for applicability of this rule in Table 4.1. It is then easy to see that, if the requirements for one Rule are fulfilled, those for none of the other Rules are fulfilled. Thus, no other rules are applicable to s.

Case $i = 14$ If Abstract Evaluation Rule 4.14 is applicable to s, then we have $t = \mathit{is}(t_1, t_2)$, where $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$ holds true. Since all other rules that are applicable to $\mathit{is}(t_1, t_2)$ require that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{false}$ does not hold true, no other rule is applicable to s.

	$safe_{\mathcal{E},\mathcal{A}}(t_1) \otimes safe_{\mathcal{E},\mathcal{A}}(t_2)$	$\mathcal{A} \models \bowtie(t_1, t_2)$	$\mathcal{A} \models \neg\bowtie(t_1, t_2)$
Rule 4.8	*true*	*true*	*false*
Rule 4.9	*true*	*false*	*true*
Rule 4.10	*true*	*false*	*false*
Rule 4.11	*maybe*	*true*	*false*
Rule 4.12	*maybe*	*false*	*true*
Rule 4.13	*maybe*	*false*	*false*

Table 4.1.: Requirements for applicability of Abstract Evaluation Rules 4.8 through 4.13

Case $i = 15$ If Abstract Evaluation Rule 4.15 is applicable to s, then $t = is(t_1, t_2)$ and $t_1 \notin \mathbb{Z} \cup \mathcal{V} \cup \mathcal{T}$. The only other rule that allows this is Abstract Evaluation Rule 4.16, which demands that $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv maybe$ holds true. Since Abstract Evaluation Rule 4.15 requires $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$ to hold true, Abstract Evaluation Rule 4.16 is not applicable. Thus, no other rule is applicable to s.

Case $i = 16$ The reasoning in this case is the same as in the previous case. For the same reason that Abstract Evaluation Rule 4.16 was not applicable in that case, Abstract Evaluation Rule 4.15 is not applicable in this case. Thus, no other rule is applicable to s.

Case $i \in \{17, 18, 19, 20, 21, 22\}$ In all of these cases we have $t = is(t_1, t_2)$ and $t_1 \in \mathbb{Z}$. Thus, the only candidates for applicable rules in these cases are Abstract Evaluation Rules 4.17 through 4.22. We summarize the requirements for applicability of these rules in Table 4.2. It is easy to see that if one of these rules is applicable to s, none of the others is. Thus, no other rule is applicable to s.

	$safe_{\mathcal{E},\mathcal{A}}(t_2)$	$\mathcal{A} \models (n = t_2)$	$\mathcal{A} \models (n \neq t_2)$
Rule 4.17	*true*	*true*	*false*
Rule 4.18	*true*	*false*	*true*
Rule 4.19	*true*	*false*	*false*
Rule 4.20	*maybe*	*true*	*false*
Rule 4.21	*maybe*	*false*	*true*
Rule 4.22	*maybe*	*false*	*false*

Table 4.2.: Requirements for applicability of Abstract Evaluation Rules 4.17 through 4.22

Case $i \in \{23, 24\}$ In these cases the only candidate for applicability is the other rule, respectively. However, since Abstract Evaluation Rule 4.23 requires $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{true}$ to hold true, whereas Abstract Evaluation Rule 4.24 requires $\mathit{safe}_{\mathcal{E},\mathcal{A}}(t_2) \equiv \mathit{maybe}$, the respective other rule is not applicable. Thus, no other rule is applicable to s.

Case $i \in \{25, 26\}$ The same reasoning as in the previous case applies here. Thus, no other rule is applicable to s.

Conclusion We have shown for all $i \in [1; 26]$ that, if Abstract Evaluation Rule 4.i is applicable to some abstract state s, none of the other abstract evaluation rules are. Thus, we have shown the claim that the set of abstract evaluation rules given in the previous sections is deterministic. □

This lemma is one of the main justifications for our informal claim that our abstract semantics is well-suited for automated analysis of programs. We state the other, more important justification for this claim in the next section, where we show that our abstraction is actually sound.

4.5.2. Soundness

The typical properties of interest for abstract semantics are those of soundness and completeness. An abstract semantics is called sound if it does not strictly underapproximate the concrete semantics. It is called complete if it does not strictly overapproximate the concrete semantics. If a semantics is both sound and complete, it provides an exact representation of the underlying concrete semantics.

The semantics that we have defined in the previous section is sound, but incomplete. We will show both of these statements in this section.

We start with the definition of the abstract evaluation relation $\Rightarrow$. This relation is defined similarly to $\rightarrow$ in the concrete case.

Definition 4.11 (Abstract evaluation relation)**.** Let s and s' be abstract states. We say that s **evaluates to** s' iff there exists an abstract evaluation rule that transforms s into s'. We write $s \Rightarrow s'$ in this case.

We furthermore write $s \Rightarrow^* s'$ if there exists a sequence $s_1, \ldots, s_k$ of states such that $s = s_1$, $s' = s_k$ and $s_i \Rightarrow s_{i+1}$ holds true for all $i \in [1; k-1]$. ■

This definition allows us to write $s \Rightarrow^* s'$ to denote that s evaluates to s' via some intermediate states. Now we formally define the notion of completeness and give an example that shows that the set of abstract evaluation rules given in the previous section is incomplete.

Definition 4.12 (Complete abstract execution relation)**.** Let

$$r = \frac{s}{s_1 \quad \ldots \quad s_n}$$

be an inference rule. We say that r is **complete** if

$$\bigcup_{i=1}^{n} \mathit{Conc}(s_i) \subseteq \mathit{Next}(\mathit{Conc}(s))$$

holds true, where $\mathit{Next}(S) := \{s' \in \mathit{ConcreteStates} \mid \exists s \in S.\ s \to s'\}$. ■

Example 4.16 (Incompleteness of Abstract Evaluation Rule 4.23)**.** We pick the abstract state $s = (\langle is(\mathtt{X}, 1), >(\mathtt{X}, 0) \mid g\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$, where X is a program variable. We see that we can apply Abstract Evaluation Rule 4.23 to this state and receive the single resulting abstract state

$$s' = (\langle >(Y, 0) \mid g\rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_{\{Y=1\}})$$

The substitution $\sigma : Y \to +(1, 0)$ conforms to s', whence the concrete state $\mathfrak{s}' := \langle >(+(1, 0), 0) \mid g\rangle$ is represented by s'. However, there exists no concrete state $\mathfrak{s}$ that is represented by s, which is evaluated to $\mathfrak{s}'$. ▲

We have seen that our abstract semantics is incomplete. However, as long as the abstract semantics is still sound, it is useful for the automated analysis of programs. We now define the notion of soundness and show that the abstract evaluation relation that we defined in the previous sections is sound.

Definition 4.13 (Sound abstract evaluation rule)**.** Let

$$r = \frac{s}{s_1 \quad \dots \quad s_n}$$

be an inference rule. We say that r is **sound** if

$$\bigcup_{i=1}^{n} \mathit{Conc}(s_i) \supseteq \mathit{Next}(\mathit{Conc}(s))$$

holds true, where $\mathit{Next}(\mathfrak{S}) := \{\mathfrak{s}' \in \mathit{ConcreteStates} \mid \exists \mathfrak{s} \in \mathfrak{S}.\ \mathfrak{s} \to \mathfrak{s}'\}$. ■

We now proceed to show that all our previously defined rules for abstract evaluation are sound.

Lemma 4.8 (Abstract execution is sound)**.** *All Abstract Evaluation Rules 4.1 through 4.26 are sound.* ■

Proof. We first show for each rule

$$r = \frac{s}{s'_1 \quad \dots \quad s'_n}$$

that, if r is applicable to some abstract state s, then for each $\mathfrak{s} \in \mathit{Conc}(s)$ there exists a state s'_i such that $\mathfrak{s}' \in \mathit{Conc}(s'_i)$, where $\mathfrak{s}'$ is the concrete state that $\mathfrak{s}$ evaluates to.

Abstract Evaluation Rules 4.1 through 4.6 The proof of soundness of these rules can be found in [GSSK$^+$12, Appendix A].

Abstract Evaluation Rule 4.7 Let $s = (\langle \bowtie(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.7 is applicable to it. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \rightarrow \mathfrak{s}'$. If we apply Abstract Evaluation Rule 4.7 to s we receive $s' = (\langle Err\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Since Abstract Evaluation Rule 4.7 is applicable to s, $safe_{\mathcal{E},\mathcal{A}(t_1)} \otimes safe_{\mathcal{E},\mathcal{A}(t_2)} \equiv false$ must hold true and hence either $safe_{\mathcal{E},\mathcal{A}(t_1)} \equiv false$ or $safe_{\mathcal{E},\mathcal{A}(t_2)} \equiv false$ must hold true, according to the definition of $\otimes$. Due to Lemma 4.3, we have either $eval_E(t_1\gamma) = \bot$ or $eval_E(t_2\gamma) = \bot$. In either case, we have $\mathfrak{s}' = \langle Err\rangle$. Since $\langle Err\rangle \in Conc(s')$, Abstract Evaluation Rule 4.7 is sound.

Abstract Evaluation Rule 4.8 Let $s = (\langle \bowtie(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.8 is applicable to it. If we apply Abstract Evaluation Rule 4.8 to s, we receive $s' = (\langle T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \rightarrow \mathfrak{s}'$. Since γ conforms to s and since $\mathcal{A} \models (t_1 \bowtie t_2)$, we know that $(t_1 \bowtie t_2)\gamma$ is a tautology. Furthermore, due to $safe_{\mathcal{E},\mathcal{A}}(t_1) \equiv safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$ and Lemma 4.3 we have $eval_C(\bowtie (t_1, t_2)\gamma \equiv true$, whence $\mathfrak{s}' = \langle T \mid G\rangle$. Since $\mathfrak{s}' \in Conc(s')$, Abstract Evaluation Rule 4.8 is sound.

Abstract Evaluation Rule 4.9 Let $s = (\langle \bowtie(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.9 is applicable to it. If we apply Abstract Evaluation Rule 4.9 to s, we receive $s' = (\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \rightarrow \mathfrak{s}'$. Since γ conforms to s and since $\mathcal{A} \models \neg(t_1 \bowtie t_2)$, we know that $\neg(t_1 \bowtie t_2)\gamma$ is a tautology. Furthermore, due to $safe_{\mathcal{E},\mathcal{A}}(t_1) \equiv safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$ and Lemma 4.3 we have $eval_C(\bowtie(t_1, t_2)\gamma \equiv false$, whence $\mathfrak{s}' = \langle G\rangle$. Since $\mathfrak{s}' \in Conc(s')$, Abstract Evaluation Rule 4.9 is sound.

Abstract Evaluation Rules 4.10 through 4.13 There are no new concepts in the proofs of these rules. They amount to nothing more but a case distinction on γ, depending on whether $eval_C(\bowtie(t_1, t_2)) \equiv false$, $eval_C(\bowtie(t_1, t_2)) \equiv maybe$ or $eval_C(\bowtie(t_1, t_2)) \equiv true$. In each of these cases, we copy the proofs of soundness of Abstract Evaluation Rule 4.7, Abstract Evaluation Rule 4.8 and Abstract Evaluation Rule 4.9, respectively and apply Lemma 4.5 and Lemma 4.6.

Abstract Evaluation Rule 4.14 Let $s = (\langle is(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.14 is applicable to it. If we apply Abstract Evaluation Rule 4.14 to s, we receive $s' = (\langle Err\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \rightarrow \mathfrak{s}'$. Since Abstract Evaluation Rule 4.14 is applicable to s, the statement $safe_{\mathcal{E},\mathcal{A}(t_2)} \equiv false$ must hold true. Due to Lemma 4.3, we have $eval_E(t_2\gamma) = \bot$. Thus, we know that $\mathfrak{s}' = \langle Err\rangle$ holds true. Since $\langle Err\rangle$ is a member of $Conc(s')$, Abstract Evaluation Rule 4.14 is sound.

Abstract Evaluation Rule 4.15 Let $s = (\langle is(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.15 is applicable to it. If we apply Abstract Evaluation Rule 4.15 to s, we receive $s' = (\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Since Abstract Evaluation Rule 4.15 is applicable to s, it must hold that $t_1 \notin \mathbb{Z} \cup \mathcal{V}$. Also, we have $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$ and hence $eval_E(t_2\gamma) = n \neq \bot$. Thus, we know that $mgu(t_1\gamma, t_2\gamma) = \bot$, which implies $\mathfrak{s}' = \langle G\rangle$. Since $\mathfrak{s}' = \langle G\rangle \in Conc(s')$, Abstract Evaluation Rule 4.15 is sound.

Abstract Evaluation Rule 4.16 Let $s = (\langle is(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.16 is applicable to it. If we apply Abstract Evaluation Rule 4.16 to s, we receive $s'_1 = (\langle Err\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ and $s'_2 = (\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the two abstract successor states. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Since Abstract Evaluation Rule 4.15 is applicable to s, it must hold that $t_1 \notin \mathbb{Z} \cup \mathcal{V}$. Now assume that $eval_E(t_2\gamma) = \bot$. Then we know that $\mathfrak{s}' = \langle Err\rangle$. Since $\langle Err\rangle \in Conc(s'_1)$, Abstract Evaluation Rule 4.16 is sound in this case. Assume now that $eval_E(t_2\gamma) = n \neq \bot$. We then know that $mgu(t_1\gamma, t_2\gamma) = \bot$, and hence $\mathfrak{s}' = \langle G\rangle$. Since $\mathfrak{s}' = \langle G\rangle \in Conc(s')$, Abstract Evaluation Rule 4.16 is sound in this case. Hence, Abstract Evaluation Rule 4.16 is sound.

Abstract Evaluation Rule 4.17 Let $s = (\langle is(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.17 is applicable to it. If we apply Abstract Evaluation Rule 4.17 to s, we receive $s' = (\langle T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Since Abstract Evaluation Rule 4.17 is applicable to s, it must hold that $t_1 = n \in \mathbb{Z}$. Also, since $\mathcal{A} \models (t_1 = t_2)$, we know that $t_1\gamma = t_2\gamma$ holds true. Furthermore, since $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$, we know that $eval_E(t_2\gamma) \neq \bot$. We then know that $mgu(t_1\gamma, t_2\gamma) \neq \bot$, and hence $\mathfrak{s}' = \langle T \mid G\rangle$. Since $\mathfrak{s}' = \langle T \mid G\rangle \in Conc(s')$, Abstract Evaluation Rule 4.17 is sound.

Abstract Evaluation Rule 4.18 Let $s = (\langle is(t_1, t_2), T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.18 is applicable to it. If we apply Abstract Evaluation Rule 4.18 to s, we receive $s' = (\langle G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Since Abstract Evaluation Rule 4.18 is applicable to s, $t_1 = n \in \mathbb{Z}$ must hold true. Also, since $\mathcal{A} \models (t_1 \neq t_2)$, we know that $t_1\gamma \neq t_2\gamma$ holds true. Furthermore, since $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$, we know that $eval_E(t_2\gamma) \neq \bot$ holds true. We then know that $mgu(t_1\gamma, t_2\gamma) = \bot$, and hence $\mathfrak{s}' = \langle G\rangle$. Since $\mathfrak{s}' = \langle G\rangle \in Conc(s')$, Abstract Evaluation Rule 4.18 is sound.

Abstract Evaluation Rules 4.19 through 4.22 Again, these proofs do not contain any new ideas. We apply a simple case distinction to decide whether or not $eval_E(t_2\gamma) = \bot$ holds true. If this is the case, we apply the same reasoning as in the proof of soundness of Abstract Evaluation Rule 4.14. If this is not the case, we apply the same reasoning as in the proofs of soundness of Abstract Evaluation Rule 4.17 and Abstract Evaluation Rule 4.18. We see in all three cases that the respective rule is sound.

Abstract Evaluation Rule 4.23 Let $s = (\langle is(t_1, t_2), T \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.23 is applicable to it. If we apply Abstract Evaluation Rule 4.23 to s, we receive $s' = (\langle T[t_1 \mapsto Y] \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E} \cup \{Y\}, \mathcal{A} \cup \{Y = t_2\})$ as the single abstract successor state. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Since Abstract Evaluation Rule 4.23 is applicable to s, it must hold true that $t_1 = n \in \mathcal{V}$. It must furthermore hold true that $eval_E(t_2\gamma) = n \neq \bot$. Thus, we have $mgu(t_1, t_2) = (t_1 \mapsto n) \neq \bot$. Hence, the concrete successor state is of the form $\mathfrak{s}' = \langle T[t_1 \mapsto Y] \mid G \rangle$. Since the substitution $\gamma' : Y \mapsto n$ is conforming to s', we have $\mathfrak{s}' \in Conc(s')$. Thus, Abstract Evaluation Rule 4.23 is sound.

Abstract Evaluation Rule 4.24 Similar to earlier proofs, this one contains no new ideas. We perform a case distinction based on whether or not $eval_E(t_2\gamma) = \bot$. If this is the case, we copy the proof of soundness from Abstract Evaluation Rule 4.14. If this is not the case, we copy the proof of soundness from Abstract Evaluation Rule 4.23. In both cases we see that Abstract Evaluation Rule 4.24 is sound.

Abstract Evaluation Rule 4.25 Let $s = (\langle is(t_1, t_2), T \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state such that Abstract Evaluation Rule 4.25 is applicable to it. If we apply Abstract Evaluation Rule 4.25 to s, we receive $s'_1 = (\langle T[t_1 \mapsto Y] \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E} \cup \{Y\}, \mathcal{A} \cup \{Y = t_2\})$ and $s'_2 = (\langle G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A} \cup \{t_1 \neq t_2\})$ as the two abstract successor states. Let γ be some substitution conforming to s, let $\mathfrak{s} = s\gamma$ and let $\mathfrak{s}'$ be such that $\mathfrak{s} \to \mathfrak{s}'$. Assume that $mgu(t_1\gamma, t_2\gamma) \neq \bot$. In this case we copy the proof of soundness from Abstract Evaluation Rule 4.23. This is possible since we know that $safe_{\mathcal{E},\mathcal{A}}(t_2) \equiv true$ and hence $eval_E(t_2\gamma) \neq \bot$. If it is the case that $mgu(t_1\gamma, t_2\gamma) = \bot$ holds true, then we have $\mathfrak{s}' = \langle G \rangle$. Since $t_1\gamma$ and $t_2\gamma$ do not unify, we see that γ also conforms to s'_2. Thus, $\mathfrak{s}' \in Conc(s')$, whence Abstract Evaluation Rule 4.25 is sound.

Abstract Evaluation Rule 4.26 Similar to earlier proofs, this one contains no new ideas. We perform a case distinction based on whether or not $eval_E(t_2\gamma) = \bot$. If this is the case, we copy the proof of soundness from Abstract Evaluation Rule 4.14. If this is not the case, we copy the proof of soundness from Abstract Evaluation Rule 4.25. In both cases we see that Abstract Evaluation Rule 4.26 is sound.

Conclusion We have shown that all Abstract Evaluation Rules 4.1 through 4.26 are sound. Thus, Lemma 4.8 holds true. □

We now lift this result to the abstract evaluation relation defined by the previously defined abstract evaluation rules.

Definition 4.14 (Sound abstract evaluation relation)**.** Let $\Rightarrow$ be the Abstract Evaluation Relation as defined in Definition 4.11. We say that $\Rightarrow$ is **sound** if the following holds true:

$$\textbf{for all } s \in \mathit{AbstractStates}, \mathfrak{s} \in \mathit{Conc}(s), \mathfrak{s}' \in \mathit{ConcreteStates} \textbf{ it holds true that} \\ \mathfrak{s} \to \mathfrak{s}' \textbf{ implies that there exists } s' \in \mathit{AbstractStates} \\ \textbf{such that } s \Rightarrow s' \textbf{ and } \mathfrak{s}' \in \mathit{Conc}(s')$$

■

Lemma 4.9. *The abstract evaluation relation $\Rightarrow$ is sound.* ■

Proof. Let s be an abstract state and let $\mathfrak{s}$ and $\mathfrak{s}'$ be concrete states such that $\mathfrak{s} \in \mathit{Conc}(s)$ and $\mathfrak{s} \to \mathfrak{s}'$. Furthermore, let

$$r = \frac{s}{s'_1 \quad \dots s'_n}$$

be an abstract evaluation rule that is applicable to s. Then, according to Lemma 4.8, there exists an $s' \in \{s'_1, \dots, s'_n\}$ such that $\mathfrak{s}' \in \mathit{Conc}(s)$. Thus, $s \Rightarrow s'$. Hence, Lemma 4.9 holds true. □

We have shown that our abstract semantics is sound. In the next chapter, we construct a termination analysis for programs written in our fragment of Prolog.

5. Termination Analysis

In the previous section, we have developed a sound abstract semantics for a fragment of Prolog that contains not only the cut, but also built-in predicates for arithmetic comparison and evaluation. In this chapter we are going to develop a termination analysis for such programs. For this we are going to use the abstract semantics presented in the previous chapter. In short, we are going to develop an algorithm that tries decides the following decision problem:

> Given some program in the considered fragment of Prolog and some function symbol f/n, do all queries of the form $f(t_1, \ldots, t_n)$, where t_1 through t_n denote arbitrary terms, eventually terminate?

Since this problem is an instance of the halting problem, which was shown to be undecidable in [Tur36], our algorithm must necessarily stay incomplete. We will, however, construct a sound algorithm that decides this question in as many cases as possible if the given program terminates. If the given program does not terminate, however, the algorithm will not show nontermination, but merely return that the termination behavior of the program is unknown.

For this we are going to proceed in two steps. First, we are going to construct a finite representation of all concrete evaluations starting with some query of the form $f(t_1, \ldots, t_n)$. We call such a representation a termination graph. The construction of such graphs is defined in Section 5.1.

We are then going to transform these graphs into another formalism, so-called integer transition systems. The termination of such a transition system then implies the termination of all runs described by the termination graph. This construction as well as a proof of its correctness is given in Section 5.2.

Finally, in Section 5.3, we briefly discuss the choices we made for the practical implementation of this analysis. We also point to [Str10] for a more detailed presentation of the implementation.

The graph construction uses the construction from [GSSK$^+$12] as its base, but extends the construction in that publication in order to take arithmetic comparisons and evaluations into account. The reduction of a termination graph into an integer transition system was inspired by a similar reduction to this formalism in [SGB$^+$14]. The goal of that work, however, was to develop a termination analysis for LLVM, which has a very different semantics and thus a very different construction of termination graphs. Hence, the extraction of information from those graphs differs widely from the extraction in this work. This new construction of integer transition systems from termination graphs is a major contribution of this thesis.

5.1. Termination Graphs

In Section 4 *(Abstract Semantics)*, we introduced a set of rules that allow us to construct a tree of abstract program states from a given program and a starting query. However, as we have seen in Example 4.9, this abstract semantics may produce infinite runs in the same manner that the concrete evaluation of a term would. Thus, it is not useful to simply evaluate the starting query abstractly in order to analyze it for termination. In this section we instead construct a so-called termination graph based on this semantics. The nodes of this graph are abstract states. Thus, we use the terms state and node interchangeably in the remainder of this section.

We start this section by defining sets of potential outgoing edges for each state. Some of these edges are based on the abstract semantics of Prolog and correspond directly to steps of the inference algorithm. These edges are defined in Section 5.1.1. Other edges do not correspond to transitions of the semantics, but merely serve the construction of a finite graph. These edges are defined in Section 5.1.2 and Section 5.1.3. Finally, the complete construction of the termination graph is detailed in pseudocode in Section 5.1.4.

A similar graph construction appeared previously in [GSSK+12], parts of which we use for our construction. We extend this graph construction, however, in order to handle the additional semantic rules that we introduced in the previous chapter.

5.1.1. Abstract Semantics

The basic idea of a termination graph is to represent the tree of abstract states induced by the abstract evaluation relation $\Rrightarrow$. In order to simplify the transformation into integer transition systems later on, we annotate the edges of this tree with information about the abstract evaluation rule that is applicable to this state. In this section we discuss the edges of the graph that correspond directly to the abstract semantics.

The simplest abstract evaluation rules are those that can be applied based purely on the syntax of the current state and do not use any additional information stored in the knowledge base. For these rules, we do not need to add any information to the graph, but we can connect the original state and the resulting state with a simple edge.

Definition 5.1 (Simple edges)**.** Let s be an abstract state to which Abstract Evaluation Rule 4.1, Abstract Evaluation Rule 4.2, Abstract Evaluation Rule 4.4, Abstract Evaluation Rule 4.5, or Abstract Evaluation Rule 4.6 is applicable and let s' be the single state for which $s \Rrightarrow s'$ holds. We define the **simple edges of** s as

$$\mathit{SimpleEdges}(s) := \{(s, s')\}$$

■

Example 5.1 (Simple edges)**.** We pick the abstract state

$$s := (\langle \square \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

The only abstract rule that is applicable to s is Abstract Evaluation Rule 4.1, the application of which yields $s' = (\langle g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$.

The simple edge of s is formally denoted as

$$\mathit{SimpleEdges}(s) = ((\langle \square \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset), (\langle g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset))$$

We provide a graphical notation of this edge in Figure 5.1.

$$\begin{tikzcd}
\boxed{(\langle \square \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)} \arrow[d] \\
\boxed{(\langle g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)}
\end{tikzcd}$$

Figure 5.1.: The simple edge of $(\langle \square \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$

▲

If we can apply Abstract Evaluation Rule 4.3 to unify two terms, we denote the unifying substitution as an annotation on the edge. We call edges that are annotated with such a substitution unification edges.

Definition 5.2 (Unification edges)**.** Let s be an abstract state to which Abstract Evaluation Rule 4.3 is applicable and let s'_{succ} and s'_{back} be the respective resulting states such that $s \Rightarrow s'_{succ}$ and $s \Rightarrow s'_{back}$ hold. Furthermore, let σ be the unifying substitution of h and t that is applied to s to receive s'_{succ}. We define the **unification edges of** s as

$$\mathit{UnificationEdges}(s) := \{(s, s'_{succ}, \sigma), (s, s'_{back}, \bot)\}$$

■

Example 5.2 (Unification edges)**.** We pick the abstract state

$$s := (\left\langle (X, f(X) \mid g)^{f(\mathtt{Y})\ \texttt{:-}\ \square} \right\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

The only abstract rule that is applicable to s is Abstract Evaluation Rule 4.3, which yields $s_{succ} = (\langle f(f(\mathtt{Y})) \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$ and $s_{back} = (\langle g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$.

The unification edges of s are formally denoted as

$$\mathit{UnificationEdges}(s) = \{(s, s_{succ}, X \mapsto \mathtt{Y}_1, \mathtt{Y} \mapsto \mathtt{Y}_1), (s, s_{back}, \bot)\}$$

These edges are shown graphically in Figure 5.2.

▲

There are two large sets of semantic rules for which we still need to define rules. These are the rules for evaluation of arithmetic comparisons as well as those for the evaluation of the *is*-predicate. We start with the edges for arithmetic comparison. For these, we denote the arithmetic comparison that is assumed to succeed on the edge.

$$\begin{tikzcd}[column sep=small]
 & \boxed{\left(\left\langle (X, f(X) \mid g)^{f(\mathtt{Y})\,\mathtt{:-}\,\square} \right\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset\right)} \arrow[dl, "X \mapsto \mathtt{Y}_1, \mathtt{Y} \mapsto \mathtt{Y}_1"'] \arrow[dr, "\perp"] & \\
\boxed{\left\langle f(f(\mathtt{Y})) \mid g \right\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)} & & \boxed{(\langle g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)}
\end{tikzcd}$$

Figure 5.2.: The unification edges of $\left(\left\langle (X, f(X) \mid g)^{f(\mathtt{Y})\,\mathtt{:-}\,\square} \right\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset\right)$

Definition 5.3 (Comparison edges)**.** Let s be an abstract state to which one of the abstract evaluation rules from Abstract Evaluation Rule 4.7 through Abstract Evaluation Rule 4.13 is applicable, let s'_{err}, s'_{succ}, and s'_{fail} be the states resulting from the application of the abstract evaluation rule, such that $s \Rightarrow s'_{err}$, $s \Rightarrow s'_{succ}$, and $s \Rightarrow s'_{fail}$ hold. We write $s' = \perp$ if these states are not produced by the applicable rule. Furthermore, let $t_1 \bowtie t_2$ be the comparison that is evaluated in s.

We then define the **comparison edges of** s as

$$\begin{aligned}
\mathit{ComparisonEdges}(s) := {} & \{(s, s'_{err}, \perp) \mid s'_{err} \neq \perp\} \\
& \cup \{(s, s'_{succ}, (t_1 \bowtie t_2)) \mid s'_{succ} \neq \perp\} \\
& \cup \{(s, s'_{fail}, \neg(t_1 \bowtie t_2)) \mid s'_{fail} \neq \perp\}
\end{aligned}$$

■

Example 5.3 (Comparison edges)**.** We pick the abstract state

$$s := (\langle X > 3, f \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_{\{X=2\}})$$

The only abstract evaluation rule that is applicable to s is Abstract Evaluation Rule 4.12, which yields the two abstract states $s_{err} = (\langle \mathit{Err} \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_{\{X=2\}})$ and $s_{fail} = (\langle g \rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X=2\}})$ as the successor states of s.

The comparison edges of s are formally denoted as

$$\mathit{ComparisonEdges}(s) = \{(s, s_{err}, \perp), (s, s_{fail}, X \leq 3)\}$$

Note that s only has two comparison edges, since Abstract Evaluation Rule 4.12 only produces two successors of s. These edges are shown graphically in Figure 5.3.

$$\begin{tikzcd}[column sep=small]
 & \boxed{(\langle X > 3, f \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_{\{X=2\}})} \arrow[dl, "\perp"'] \arrow[dr, "X \leq 3"] & \\
\boxed{(\langle \mathit{Err} \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_{\{X=2\}})} & & \boxed{(\langle g \rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X=2\}})}
\end{tikzcd}$$

Figure 5.3.: The comparison edges of $s := (\langle X > 3, f \mid g \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_{\{X=2\}})$

▲

The final set of abstract evaluation rules for which we need edges are those that are used to evaluate the *is*-predicate, namely Abstract Evaluation Rule 4.14 through Abstract Evaluation Rule 4.26. For these, we denote both the relation that is assumed to succeed as well as the substitution that we applied to the remainder of the state on the edge.

Definition 5.4 (Arithmetic assignment edges)**.** Let s be an abstract state to which one of the abstract evaluation rules from Abstract Evaluation Rule 4.14 through Abstract Evaluation Rule 4.26 is applicable, let s'_{err}, s'_{succ}, and s'_{fail} be the states resulting from the application of the abstract evaluation rule, such that $s \Rightarrow s'_{err}$, $s \Rightarrow s'_{succ}$, and $s \Rightarrow s'_{fail}$ hold true. We write $s' = \bot$ if either of these states are not produced by the applicable rule. Furthermore, let σ be the substitution that is applied to s in the case that the evaluation and unification both succeed.

We then define the **arithmetic assignment edges of** s as

$$\begin{aligned} \mathit{ArithmeticAssignmentEdges}(s) := &\{(s, s'_{err}, \bot, \bot) \mid s'_{err} \neq \bot\} \\ &\cup \{(s, s'_{succ}, (t_1 = t_2), \sigma) \mid s'_{succ} \neq \bot\} \\ &\cup \{(s, s'_{fail}, \bot, \bot) \mid s'_{fail} \neq \bot\} \end{aligned}$$

■

Example 5.4 (Arithmetic assignment edges)**.** We pick the abstract state

$$s := (\langle is(X, Y), f(X) \mid g \rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_\emptyset)$$

The only abstract rule that is applicable to s is Abstract Evaluation Rule 4.25, which yields the two abstract successor states $s_{succ} = (\langle f(Z) \mid g \rangle, \{Y, Z\}, \emptyset, \{Y, Z\}, \mathcal{A}_{\{Z=Y\}})$ and $s_{fail} = (\langle g \rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_\emptyset)$ as the successor states of s.

The arithmetic assignment edges of s are formally denoted as

$$\mathit{ArithmeticAssignmentEdges}(s) = \{(s, s_{succ}, X = Y, X \mapsto Z, Y \mapsto Z), (s, s_{fail}, \bot, \bot)\}$$

Note that s only has two arithmetic assignment edges, since Abstract Evaluation Rule 4.12 only produces two successors of s. These edges are shown graphically in Figure 5.4.

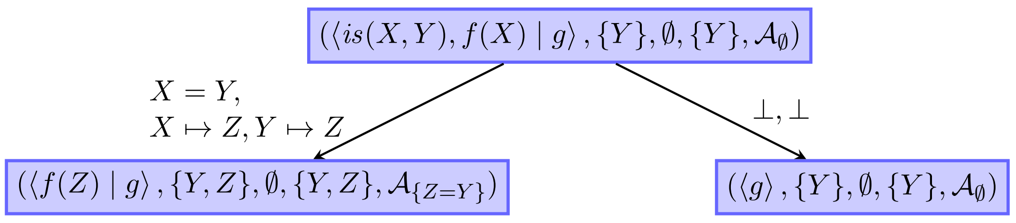

$(\langle is(X, Y), f(X) \mid g \rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_\emptyset)$
$X = Y,$
$X \mapsto Z, Y \mapsto Z$
$\bot, \bot$
$(\langle f(Z) \mid g \rangle, \{Y, Z\}, \emptyset, \{Y, Z\}, \mathcal{A}_{\{Z=Y\}})$
$(\langle g \rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_\emptyset)$

Figure 5.4.: The arithmetic assignment edges of $s := (\langle (X > 3, f \mid g) \rangle, \{Y\}, \emptyset, \{Y\}, \mathcal{A}_{\{X=2\}})$

▲

These edges allow us to construct a graph based completely on the abstract interpretation of a term. However, using only these edges we would not receive a finite graph in nearly all cases, since an infinite concrete evaluation implies an infinite abstract evaluation. Thus, the construction of the graph would never terminate.

In order to tackle this, we introduce additional rules in the following sections that allow us to construct such a finite graph. We start by introducing rules that allow us to avoid repeating very similar states in the following section.

5.1.2. Instantiation and Generalization

In this section we define a set of edges for each state s that allows us to denote in the graph that we have already seen a similar node. This set of edges is necessary in order to construct a finite graph. We call these edges instance- and generalization edges.

In the earlier work of [GSSK+12], these edges were defined as inference rules of the abstract semantics. In this thesis, however, we separate inference rules, which define an abstract semantics, from these instance- and generalization edges in order to have a general-purpose semantics, which is not specifically geared towards termination analysis.

First, we introduce instance edges that allow us to return to a more general abstract state.

Example 5.5 (Infinite graph without instance edges)**.** Consider Program 5.1 and the initial query $q = f(X)$. We construct the graph shown in Figure 5.5a by using the simple edges and the unification edges of each state. Note that the resulting graph is infinite. Thus, an algorithm that tries to construct this graph explicitly will never terminate. ▲

Program 5.1 Example for necessity of instance edges

```
f(X) :- f(f(X)).
```

In Figure 5.5a we see that the set of states represented by $(\langle f(f(Y))\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$ is a subset of those represented by the earlier state $(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$. So, instead of continuing the use of semantic edges when reaching the former state, we could stop at this point, since we already have semantic edges in the graph that describe the evaluation of all its concretizations.

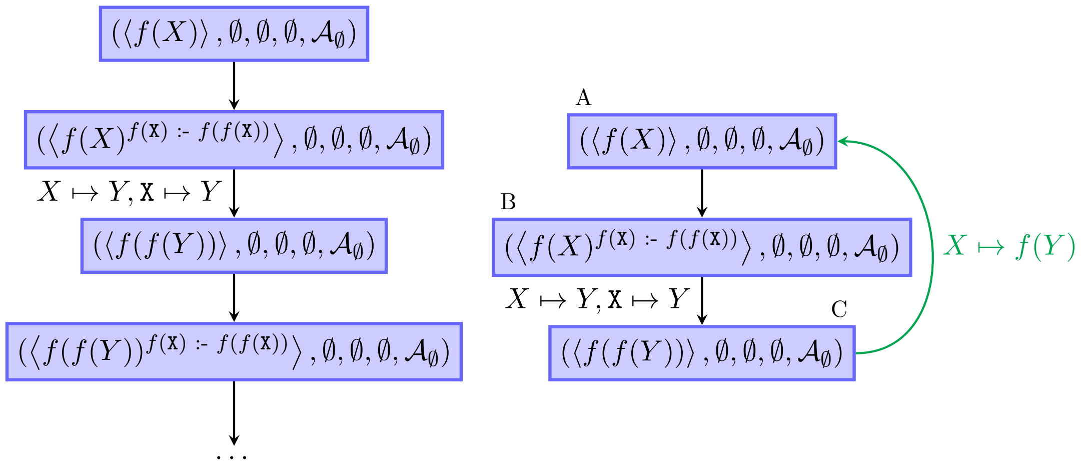
We formalize this intuitive notion of more precise states by using instance edges. Two abstract states s and s' are connected with an instance edge, if s represents a subset of the concrete states represented by s'. If this is the case, we say that s is an instance of s'.

Definition 5.5 (Instance)**.** Let

$$s_{inst} = (S_{inst}, \mathcal{G}_{inst}, \mathcal{U}_{inst}, \mathcal{E}_{inst}, \mathcal{A}_{inst})$$

and

$$s_{gen} = (S_{gen}, \mathcal{G}_{gen}, \mathcal{U}_{gen}, \mathcal{E}_{gen}, \mathcal{A}_{gen})$$


$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$(\langle f(X)^{f(\mathtt{X})\ \text{:-}\ f(f(\mathtt{X}))}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$X \mapsto Y, \mathtt{X} \mapsto Y$
$(\langle f(f(Y))\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$(\langle f(f(Y))^{f(\mathtt{X})\ \text{:-}\ f(f(\mathtt{X}))}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
A
B
C
$X \mapsto f(Y)$

(a) Without instance rule

(b) With instance rule

Figure 5.5.: Graph induced by the query $f(X)$ on Program 5.1

be abstract states. We say that s_{inst} is an **instance** of s_{gen} if there is a substitution $\sigma : \mathcal{T}(s_{gen}) \rightarrow \mathit{Terms}_{\Sigma\setminus\mathcal{V},\mathcal{V}(s_{inst})\cup\mathcal{T}(s_{inst})}$ such that all of the following conditions hold:

$$S_{gen}\sigma = S_{inst} \qquad \mathcal{T}(\mathcal{G}_{gen}\sigma) \subseteq \mathcal{G}_{inst} \qquad \mathcal{U}_{gen}\sigma \subseteq \mathcal{U}_{inst}$$

$$\forall X \in \mathcal{E}_{gen}.\ \mathit{safe}_{\mathcal{E}_{inst},\mathcal{A}_{inst}}(X\sigma) = \mathit{true}$$

$$\forall \delta \in \mathcal{A}_{inst}.\ \mathit{eval}_E \circ \delta \circ \sigma \in \mathcal{A}_{gen}$$

We call σ the **instantiation substitution**. ■

The four former conditions can be checked automatically quite easily, as they all amount to iterations over finite sets. The latter condition, however, poses a challenge for the implementation of this analysis, as arithmetic states are typically infinite. We discuss the implementation of this check in Section 5.3.

Example 5.6 (Instantiation)**.** Consider the two abstract states

$$s_{gen} = \left(\langle f(X_1), g(X_2)\rangle, \{X_1, X_2\}, \emptyset, \{X_1\}, \mathcal{A}_{\{X_1>0\}}\right)$$

and

$$s_{inst} = \left(\langle f(-(Y_1, 1)), g(g(Y_2))\rangle, \{Y_1, Y_2\}, \emptyset, \{Y_1\}, \mathcal{A}_{\{Y_1>0\}}\right).$$

We show that s_{inst} is an instance of s_{gen}.

We pick $\sigma := X_1 \mapsto -(Y_1, 1), X_2 \mapsto g(Y_2)$ and apply σ to $f(X_1), g(X_2)$ to receive $f(-(Y_1, 1)), g(g(Y_2)))$. Thus, the first condition of instantiation holds true.

We also see that $\mathcal{T}(\{X_1, X_2\}\sigma) = \mathcal{T}(-(Y_1, 1), g(Y_2)) = \{Y_1, Y_2\}$ and hence, the second condition also holds true. The third condition amounts to the simple check $\emptyset \subseteq \emptyset$, which vacuously holds true.

The fourth condition holds true, since

$$\mathit{safe}_{\{Y_1\},\mathcal{A}_{\{Y_1>1\}}}(X_1\sigma) = \mathit{safe}_{\{Y_1\},\mathcal{A}_{\{Y_1>1\}}}(-(Y_1, 1)) = \mathit{true}$$

holds true.

Finally, let δ be a member of $\mathcal{A}_{\{Y_1>1\}}$. Then $\delta : Y_1 \mapsto n$ holds true for some $n > 1$. Thus, $\delta \circ \sigma$ maps X_1 to $-(n, 1)$ for some $n > 1$. This implies that $\mathit{eval}_E \circ \delta \circ \sigma$ maps X_1 to $n - 1$. Since $n > 1$, we know that $n - 1 > 0$ holds true, whence $\mathit{eval}_E \circ \delta \circ \sigma$ maps X_1 to some value larger than 0. Hence, $\mathit{eval}_E \circ \delta \circ \sigma$ is a member of $\mathcal{A}_{\{X_1>0\}}$ for all $\delta \in \mathcal{A}_{\{Y_1>0\}}$.

Since all five conditions of Definition 5.5 hold true, s_{inst} is an instance of s_{gen}. ▲

This definition allows us to detect states that are more precise states than those that we have already created during the construction of the termination graph. It remains to show that these conditions actually characterize the intuitive instance-relation.

Lemma 5.1 (Soundness of instantiation)**.** *Let s_{inst} and s_{gen} be abstract states such that s_{inst} is an instance of s_{gen}. Then*

$$\mathit{Conc}(s_{\mathit{inst}}) \subseteq \mathit{Conc}(s_{\mathit{gen}})$$

holds true. ■

Proof. Let $s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ and $s' = (S', \mathcal{G}', \mathcal{U}', \mathcal{E}', \mathcal{A}')$ be abstract states and assume that s is an instance of s' with the instantiation substitution σ. Furthermore let $\mathfrak{s}_{\mathit{conc}}$ be a concrete state in $\mathit{Conc}(s)$. We show that $\mathfrak{s} \in \mathit{Conc}(s')$ holds true.

Since $\mathfrak{s} \in \mathit{Conc}(s)$, there must exist a concretization γ that conforms to s such that $S\gamma = \mathfrak{s}$. As $S'\sigma = S$, we also have $S'\sigma\gamma = \mathfrak{s}$. It remains to show that $\sigma\gamma$ conforms to s'. For this, we show the four conditions from Definition 4.6 in order.

We first show that $\mathcal{V}(X\sigma\gamma) = \emptyset$ holds true for all $X \in \mathcal{G}'$. For this, let $X \in \mathcal{G}'$. We see that due to the definition of an instantiating substitution, we have $\mathcal{T}(X\sigma) \subseteq \mathcal{G}$. Now let $Y \in \mathcal{T}(X\sigma)$. Due to the previous statement, we know that $Y \in \mathcal{G}$ and hence $\mathcal{T}(Y\gamma) = \emptyset$, as γ conforms to s. Hence, we have $\mathcal{T}(X\sigma\gamma) = \emptyset$.

The next condition to show is $\mathit{mgu}(t_1\sigma\gamma, t_2\sigma\gamma) = \bot$ for all pairs of terms $(t_1, t_2) \in \mathcal{U}'$. Due to the definition of instantiating substitutions, we know that $(t_1\sigma, t_2\sigma) \in \mathcal{U}$ holds true as well. Since γ conforms to s, we can conclude that $\mathit{mgu}(t_1\sigma\gamma, t_2\sigma\gamma) = \bot$ holds true.

We now continue to show that $\mathit{eval}_E(X\sigma\gamma) \neq \bot$ holds true for all $X \in \mathcal{E}'$. We know that $\mathit{safe}_{\mathcal{E},\mathcal{A}}(X\sigma) \equiv \mathit{true}$, since σ is a instantiating substitution. Hence, we can conclude that $\mathit{eval}_E(X\sigma\gamma) \neq \bot$ holds true, due to Lemma 4.3, or, more precisely, due to Statement 4.1.

The final condition that we have to show is that $(\mathit{eval}_E \circ \sigma\gamma)\big|_{\mathcal{T}}$ is an element of $\mathcal{A}'$. Since $\sigma\gamma$ is just a shorthand for $\gamma \circ \sigma$, this amounts to showing that $(\mathit{eval}_E \circ \gamma \circ \sigma)\big|_{\mathcal{T}}$ is an element of $\mathcal{A}'$. We know that γ conforms to s, which implies that $(\mathit{eval}_E \circ \gamma)\big|_{\mathcal{T}}$ is a member of $\mathcal{A}$, due to Definition 4.6. Furthermore, since s is an instance of s', it holds true that $\mathit{eval}_E \circ \delta \circ \sigma$ is a member of $\mathcal{A}'$ for all $\delta \in \mathcal{A}$. We combine these two facts and receive that $eval_E \circ \left[(eval_E \circ \gamma)\big|_{\mathcal{T}}\right] \circ \sigma$ is a member of $\mathcal{A}'$. Due to the restriction of $eval_E \circ \gamma$ to $\mathcal{T}$, the resulting function $eval_E \circ \left[(eval_E \circ \gamma)\big|_{\mathcal{T}}\right] \circ \sigma$ is undefined on all $X \notin \mathcal{T}$. We can extend the restriction to the whole function and write it as $(eval_E \circ eval_E \circ \gamma \circ \sigma)\big|_{\mathcal{T}}$. Since $eval_E$ is idempotent[1], we can drop its second application and receive that $eval_E \circ \left[(eval_E \circ \gamma)\big|_{\mathcal{T}}\right] \circ \sigma = (eval_E \circ \gamma \circ \sigma)\big|_{\mathcal{T}}$ holds true. Finally, since we have previously argued that the former function is a member of $\mathcal{A}'$, the latter function is a member of it as well. Thus, we have shown that $\sigma\gamma$ fulfills the final condition of the definition of conforming substitutions.

We have shown that $\sigma\gamma$ fulfills all conditions of Definition 4.6 and hence, that $\sigma\gamma$ conforms to s'. Hence, we conclude that $\mathfrak{s} \in Conc(s')$ holds true. Since we picked $\mathfrak{s}$ arbitrarily from $Conc(s)$, we have shown that Lemma 5.1 holds true. □

If we include such an instance edge in the graph, we label it with the instantiating substitution.

Definition 5.6 (Instance edge)**.** Let s_{inst} and s_{gen} be abstract states. We define the **instance edge** between s_{inst} and s_{gen} as

$$\begin{aligned}&\mathit{InstanceEdge}(s_{inst}, s_{gen}) = \\ &\quad \{(s_{inst}, s_{gen}, \sigma) \mid s_{inst} \text{ is an instance of } s_{gen} \text{ with the instantiating substitution } \sigma\}\end{aligned}$$

■

Example 5.7 (Finite graph with instance edges)**.** Consider Program 5.1 again. We construct the graph shown in Figure 5.5b using the simple edge of node A and the unification edge of node B. Instead of using the simple edge of node C, we instead use the instance edge of nodes C and A with the instantiating substitution $X \mapsto f(Y)$. Thus, we have constructed a finite termination graph that represents the infinite abstract evaluation of Program 5.1. ▲

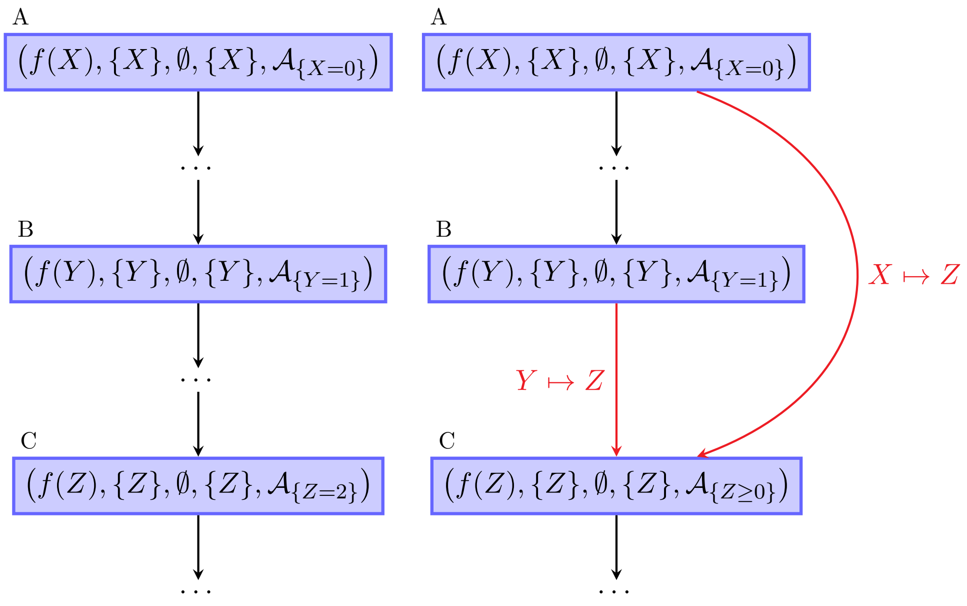
Although instantiation edges are the major concept that we use to construct finite graphs, it is not sufficient to only take these edges back to already existing, more general states in the graph. Such edges only allow us to go back to strictly more general states that we already visited, but not to any "similar" states.

Example 5.8 (Infinite graph without generalization edges)**.** Consider Program 5.2. We construct the graph shown in Figure 5.6a from the starting query $f(0)$. There exists no pair of nodes between which an instance edge can be drawn, since the arithmetic state in every state fixes a precise value n for the single argument of the function symbol f. Since neither $X = n$ implies $X = n + 1$, nor vice versa, it is not possible to connect two such states with an instance edge. ▲

[1] This is due to the fact that $eval_E$ maps terms to $\mathbb{Z}$. According to Definition 3.8, it also maps integers to themselves. Thus, multiple applications of this function do not change the result, hence it is idempotent.

Program 5.2 Example for necessity of generalization edges

```
f(X) :- is(Y,X+1), f(Y).
```



A
$(f(X), \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X=0\}})$
B
$(f(Y), \{Y\}, \emptyset, \{Y\}, \mathcal{A}_{\{Y=1\}})$
C
$(f(Z), \{Z\}, \emptyset, \{Z\}, \mathcal{A}_{\{Z=2\}})$
A
$(f(X), \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X=0\}})$
$X \mapsto Z$
B
$(f(Y), \{Y\}, \emptyset, \{Y\}, \mathcal{A}_{\{Y=1\}})$
$Y \mapsto Z$
C
$(f(Z), \{Z\}, \emptyset, \{Z\}, \mathcal{A}_{\{Z\geq 0\}})$

(a) Without generalization rule (b) With generalization rule

Figure 5.6.: Graph of Program 5.2

In order to go back from a state to a similar one, we first need to construct a state that is more general than both of them. We can then use the known concept of instantiation to transition to the more general state. We formalize this idea using generalization edges. These edges connect two similar states to such a more general state.

Definition 5.7 (Generalization edges)**.** Let s_{inst}, s'_{inst} and s_{gen} be abstract states. We define the **generalization edges** of s_{inst}, s'_{inst} and s_{gen} as

$$\begin{aligned}&\mathit{GeneralizationEdges}(s_{inst}, s'_{inst}, s_{gen}) := \\ &\quad \{(s_{inst}, s_{gen}, \sigma) \mid s_{inst} \text{ is an instance of } s_{gen} \text{ with the instantiating substitution } \sigma\} \\ &\cup \{(s'_{inst}, s_{gen}, \sigma') \mid s'_{inst} \text{ is an instance of } s_{gen} \text{ with the instantiating substitution } \sigma'\}\end{aligned}$$

■

This definition poses a problem to the practical implementation of this analysis, since there are infinitely many states s_{gen} that generalize existing states. In our implementation, we use heuristics for finding suitable states to generalize existing ones to. The use of these heuristics in the construction of the termination graphs is shown in Section 5.1.4. We briefly discuss the heuristics themselves in Section 5.3.

Example 5.9 (Finite graph with generalization edges)**.** Consider Program 5.2 again. Using generalization edges we can now construct the graph shown in Figure 5.6b, where we generalize states A and B to state C. We draw these generalization edges in red. ▲

In this section we have defined the concepts of instantiation and generalization that allow us to "backtrack" to already visited, similar states in the graph. However, these concepts are only useful if there already exist states that are candidates for instantiation or generalization. There may also occur situations in which no such states ever exist. In the next section, we treat such cases.

5.1.3. Splitting and Parallelization

In the previous section, we introduced instance- and generalization edges that allow us to transition to an existing state if this existing state is similar enough to a newly constructed one. The notion of "similar enough" was defined using the instantiation-relation.

This relation is very useful in a lot of cases, but there is one major downside to it. It requires that the two states have the same number of goals and all goals have the same number of terms in the two states. This is a problem in programs that do not terminate due to unbounded growth of the number of goals or the number of terms in a goal.

Example 5.10 (Infinite graph without split edges)**.** Consider Program 5.3. We construct the graph shown in Figure 5.7a from this program and the starting query $f(X)$. This graph is infinite, since every new application of the single rule of the program adds one more term $g(X)$ to the goal. Thus, the single goal of the state grows without any bounds and we will receive an endless sequence of states.

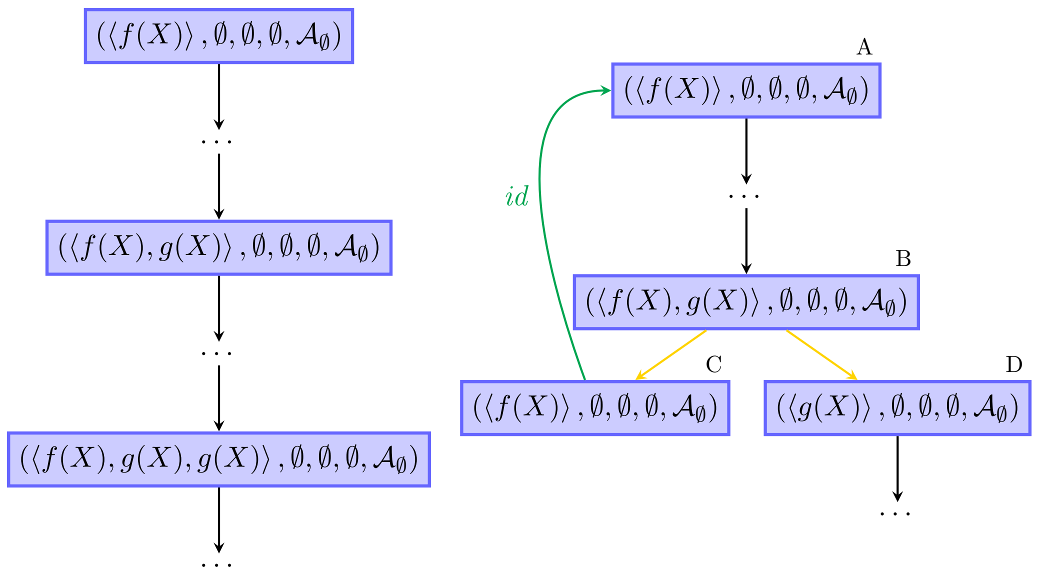
Program 5.3 Example for necessity of split edges

```
f(X) :- f(X), g(X).
```

It is not possible to apply instance- or generalization edges, since there is no way to create or remove terms or goals using substitutions. Since each node has one term more than the previous one, there is no instantiating substitution for any pair of states. Also, for the same reason, there does not exist a state that we could generalize two states to. ▲

In order to create a finite graph even in these situations, we use the concept of a split. The idea is to split off the first term of the first goal of a state in order to treat it separately.

Without such edges, each path through the graph corresponds to an abstract evaluation. At split edges, however, an evaluation is represented by a traversal of the left-hand successor of the node first, after which the right-hand successor of the node has to be traversed.


$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
. . .
$(\langle f(X), g(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
. . .
$(\langle f(X), g(X), g(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
. . .
A
$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
id
. . .
B
$(\langle f(X), g(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
C
$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
D
$(\langle g(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
. . .

(a) Without split edges (b) With split edges

Figure 5.7.: Graph of Program 5.3

Example 5.11 (Reconstructing executions from split edges)**.** Consider the abstract state

$$s := (\langle f(X,Y), g(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

which we split up into the abstract states

$$s'_1 = (\langle f(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$
$$s'_2 = (\langle g(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

This split is shown graphically in Figure 5.8, where the split-edges are drawn in yellow. In order to model the evaluation of node A, we first have to descend into node B. If the evaluation of this node finishes at some point, we have to return to node B and descend into node C. This order of evaluation is denoted by the dashed path.

Note that we first show termination of state B and only if we could successfully show the termination of this state, we continue to show the termination of state C. Thus, when we analyze state C we may use the assumption that we have already shown that state B terminated. ▲

In this example we have simply copied the knowledge base of the split state to the two split results. In general, however, this would be unsound, as we can see in the following example.

Example 5.12. Consider the abstract state

$$s := (\langle is(\texttt{X}, 2), is(\texttt{X}, 3)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

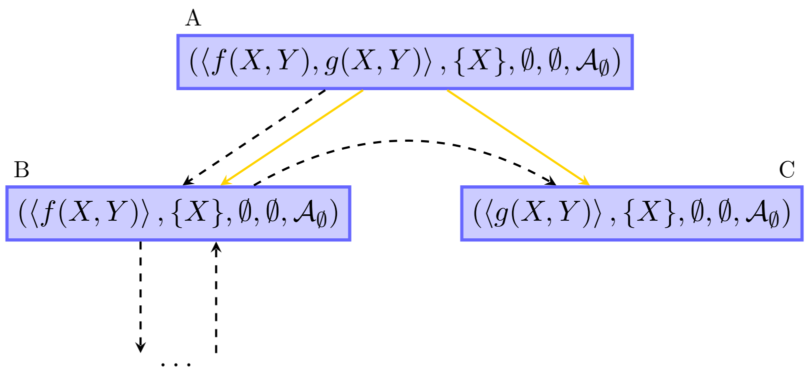

A
$(\langle f(X,Y), g(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
B
$(\langle f(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
C
$(\langle g(X,Y)\rangle, \{X\}, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...

Figure 5.8.: Example of a split

which only represents the concrete state $\langle f(2, \mathtt{X}), \mathit{is}(\mathtt{X}, 3)\rangle$. This concrete state evaluates as follows:

$$\langle \mathit{is}(\mathtt{X}, 2), \mathit{is}(\mathtt{X}, 3)\rangle \qquad \to \qquad \langle \mathit{is}(2, 3)\rangle \qquad \to \qquad \langle \epsilon\rangle$$

If we split the abstract state s as we did in the previous example, we would receive the two abstract states

$$s'_1 = (\langle \mathit{is}(\mathtt{X}, 2)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$
$$s'_2 = (\langle \mathit{is}(\mathtt{X}, 3)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

which does not fit our intuition about the split anymore, since, after evaluating $\mathit{is}(\mathtt{X}, 2)$ we end up in $\langle \mathit{is}(2, 3)\rangle$. This state is not represented by s'_2, since $\mathtt{X}$ is a program variable. ▲

The previous example showed us that we need to model the effect that the evaluation of the first term of the first goal of the abstract state has on the remainder of the goal. In order to do so, we replace all those variables that are effected by the evaluation of this predicate by fresh term variables. This models the fact that the evaluation of the first predicate may assign arbitrary terms to its arguments.

In order to make this construction more precise, we additionally use a function for groundness analysis. This function takes a program and a predicate as well as a set of input variables that are known to be ground. It returns those variables that are known to be ground after the evaluation of the predicate. Such a function is presented, for example, in [HK03]. In order to preserve soundness of the transformation of the resulting graph into integer transition systems later on, we only define these edges for states containing only a single goal.

Definition 5.8 (Split states, split edges)**.** Let $s = (\langle t, T\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. We define the **split states** s_{split} and s'_{split} of s as

$$s_{\mathit{split}} := (\langle t\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$$
$$s'_{\mathit{split}} := (\langle T\sigma\rangle, \mathcal{G}\sigma \cup \mathit{NextG}(t, \mathcal{G})\sigma, \mathcal{U}\sigma, \mathcal{E}\sigma, \mathcal{A}\sigma)$$

where $NextG(t, \mathcal{G})$ is a function returning those abstract variables that are known to be ground after the evaluation of t with the ground variables $\mathcal{G}$ and σ is a function that replaces all variables with previously unused term variables.

We define the **split edges** of s as

$$SplitEdges(s) := \{(s, s_{split}), (s, s'_{split})\}$$

■

Example 5.13 (Split edges)**.** Consider Program 5.3 again. We construct the graph shown in Figure 5.7b from the starting query $f(X)$ by first using simple and evaluation edges from node A and then using the split edges of node B. Using these split edges, we split node B into nodes C and D. We can then use the instance edge of node C to connect it back to node A with the identity function as the instantiating substitution. The evaluation of node D can continue as usual, using simple, unification, instantiation, generalization and split edges as needed. ▲

We have argued the soundness of this rule informally in the previous examples. It remains to show its soundness formally with respect to termination analysis. This soundness is formalized in the following statement.

Lemma 5.2 (Soundness of the split rule)**.** *Let s be an abstract state for which split edges are defined and let $(s, s_{split}), (s, s'_{split})$ be its split edges. If s is nonterminating, then either s_{split} or s'_{split} is nonterminating.* ■

Proof. Since $s = (\langle t, T\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ is nonterminating, there exists a nonterminating concrete state $s_{conc} = \langle t\gamma, T\gamma\rangle$ that is a member of $Conc(s)$, where γ is a substitution that conforms to s. Consider the infinite sequence of concrete state $s_1 \to s_2 \to \ldots$, with $s_1 = s_{conc}$. It is then either the case that all s_i are of the form $\langle T'_1, T\gamma\gamma' \mid \cdots \mid T'_i, T\gamma\rangle$ or that there is some first state of the form $\langle T\gamma\rangle$.

In the former case, we see that the suffix $T\gamma$ is not changed during the remainder of the evaluation. Hence, for each $s_i = \langle T'_1, T\gamma \mid \cdots \mid T'_i, T\gamma\rangle$, the concrete state $\langle t\gamma\rangle$ evaluates to $\langle T'_1 \mid \cdots \mid T'_i\rangle$ holds true. Since there are an infinite number of such s_i, $\langle t\gamma\rangle$ is nonterminating. Furthermore, since the knowledge bases of s and s_{split} are identical and since γ conforms to s, γ also conforms to s_{split}. Hence, $\langle t\gamma\rangle$ is a member of $Conc(s_{split})$, whence this abstract state is nonterminating.

In the latter case we know that $s_{conc} = \langle t\gamma, T\gamma\rangle$ evaluates to $\langle T\gamma\gamma'\rangle$. The concrete state $\langle T\gamma\gamma'\rangle$ is nonterminating due to our assumption. If this is the case, then s'_{split} is nonterminating, as we now show. Due to Lemma 5.3, $\sigma^{-1}\gamma\gamma'$ conforms to s'_{split}, whence $\left\langle T\sigma\sigma^{-1}\gamma\gamma'\right\rangle$ is a member of $Conc(s'_{split})$. Furthermore, since $\langle T\gamma\gamma'\rangle$ is nonterminating, and since $\sigma\sigma^{-1}\gamma\gamma' = \gamma\gamma'$ holds true, we know that $\left\langle T\sigma\sigma^{-1}\gamma\gamma'\right\rangle$ is nonterminating. Since this nonterminating state is a member of $Conc(s'_{split})$, the abstract state s'_{split} is nonterminating. □

In this proof we used the following additional lemma. Its proof can be found in Appendix A.

Lemma 5.3. *Let $s = (\langle t, T\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state, let NextG be a sound groundness analysis function and let σ be a function that replaces all variables in $\langle t, T\rangle$ with fresh term variables. Furthermore, let γ be some concretization of s. If $\langle t\gamma, T\gamma\rangle$ evaluates to $\langle T\gamma\gamma'\rangle$, then $\sigma^{-1}\gamma\gamma'$ conforms to $s' = (\langle T\sigma\rangle, \mathcal{G}\sigma \cup NextG(t, \mathcal{G})\sigma, \mathcal{U}\sigma, \mathcal{E}\sigma, \mathcal{A}\sigma)$.* ■

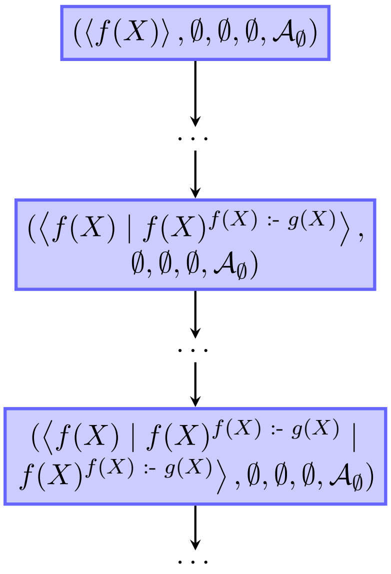
We have just seen how we can break up goals into their individual terms using split rules. This solved a problem where a goal would grow without any bound. The same thing may happen with the state itself, which consists of a sequence of goals, as we will see in the following example.

Example 5.14 (Infinite graph without parallel edges)**.** Consider Program 5.4. We construct the graph shown in Figure 5.9a from this program. Note that this graph is infinite, since every application of the case-rule results in another goal that is added to the state. It is not possible to use instance- or generalization rules in order to make this graph finite, since these rules cannot be applied to states with differing numbers of goals.

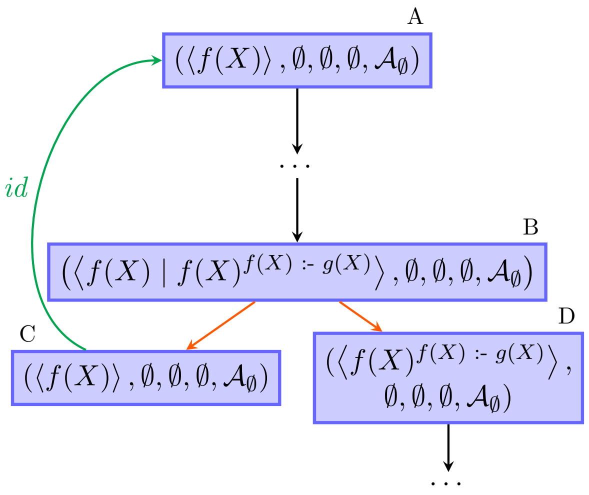
Program 5.4 Example for necessity of parallel edges

```
f(X) :- f(X).                          f(X) :- g(X).
```

▲


$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
$(\langle f(X) \mid f(X)^{f(X) \,:\!-\, g(X)}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
$(\langle f(X) \mid f(X)^{f(X) \,:\!-\, g(X)} \mid f(X)^{f(X) \,:\!-\, g(X)}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...

(a) Without parallel edges


A
$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
id
...
B
$(\langle f(X) \mid f(X)^{f(X) \,:\!-\, g(X)}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
C
$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
D
$(\langle f(X)^{f(X) \,:\!-\, g(X)}\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...

(b) With parallel edges

Figure 5.9.: Graph of Program 5.4

In order to handle such situations as in the previous example, we introduce parallel edges. The idea behind these edges is to treat each goal in parallel. This is very similar to the split edges defined previously. However, the evaluation of a complete goal does not have an impact on other goals, except for the evaluation of a cut. Thus, we are able to define the rule in a simpler way than the previous one, without relying on a groundness analysis.

The only thing to consider for soundness is the reachability of cuts. We must not be able to reach cuts in the resulting states that were not reachable in the original state. For this, we first define active cuts and active marks, which we then use to define the applicability of parallelization.

Definition 5.9 (Active cuts, active marks, parallel states, parallel edges)**.** Let g be a goal and let G be a sequence of goals. We define the **active cuts** of g as the set of those m, for which $g = T_1 \mid !_\mathrm{m} \mid T_2$ holds true for some sequences of terms T_1 and T_2. We also define the **active marks** of G as the set of those m, for which $G = G_1 \mid ?_\mathrm{m} \mid G_2$ holds true for some sequences of goals G_1 and G_2. These definitions are taken directly from [Str10, Definition 3.45].

Let $s = (\langle g \mid G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. If the active cuts of g and the active marks of G are disjoint, then we define the **parallel states** s_{par} and s'_{par} of s as

$$\begin{aligned} s_{par} &:= (\langle g \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \\ s'_{par} &:= (\langle G \rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \end{aligned}$$

Otherwise, we define $s_{par} = s'_{par} = \bot$.

We then define the **parallel edges of** s as

$$\mathit{ParallelEdges}(s) := \{(s, s_{par}), (s, s'_{par}) \mid s_{par} \neq \bot, s'_{par} \neq \bot\}$$

■

Example 5.15 (Finite graph without parallel edges)**.** Consider Program 5.4 again. Using the starting query $f(X)$, we construct the graph shown in Figure 5.9b from it. First, we use the simple- and unification edges of node A and its successor to receive node B. We use the parallel edges of node B and receive nodes C and D. Node C can be lead back to node A using the instance edge of node C and node A directly. Afterwards, we can continue constructing the graph from node D, which we can do using only simple- and unification edges. ▲

We now have the possibility to split up sequences of goals and terms in order to treat each element individually. There is, however, one technical detail to consider, which concerns the cut and its treatment across parallel edges. It may be the case that, by using parallel edges, we construct states that contain a cut, but no end-of-scope marker for the cut.

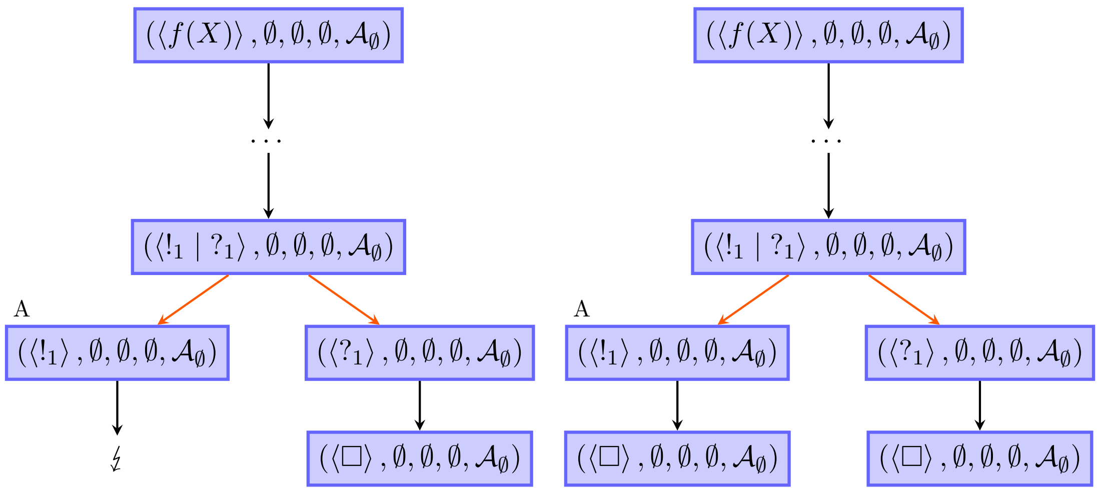
Example 5.16 (Parallelizing the end-of-scope marker)**.** Consider Program 5.5 and the initial query $q = f(X)$. Using a simple- and a unification edge, we receive the abstract state

$$s = (\langle !_1 \mid ?_1 \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

If we decide at this point to use a parallel edge, we receive the graph shown in Figure 5.10a. We see that node A has no edges that we can use anymore, since we would need to evaluate a cut without its corresponding end-marker.

Program 5.5 Example for parallelization of end-of-scope marker

```
f(X) :- !.
```



$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
$(\langle !_1 \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
A
$(\langle !_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$(\langle ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$\lightning$
$(\langle \square\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$

(a) Without handling of cut without scope (b) With handling of cut without scope

Figure 5.10.: Parallelization of end-of-scope marker

▲

Since the only way such a state may occur is by the application of the parallel rule, there is a simple remedy. We can simply use an "imaginary" end-of-scope marker at the end of the list of goals. This is formalized using cut edges.

Definition 5.10 (Cut edge)**.** Let $s = (\langle !_m, T \mid G\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. We define the cut edge of s as

$$\mathit{CutEdge}(s) := \{(s, s') \mid G \text{ does not contain } ?_m\},$$

where s' is defined as $(\langle T\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$. ■

Example 5.17 (Parallelizing the end-of-scope marker)**.** Consider again Program 5.5 and the starting query $q = f(X)$. We construct the graph shown in Figure 5.10b from this program and this query. Using the cut edge of node A, we are now able to finish the construction of the graph at node A. ▲

We have now defined all the edges that we need for the construction of a complete termination graph. In the next section, we are going to describe this construction in detail.

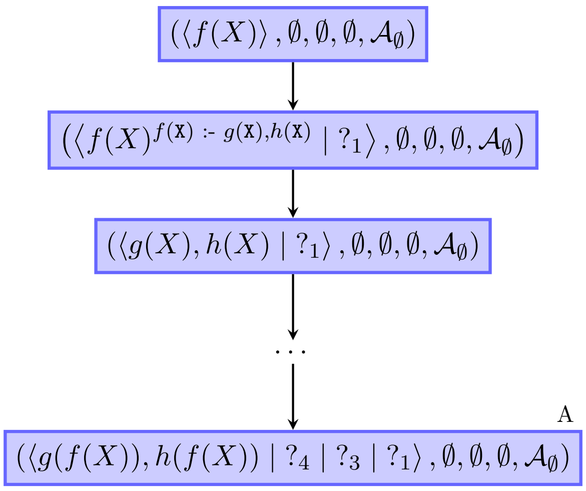

$(\langle f(X)\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$(\langle f(X)^{f(\mathtt{X})\ \mathtt{:-}\ g(\mathtt{X}), h(\mathtt{X})} \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
$(\langle g(X), h(X) \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$
...
A
$(\langle g(f(X)), h(f(X)) \mid ?_4 \mid ?_3 \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$

Figure 5.11.: Unfinished graph of Program 5.6

5.1.4. Construction of Termination Graphs

In the previous section, we have defined multiple types of edges for each state. Some of these edges depend on the structure of the first term of the first goal of the state, such as the simple and unification edges. Others depend on the relation between the state and other states, such as the instance- and generalization edges. Again others are only applicable if the state is large enough and split it up to be treated separately, such as the split- and parallel edges. There is no state for which all of these edges are defined. However, for nearly all states, more than one kind of edges is defined.

Program 5.6 Example for necessity of split edges

```
f(X) :- g(X), h(X).                h(X) :- f(f(X)).
g(X).
```

Example 5.18 (Multiple outgoing edges for a state)**.** Consider Program 5.6 and its partially constructed graph in Figure 5.11. Node A has multiple outgoing edges.

First of all, Abstract Evaluation Rule 4.2 is applicable to A, which would result in the abstract state

$$s_{sem} := \left(\left\langle (g(f(X)), h(f(X)))^{g(\mathtt{X})\ \mathtt{:-}\ \square} \mid ?_4 \mid ?_3 \mid ?_1 \right\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset\right)$$

Node A also has split edges, which would result in the split states

$$\begin{aligned} s_{split} &= (\langle g(f(X))\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset) \\ s'_{split} &= (\langle h(f(X)) \mid ?_4 \mid ?_3 \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset) \end{aligned}$$

It furthermore has parallel edges that allow us to construct the parallel states

$$\begin{aligned} s_{par} &= (\langle g(f(X)), h(f(X))\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset) \\ s'_{par} &= (\langle ?_4 \mid ?_3 \mid ?_1\rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset) \end{aligned}$$

This shows that there exist states for which multiple types of outgoing edges are defined. In fact, this statement holds for most states. ▲

In order to analyze a program for termination, we need to construct a finite graph explicitly using the edges that we have defined in the previous sections. We provide a framework for an algorithm that does precisely that in this section. This framework is used in our implementation of the termination analysis. Since we reuse parts of the implementation from [Str10], we only provide the general framework and point to that work for a precise description of the deterministic algorithm that we use for the construction of the graph.

The most important requirement for the resulting graph is that it is finite and that its construction terminates. The secondary requirement is that the graph shall be as concise as possible. Furthermore it shall describe the potentially infinite set of abstract states as precisely as possible.

The complete algorithm for the construction of a termination graph is formalized as CONSTRUCTGRAPH($\mathcal{P}, Q$), which takes a program P and a query q and returns a termination graph of the program and the query. We give a description of this algorithm in pseudocode in Algorithm 5.1.

Algorithm 5.1 High level graph construction

```
function ConstructGraph(Prolog program $\mathcal{P}$, Query $q$)
    $graph \leftarrow (\{s^q_{init}\}, \emptyset)$
    $workList \leftarrow [s^q_{init}]$
    while $workList$ is not empty do
        $currentState \leftarrow$ popped head of $workList$
        $allEdges \leftarrow$ GetAllEdges($currentState, graph$)
        $pickedEdges \leftarrow$ PickEdges($allEdges, currentState, graph$)
        for all $edge$ in $pickedEdges$ do
            $newVertices \leftarrow graph.$Vertices $\cup$ $edge.$Target
            $newEdges \leftarrow graph.$Edges $\cup \{edge\}$
            if $edge.$Target is not contained in $graph$ then
                add $edge.$Target to $workList$
            $graph \leftarrow (newVertices, newEdges)$
    return $graph$
```

This algorithm is nothing but a simple construction of the graph using the standard graph traversal. It starts with the initial state of the abstract evaluation as defined in Definition 4.7 on page 37. From there it traverses the tree and constructs it along the way using a standard graph traversal.

For every node we first construct a set of outgoing edges in GETALLEDGES. Since there are usually infinitely many such edges, especially regarding generalization edges, this function constructs a finite set of sets of edges. We show our implementation of GETALLEDGES in Algorithm 5.2.

Algorithm 5.2 Construct possible edges from node

```
function GetAllEdges(Abstract state $s$, Graph $G$)
    $edges \leftarrow \{SimpleEdges(s)\}$
    $edges \leftarrow edges \cup \{UnificationEdges(s)\}$
    $edges \leftarrow edges \cup \{ComparisonEdges(s)\}$
    $edges \leftarrow edges \cup \{ArithmeticAssignmentEdges(s)\}$
    for all States $s'$ in $G$ do
        $edges \leftarrow edges \cup \{InstanceEdge(s, s')\}$
        $genCandidate \leftarrow$ GeneralizationCandidate$(s, s')$
        if $genCandidate \neq \bot$ then
            $edges \leftarrow edges \cup \{GeneralizationEdges(s, s', genCandidate)\}$
    $edges \leftarrow edges \cup \{SplitEdges(s)\}$
    $edges \leftarrow edges \cup \{ParallelEdges(s)\}$
    return $\{E \mid E \in edges$ and $E \neq \emptyset\}$
```

This implementation first constructs those edges that are based on the abstract semantics. It then tries to find another already existing node s' in the graph such that the new node is an instance of this node and also stores all possible instance edges. It also attempts to construct a more general state that describes both the newer state and the already existing one in the function GeneralizationCandidate. If such a candidate is found, the corresponding generalization edges are stored as well. The implementation finally checks if it is possible to split or parallelize the current node and stores the corresponding edges, if this is the case.

The choice of GeneralizationCandidate is crucial for making this termination analysis automatic. We reuse large parts of the choice of this function in [Str10, Section 6.2], but adapt it slightly to take the arithmetic components of the knowledge base into account. This extension is described in more detail in Section 5.3.

Example 5.19 (Execution of GetAllEdges)**.** We consider the graph under construction shown in Figure 5.11 again, which we call G. Let s be the node labeled with A, namely

$$s = (\langle g(f(X)), h(f(X)) \mid ?_4 \mid ?_3 \mid ?_1 \rangle, \emptyset, \emptyset, \emptyset, \mathcal{A}_\emptyset)$$

As described in the previous example, this node's simple edges, as well as its split- and parallel edges are defined. Thus, the function GetAllEdges(s, G) returns the following set of sets of edges:

$$returnValue = \{\{(s, s_{sem})\}, \{(s, s_{split}), (s, s'_{split})\}, \{(s, s_{par}), (s, s'_{par})\}\}$$

where s_{sem}, s_{split}, s'_{split}, s_{par} and s'_{par} are defined as they were in the previous example.
▲

Out of this selection of sets of edges, the function PickEdges picks a single set which is then added to the graph, along with the new states that it connects to. For our implementation, we chose a function that is heuristics-driven, which works very well in practice. The choice of this function does not impact the soundness of the graph construction, but only its termination. Any reasonable implementation of PickEdges should pick the edges in such a way that it guarantees termination of the complete construction. Similarly to GeneralizationCandidate, we use large parts of the heuristic presented in [Str10, Section 6.2] for this. We Its termination is shown in [Str10, Section 6.3].

Definition 5.11 (Termination graph)**.** Let P be a program and let q be a query. We call any finite graph that is constructed using Algorithm 5.1 using any implementation of PickEdges a **Termination Graph of P on q.** ■

Program 5.7 Computation of the factorial with cut (reprint of Program 2.3)

```
fac(X, Y) :- X > 0, !, fac(X - 1, Y1), Y is Y1 * X.
fac(X, Y) :- X =:= 0, Y is 1.
```

Example 5.20 (Termination graph of Program 2.3)**.** Consider Program 2.3 on page 13, which is reprinted as Program 5.7 for the sake of readability. We present a termination graph of this program on $fac(X_1, X_2)$ in Figure 5.12.

▲

Each termination graph represents an overapproximation of the set of abstract runs of the program. Thus, if there is a nonterminating run of the original program, this run is also represented by the termination graph. It would be possible to analyze this graph directly for such a run. We choose, instead, to transform it into the well-studied formalism of an integer transition system, which we can then analyze for termination using well-known off-the-shelf techniques.

5.2. Transformation of Termination Graphs into Integer Transition Systems

We have defined a transformation of programs into termination graphs in the previous section. The termination graph of a program on a query is a finite description of all possible evaluations of the program on the query.

Thus, we now need a method to determine whether or not a given termination graph describes a nonterminating run. In order to do so, we transform the termination graph into an integer transition system, or *ITS* for short. An integer transition system is a very simple formalism for which the problem of deciding termination has been well studied. The idea is that termination of the *ITS* implies termination of all runs described by the termination graph. This in turn implies termination of the program on the given query.

We first give a short introduction to the formalism of integer transition systems in the following section. Following that, we show the construction of an *ITS* from a termination graph in Section 5.2.2. In Section 5.2.3 we prove that the termination of this *ITS* implies the termination of all runs of the program. Finally, we give a pointer to the literature

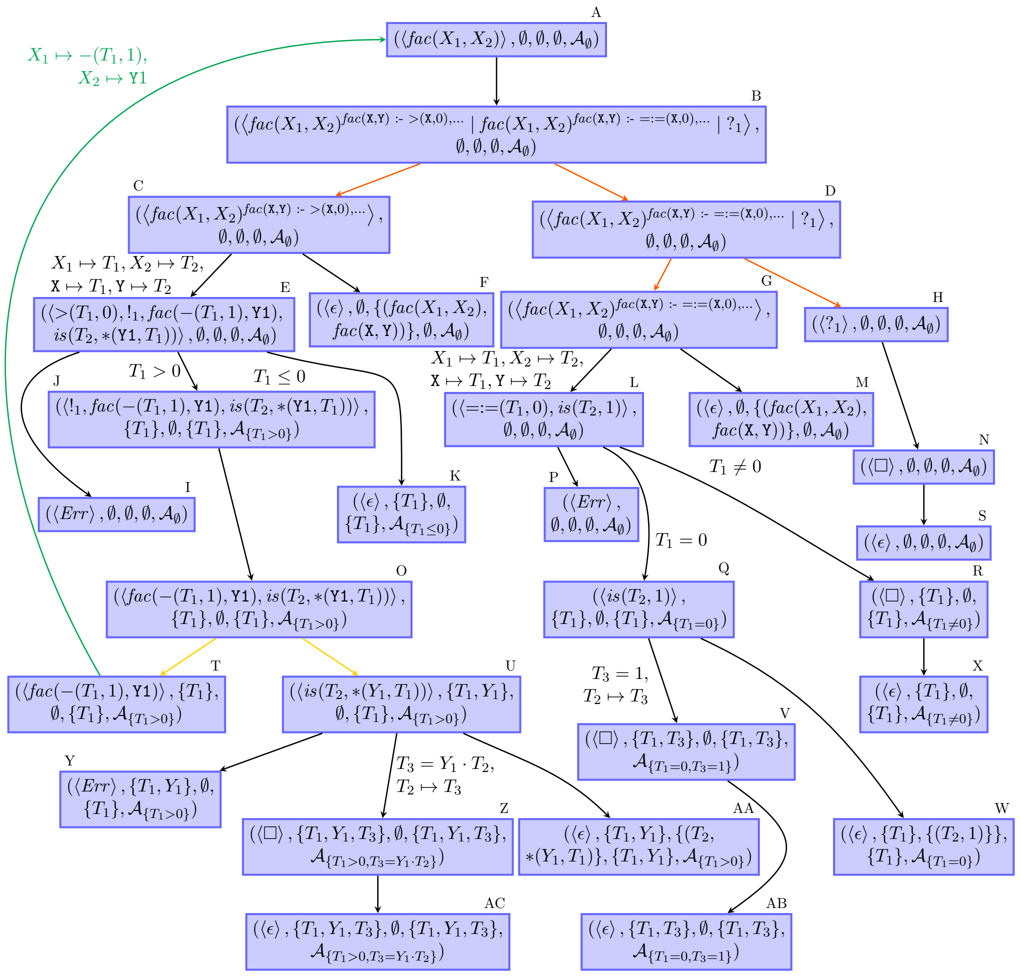


Figure 5.12.: A termination graph of Program 5.7 on $fac(X_1, X_2)$

on techniques for the termination analysis of *ITS*s in Section 5.2.4. We are, however, going to use these techniques as a black box.

5.2.1. Integer Transition Systems

An integer transition system, or *ITS* for short, is a very simple formalism. Similar to a Prolog program, such a system consists of a set of rules that describe admissible transformations of terms. In contrast to Prolog, however, these rules may also have a condition on the values of the integers occurring in the terms. It is only possible to apply this rule if the condition evaluates to true.

Just like in Prolog, the only data structure appearing in integer transition systems are terms. A computation of an integer transition system consists of a sequence of applications of its rules, beginning with some given start term.

Example 5.21 (Integer transition system)**.** The following set of rules is an integer transition system:

$$\begin{array}{rcll} f(X,Y) & \to & g(X-1,Y) & \mid X > 0 \\ g(X,h(Y)) & \to & f(X,Y) & \mid \mathit{true} \end{array}$$

Consider the term $t_1 := f(2,h(h(i)))$, where i is come constant. We can apply the first rule to t_1 to receive the term $t_2 = g(1,h(h(i)))$. We cannot apply the first rule to t_2, since t_2 does not match the left-hand part of that rule. Instead, we can apply the second rule to t_2 to receive the term $t_3 = f(1,h(i))$.

Another round of applications of the first and second rule yields the terms $t_4 = g(0,h(i))$ and $t_5 = f(0,i)$. At this point, there is no rule that we can apply anymore. We cannot apply the first rule, since $0 > 0$ does not hold true. We can also not apply the second rule, as its left-hand side does not match the term t_5. Hence, the computation has terminated. ▲

An *ITS* is very similar to a Prolog program without cuts. Its basic building blocks are nearly identical, although rules are of a much simpler form, since they only allow a single term on both sides. The conditions consist of a finite conjunction of relations over the integers and variables. The only variables that may occur in conditions are those that already occur in the left- or right-hand side of a rule.

Definition 5.12 (Terms, conditions, rules)**.** Let $\mathcal{V}$ be a countably infinite, nonempty set of variables. We define the set $\mathit{ITSExpressions}_{\mathcal{V}}$ as the smallest superset of $\mathcal{V} \cup \mathbb{Z}$ that fulfills the following two conditions:

For all $e \in \mathit{ITSExpressions}_{\mathcal{V}}$ **it holds true that**

$$\{\mathit{abs}(e), \mathit{sign}(e), (-e)\} \subseteq \mathit{ITSExpressions}_{\mathcal{V}}$$

For all $e_1, e_2 \in \mathit{ITSExpressions}_{\mathcal{V}}$ **it holds true that**

$$\{(e_1 + e_2), (e_1 - e_2), (e_1 * e_2), (e_1/e_2), (e_1 ** e_2), (e_1\ \mathit{mod}\ e_2), (e_1\ \mathit{rem}\ e_2)\} \subseteq \mathit{ITSExpressions}_{\mathcal{V}}$$

An element of $\mathit{ITSExpressions}_{\mathcal{V}}$ is called an **expression**. We omit brackets whenever they are not necessary. The operators have their usual precedence.

Now let Σ be a signature as defined in Definition 3.1. We define the set $\mathit{ITSTerms}_{\Sigma,\mathcal{V}}$ as the smallest superset of Σ_0 that fulfills the following condition:

For all $n \in \mathbb{N}_0$ **and all** $f \in \Sigma_n$ **it holds true that**

$t_1, \ldots, t_n \in \mathit{ITSTerms}_{\Sigma,\mathcal{V}} \cup \mathit{ITSExpressions}_{\mathcal{V}} \cup \mathcal{V} \cup \mathbb{Z}$ **implies**

$$f(t_1, \ldots, t_n) \in \mathit{Terms}_{\Sigma,\mathcal{V}}$$

The elements of *ITSTerms* are called **terms**. We write $\mathcal{V}(t)$ for any term t to denote the variables contained in t.

Using this definition we proceed to define the set

$$\mathit{ITSConditions}_{\mathcal{V}} := \{e_1 \bowtie e_2 \mid e_1, e_2 \in \mathit{ITSExpressions}_{\mathcal{V}}, \bowtie \in \{=, \neq, <, >, \leq, \geq\}\}$$

the elements of which are called **conditions**. Similarly to terms, we denote the variables occurring in a condition c by $\mathcal{V}(c)$

We the finally define

$$\mathit{ITSRules}_{\Sigma,\mathcal{V}} :=$$
$$\{(l, r, C) \in \mathit{ITSTerms}^2_{\Sigma\setminus\mathcal{V},\mathcal{V}} \times 2^{\mathit{ITSConditions}_{\mathcal{V}}} \mid C \text{ is finite}, \forall c \in C.\ \mathcal{V}(c) \subseteq \mathcal{V}(l) \cup \mathcal{V}(r)\}$$

An element of $\mathit{ITSRules}_{\Sigma,\mathcal{V}}$ is called a **rule**. If $(l, r, \{c_1, \ldots, c_n\})$ is a rule, we also write $l \to r \mid c_1 \wedge \cdots \wedge c_n$. If C is empty, we write $l \to r \mid \mathit{true}$. ■

Using this set of rules, we can now define integer transition systems. An integer transition system is nothing but a finite set of rules. In contrast to Prolog, the order of the rules does not matter. Also in contrast to that language, we do not distinguish between arithmetic expressions and their result. For example, we use $+(1, 1)$ and 2 interchangeably even though they are, formally speaking, different terms.

Definition 5.13 (Integer transition system)**.** Let Σ be a signature, let $\mathcal{V}$ be an infinite, countable set of variables and let I be a subset of $\mathit{ITSRules}_{\Sigma,\mathcal{V}}$. If I is finite, we call I an **integer transition system** or *ITS*. ■

As described previously, a term can be evaluated using the rules of an *ITS*. In order to describe this formally, we use the notion of unification that we already used in Prolog. We defined this notion in Definition 3.7. One major difference between Prolog and integer transition systems is the fact that the intermediate terms occurring during an evaluation may not contain variables in an *ITS*. At any point during the computation, its current state can be described by a single ground term.

Definition 5.14 (Evaluation)**.** Let t and t' be ground terms and let I be an integer transition system. We say that I **evaluates** t **to** t' if there exists a rule $(l, r, c) \in I$ and a substitution $\delta : (\mathcal{V}(r) \setminus \mathcal{V}(l)) \to \mathit{ITSTerms}_{\Sigma\setminus\mathcal{V},\emptyset}$ such that the following three conditions hold:

$$\mathit{mgu}(t, l) = \sigma \neq \bot \qquad t' = r\sigma\delta \qquad \forall c \in C.\ c\sigma \text{ is a tautology}$$

We write $t \to_I t'$ in this case. If the integer transition system I is clear from the context, we omit it and simply write $t \to t'$.

We furthermore write $t \to^* t'$ if there exists a finite sequence of terms $\pi = t_1, \ldots, t_n$ such that $t_1 = t$, $t_n = t'$ and for all $i \in [1; n)$, the statement $t_i \to t_{i+1}$ holds true. If there exists no rule that can be used to evaluate t_n, we call π an **evaluation** of π. If $\pi = t_1, \ldots$ is an infinite sequence with $t_i \to_I t_{i+1}$ for all $i \geq 1$, we also call it an evaluation. ■

ITS are inherently indeterministic. This means that there exist terms that have multiple evaluations over a single program.

Example 5.22 (Indeterminism of *ITS*s)**.** Let I be the following *ITS*:

$$\begin{array}{lcll} f(X) & \to & g(X) & \mid \mathit{true} \\ f(X) & \to & h(X) & \mid \mathit{true} \\ h(X) & \to & h(X) & \mid \mathit{true} \end{array}$$

The term $f(0)$ has the two evaluations

$$\pi_1 = f(0) \to g(0) \qquad\qquad \pi_2 = f(0) \to h(0) \to h(0) \to \dots$$

▲

Indeterminism cannot only be introduced through multiple applicable rules, but also through free variables in the right-hand side of a rule. These free variables can be instantiated to any ground term using the substitution δ from Definition 5.14. We illustrate this in the following example.

Example 5.23 (Nondeterminism of *ITS* through free variables)**.** Consider the *ITS* I containing the single rule

$$f(X) \to f(Y) \mid Y > X$$

The term $f(1)$ has infinitely many nonterminating evaluations, for example

$$\begin{aligned} \pi_1 :=& f(1) \to f(2) \to f(3) \to \dots \\ \pi_2 :=& f(1) \to f(3) \to f(5) \to \dots \end{aligned}$$

▲

Using this notion of evaluation, we can now define a terminating term. The idea is that a term t is called terminating if every ground term that unifies with t only has finite evaluations.

Definition 5.15 (Terminating term)**.** Let t be a ground term and let I be an *ITS*. We say that t is **terminating** if it only has finite evaluations with respect to I.

Now let t be a term that may contain variables. We then say that t is **terminating** if for all substitutions σ the condition

$$t\sigma \textbf{ is ground implies that } t\sigma \textbf{ is terminating with respect to } I$$

holds true. ■

Example 5.24 (Terminating and nonterminating terms)**.** Consider the *ITS* consisting of only the single rule $f(X) \to f(X+1) \mid X > 0$. The term $f(0)$ only has a single evaluation which consists only of itself. The term $f(1)$ on the other hand has the infinite evaluation $f(1) \to f(2) \to \dots$. Hence, the term $f(0)$ is terminating, whereas $f(1)$ is not.

Similarly, the term $f(X)$ is nonterminating. Consider the substitution $\sigma : X \mapsto 1$. This substitution produces $f(X)\sigma = f(1)$, which is nonterminating. Thus, the term $f(X)$ is nonterminating. ▲

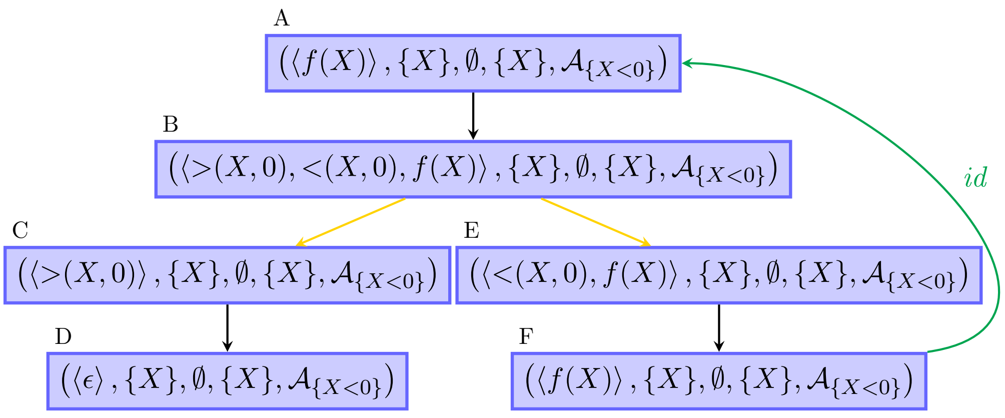

A
$(\langle f(X)\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
B
$(\langle >(X,0), <(X,0), f(X)\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
C
$(\langle >(X,0)\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
E
$(\langle <(X,0), f(X)\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
D
$(\langle \epsilon\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
F
$(\langle f(X)\rangle, \{X\}, \emptyset, \{X\}, \mathcal{A}_{\{X<0\}})$
id

Figure 5.13.: A termination graph of Program 5.8

5.2.2. Reduction of Termination Graphs to ITSs

In this section we define a construction of an *ITS* from a termination graph. Our goal is that the termination of the *ITS* implies termination of all runs described by the termination graph. Since the termination graph describes a superset of all the evaluations of the original program, this in turn implies the termination of the original program.

The general idea is that we construct one rule of the *ITS* for each edge in the termination graph. One of the easiest constructions to translate a graph into an *ITS* would just create a single function symbol f_s with arity 0 for each state s and create rules of the form $f_s \to f_{s'} \mid \mathit{true}$ for all states s and s' that are connected with an edge in the graph. Its start term would be $f_{s^q_{init}}$. Even though this translation is sound, we immediately see that it is not very powerful. In fact, this construction would fail to produce a terminating *ITS* as soon as the termination graph contains a loop.

We obviously need to be able to argue about the values of the variables in the states in order to model the behavior of Prolog more closely. Since all the terms occurring during a computation of an *ITS* must be ground, we may include the ground variables in the terms corresponding to states. We can then apply the substitutions that the edges are annotated with to the terms in order to handle assignments. This, however, does still not suffice to construct sufficiently powerful *ITS*s.

Program 5.8

```
f(X) :- X > 0, X < 0, f(X).
```

Example 5.25 (Termination in the presence of split edges)**.** Consider Program 5.8 and a simplified termination graph of this program on the query $f(X)$ in Figure 5.13. In this simplified example, we assume that we already know that X is some integer less than 0. Program 5.8 is obviously terminating, since there is no term that is both larger and smaller than 0. However, using the previously described naive construction, we receive the following simplified *ITS*:

$$\begin{array}{lcll} f_A(X) & \to & f_B(X) & \mid \mathit{true} \\ f_B(X) & \to & f_E(X) & \mid \mathit{true} \\ f_E(X) & \to & f_F(X) & \mid \mathit{true} \\ f_F(X) & \to & f_A(X) & \mid \mathit{true} \end{array}$$

in which the term $f_A(X)$ is obviously nonterminating, even though all evaluations of states represented by the abstract state A are terminating. ▲

The main problem with this construction is the handling of split nodes. Recall that the idea behind these nodes is that we only need to show that its right-hand successor is terminating under the assumption that the left-hand successor succeeded. Furthermore recall that our idea for reconstructing an evaluation from a split node was to descend into its left-hand successor first and only descend into the right-hand one if we return from the left-hand one. This was shown in Example 5.11 on page 81.

In order to model this behavior in an *ITS*, we use two function symbols f^s_{in} and f^s_{out} for each node s in the termination graph. We use the former to descend into the state and we use the latter to model the ascent after we have reached the end of the evaluation of the state. The arguments for these terms are those variables that are known to be ground in the given state.

Definition 5.16 (Encoding states as terms)**.** Let $s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. We assume that the variables contained in $\mathcal{G}$ are ordered.

We define two terms for s as follows:

$$\mathit{term}_{in}(s) := f^{in}_s(\mathcal{G}) \qquad\qquad \mathit{term}_{out}(s) := f^{out}_s(\mathcal{G})$$

■

This idea is inspired by the encoding of states as terms presented in [GSSK$^+$12, Definition 12]. In that work, the arguments of $\mathit{term}_{out}(s)$ contained not only the ground variables of s, but also those that were known to be ground after the evaluation of s. These variables were determined by the same groundness analysis that we used in the definition of the split rule in Definition 5.8. However, we have seen in the empirical evaluation of this method that the construction presented in this thesis actually suffices to show termination for large sets of programs.

Example 5.26. Consider the node A in the graph shown in Figure 5.12 on page 92. Since we do not know of any ground variables in the state A, we have $\mathit{term}_{in}(A) = f^{in}_A()$ and $\mathit{term}_{out}(A) = f^{out}_A()$.

In contrast to this, we know that the variables T_1, Y_1 and T_3 are ground in node Z. Hence, we get $\mathit{term}_{in}(Z) = f^{in}_Z(T_1, T_3, Y_1)$ and $\mathit{term}_{out}(Z) = f^{out}_Z(T_1, T_3, Y_1)$, if we order the ground variables lexicographically. ▲

Using these two terms, we can now use *ITS* rules to describe paths through the graph that correspond to the sequences of states that are visited during an execution of the program. We want to ensure that, for each abstract state s, the termination of $term_{in}(s)$ implies the termination of all concrete states described by s. Furthermore, we are going to use $term_{out}(s)$ to backtrack up the graph after the successful evaluation of a node.

There are several types of nodes that we have to consider when defining *ITS* rules. We mainly separate the nodes into categories based on the kind of edge that is attached to it. If the outgoing edge of a node is a simple edge, we call the node a simple node. We use similar nomenclature for all other kinds of edges that were discussed in Section 5.1.

The only two exceptions to this naming scheme are the abstract states of the form $s = (\langle\square\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ and those without outgoing edges. The former nodes symbolize a succeeding computation, so we call them **success nodes**. The latter nodes correspond to failed evaluations. Thus, we call them **failure nodes**.

Example 5.27 (Types of nodes)**.** Consider again the termination graph shown in Figure 5.12. We show the categories of all of its nodes in Table 5.1. ▲

Type of node	Nodes of the type
Success node	$\{N, R, V, Z\}$
Failure node	$\{F, I, K, M, P, S, W, X, Y, AA, AB, AC\}$
Simple node	$\{A, H, J\}$
Unification node	$\{C, G\}$
Comparison node	$\{E, L\}$
Arithmetic assignment node	$\{Q, U\}$
Instance node	$\{T\}$
Generalization node	$\emptyset$
Split node	$\{O\}$
Parallel node	$\{B, D\}$

Table 5.1.: Node types occurring in Figure 5.12

Using this idea, the simplest nodes to encode are success and failure nodes. The former nodes induce a "switch" from $term_{in}(s)$ to $term_{out}(s)$. The latter nodes do not induce any rules, since they symbolize failed evaluations.

Definition 5.17 (*ITS* rules for success nodes)**.** Let $s = (\langle\square\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state. We define the *ITS* rules corresponding to s as follows:

$$SuccessRules(s) := \{term_{in}(s) \to term_{out}(s) \mid true\}$$

■

Example 5.28. Consider the termination graph shown in Figure 5.12. This graph has the success nodes N, R, V and Z. Hence, we receive the following *ITS* rules from the encoding of these nodes.

$$f_N^{in}() \to f_N^{out}() \mid true \qquad f_R^{in}(T_1) \to f_R^{out}(T_1) \mid true$$
$$f_V^{in}(T_1, T_3) \to f_V^{out}(T_1, T_3) \mid true \qquad f_Z^{in}(T_1, Y_1, T_3) \to f_Z^{out}(T_1, Y_1, T_3) \mid true$$

▲

The definition of rules for failure nodes is quite simple as well. Since these nodes symbolize failing computations, we do not even need to switch from the incoming to the outgoing symbol. Instead, we just abort the computation at that point by not including any rules at all.

Definition 5.18 (*ITS* rules for failure nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be a failure node. We define the set of *ITS* rules corresponding to s as follows:

$$\mathit{FailureRules}(s) := \emptyset$$

■

It is quite easy to define rules for simple nodes. Recall that these edges to not have any conditions or substitutions as their annotations. These states are terminating if and only if their single successor state is terminating. Hence, it suffices to connect their in-terms and their out-terms with those of their successor.

Definition 5.19 (*ITS* rules for simple nodes)**.** Let $G = (V, E)$ be a termination graph and let $s = (S, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A}) \in V$ be a simple node with $S \neq \langle \square \rangle$. Furthermore, let s' be the single abstract successor state of s.

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned} \mathit{SimpleRules}(s) := \{&\mathit{term}_{\mathit{in}}(s) \to \mathit{term}_{\mathit{in}}(s') \mid \mathit{true}, \\ &\mathit{term}_{\mathit{out}}(s') \to \mathit{term}_{\mathit{out}}(s) \mid \mathit{true}\} \end{aligned}$$

■

Example 5.29 (*ITS* rules for simple nodes)**.** Consider the termination graph shown in Figure 5.12 again. This graph has the simple nodes A, H and J. The encoding of these nodes as *ITS* rules yields the following rules:

$$\begin{array}{ll} f_A^{\mathit{in}}() \to f_B^{\mathit{in}}() \mid \mathit{true} & f_B^{\mathit{out}}() \to f_A^{\mathit{out}}() \mid \mathit{true} \\ f_H^{\mathit{in}}() \to f_N^{\mathit{in}}() \mid \mathit{true} & f_N^{\mathit{out}}() \to f_H^{\mathit{out}}() \mid \mathit{true} \\ f_J^{\mathit{in}}(T_1) \to f_O^{\mathit{in}}(T_1) \mid \mathit{true} & f_O^{\mathit{out}}(T_1) \to f_J^{\mathit{out}}(T_1) \mid \mathit{true} \end{array}$$

▲

We handle unification nodes next. The outgoing edges of unification nodes denote successful or failing unifications. Recall that there are two possibilities to evaluate such states in the abstract semantics.

If we can infer that the unification is certainly failing, then we only produce a state modeling this. This was formalized in Abstract Evaluation Rule 4.4.

The other possibility was that we were not able to infer certain failure of the evaluation. In this case we produce two states, one for success of the unification and one for its failure. For this we used Abstract Evaluation Rule 4.3. The edge leading to the success state is labeled with the unification used. Using this information, we can easily encode such nodes as *ITS* rules.

Definition 5.20 (*ITS* rules for unification and evaluation nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be a unification node that has the outgoing edges (s, s'_{succ}, σ) and $(s, s'_{back}, \bot)$.

If s has both edges, we define the *ITS* rules corresponding to s as

$$\begin{aligned} \mathit{UnificationRules}(s) := \{&\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{succ})\sigma \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{succ})\sigma \to \mathit{term}_{out}(s) \mid \mathit{true},\\ &\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{back}) \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{back}) \to \mathit{term}_{out}(s) \mid \mathit{true}\} \end{aligned}$$

If s only has the outgoing edge $(s, s'_{back}, \bot)$, that is, if Abstract Evaluation Rule 4.4 was applied to it, its *UnificationRules* contain only the former two rules. ■

Example 5.30 (*ITS* rules for unification and evaluation nodes)**.** We now encode the unification nodes of the termination graph shown in Figure 5.12. The unification nodes of this graph are C and G. The construction of *ITS* rules for these nodes yields the following rules:

$$\begin{array}{ll} f^{in}_C() \to f^{in}_E() \mid \mathit{true} & f^{out}_E() \to f^{out}_C() \mid \mathit{true}\\ f^{in}_C() \to f^{in}_F() \mid \mathit{true} & f^{out}_F() \to f^{out}_C() \mid \mathit{true}\\ f^{in}_G() \to f^{in}_L() \mid \mathit{true} & f^{out}_L() \to f^{out}_G() \mid \mathit{true}\\ f^{in}_G() \to f^{in}_M() \mid \mathit{true} & f^{out}_M() \to f^{out}_G() \mid \mathit{true} \end{array}$$

▲

There are two more kinds of edges based on the abstract semantics for which we have to define *ITS* rules. They are comparison edges and arithmetic assignment edges. We handle the former one first.

For this definition we assume that s is a comparison node that has all three possible successors. The three possible cases for an arithmetic comparison were an error during the evaluation of the expressions that are to be compared, the success of the comparison, and its failure. The edges leading to these cases are labeled with the comparison that has to hold true in order to reach the respective case. Hence, we can use these comparisons as conditions on the rules that encode the transitions.

Definition 5.21 (*ITS* rules for comparison nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be a comparison node with the outgoing edges $(s, s'_{err}, \bot)$, $(s, s'_{succ}, t_1 \bowtie t_2)$ and $(s, s'_{fail}, t_1 \bowtie' t_2)$.

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned} \mathit{ArithmeticRules}(s) := &\{\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{err}) \mid \mathit{true},\\ &\quad \mathit{term}_{out}(s'_{err}) \to \mathit{term}_{out}(s) \mid \mathit{true}\}\\ \cup\ &\{\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{succ}) \mid t_1 \bowtie t_2,\\ &\quad \mathit{term}_{out}(s'_{succ}) \to \mathit{term}_{out}(s) \mid t_1 \bowtie t_2\}\\ \cup\ &\{\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{fail}) \mid t_1 \bowtie' t_2,\\ &\quad \mathit{term}_{out}(s'_{fail}) \to \mathit{term}_{out}(s) \mid t_1 \bowtie' t_2\} \end{aligned}$$

If either of the outgoing edges do not exist, the corresponding *ITS* rules are removed from *ArithmeticRules*(s). ■

Example 5.31 (*ITS* rules for comparison nodes)**.** Consider the termination graph shown in Figure 5.12 again. We now encode the comparison nodes E and L of this graph. Since both of these comparison nodes have all three possible successors, we receive the following twelve rules as the *ITS* encoding of these nodes:

$$\begin{array}{lll}
f^{in}_E() \to f^{in}_I() \mid \mathit{true} & f^{in}_E() \to f^{in}_J(T_1) \mid T_1 > 0 & f^{in}_E() \to f^{in}_K(T_1) \mid T_1 \leq 0 \\
f^{in}_I() \to f^{in}_E() \mid \mathit{true} & f^{in}_J(T_1) \to f^{in}_E() \mid T_1 > 0 & f^{in}_K(T_1) \to f^{in}_E() \mid T_1 \leq 0 \\
f^{in}_L() \to f^{in}_P() \mid \mathit{true} & f^{in}_L() \to f^{in}_Q(T_1) \mid T_1 = 0 & f^{in}_L() \to f^{in}_R(T_1) \mid T_1 \neq 0 \\
f^{in}_P() \to f^{in}_L() \mid \mathit{true} & f^{in}_Q(T_1) \to f^{in}_L() \mid T_1 = 0 & f^{in}_R(T_1) \to f^{in}_L() \mid T_1 \neq 0
\end{array}$$

▲

Finally, arithmetic assignment combines unification and arithmetic evaluation. Thus, we translate arithmetic assignments edges to *ITS* by combining the ideas for the encoding of unification edges and those for the encoding of comparison edges. We use the comparison that has to succeed for the evaluation to succeed as the condition of the rule. Furthermore, we apply the resulting substitution to the term of the succeeding state. We again assume that the arithmetic assignment node has all three possible successors.

Definition 5.22 (*ITS* rules for arithmetic assignment nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be an arithmetic assignment node with the outgoing edges $(s, s'_{err}, \bot, \bot)$, $(s, s'_{succ}, t_1 = t_2, \sigma)$ and $(s, s'_{fail}, \bot, \bot)$.

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned}
\mathit{ArithmeticAssignmentRules}(s) := \{ & \mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{err}) \mid \mathit{true}, \\
& \quad \mathit{term}_{out}(s'_{err}) \to \mathit{term}_{out}(s) \mid \mathit{true}\} \\
\cup \{ & \mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{succ})\sigma \mid t_1 = t_2, \\
& \quad \mathit{term}_{out}(s'_{succ})\sigma \to \mathit{term}_{out}(s) \mid t_1 = t_2\} \\
\cup \{ & \mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{fail}) \mid \mathit{true}, \\
& \quad \mathit{term}_{out}(s'_{fail}) \to \mathit{term}_{out}(s) \mid \mathit{true}\}
\end{aligned}$$

If either of the outgoing edges do not exist, the corresponding *ITS* rules are removed from *ArithmeticAssignmentRules*(s). ■

Example 5.32 (*ITS* rules for arithmetic assignment nodes)**.** We encode the arithmetic assignment nodes Q and U of the termination graph shown in Figure 5.12. The former node has no successor that models the error case, whereas the latter node has all three possible successor states. Thus, we receive the following twelve rules that encode these nodes in an *ITS*:

$$\begin{array}{ll}
f^{in}_Q(T_1) \to f^{in}_V(T_1,T_3) \mid T_3 = 1 & f^{out}_V(T_1,T_3) \to f^{out}_Q(T_1) \mid T_3 = 1 \\
f^{in}_Q(T_1) \to f^{in}_W(T_1) \mid \mathit{true} & f^{out}_W(T_1) \to f^{out}_Q(T_1) \mid \mathit{true} \\
f^{in}_U(T_1,Y_1) \to f^{in}_Y(T_1,Y_1) \mid \mathit{true} & f^{out}_Y(T_1,Y_1) \to f^{out}_U(T_1,Y_1) \mid \mathit{true} \\
f^{in}_U(T_1,Y_1) \to f^{in}_Z(T_1,T_3,Y_1) \mid T_3 = Y_1 \cdot T_2 & f^{out}_Z(T_1,T_3,Y_1) \to f^{out}_U(T_1,Y_1) \mid T_3 = Y_1 \cdot T_2 \\
f^{in}_U(T_1,Y_1) \to f^{in}_{AA}(T_1,Y_1) \mid \mathit{true} & f^{out}_{AA}(T_1,Y_1) \to f^{out}_U(T_1,Y_1) \mid \mathit{true}
\end{array}$$

The application of the substitution $\sigma : T_2 \mapsto T_3$ has no effect in these cases, since the variable T_2 is not known to be ground prior in Q and U and thus is not available as an argument in either term. ▲

We have now defined rules for each of the semantics-based edges that can occur in a termination graph. There are four more kinds of edges that can occur in such a graph, namely instance and generalization edges as well as split and parallelization edges. We first consider instance and parallelization nodes, which are converted to *ITS* rules in the same way.

The idea behind this encoding is that, if s is an instance of s', instead of showing that node s terminates, we can also instead show that s' is terminating. This is sound due to Lemma 5.1. We furthermore apply the instantiating substitution to s'.

Definition 5.23 (*ITS* rules for instance- and generalization nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be either an instance- or a generalization node. Let (s, s_{gen}, σ) be the outgoing instance- or generalization edge of s.

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned} \mathit{InstanceRules}(s) := \{&\mathit{term}_{in}(s) \to \mathit{term}_{in}(s_{gen})\sigma \mid \mathit{true}, \\ &\mathit{term}_{out}(s_{gen})\sigma \to \mathit{term}_{out}(s) \mid \mathit{true}\} \end{aligned}$$

■

Example 5.33 (*ITS* rules for instance- and generalization nodes)**.** The termination graph shown in Figure 5.12 only has the single instance node T and no generalization nodes. This node is encoded with the following *ITS* rules:

$$f^{in}_T(T_1) \to f^{in}_A() \mid \mathit{true} \qquad\qquad f^{out}_A() \to f^{out}_T(T_1) \mid \mathit{true}$$

▲

The one category of nodes for which we still have to define *ITS* rules are split- and parallel rules. The intuition behind the encoding of both nodes is quite simple. These nodes always have two successors. Any computation of a node represented by a split node can be emulated by first evaluating the left-hand side of the split node and the right-hand side afterwards.

We model this using the incoming and outgoing terms of the nodes. The idea is that we only need to enter the right-hand node if we have finished evaluation of the left-hand one. This mirrors our idea of showing termination of the right-hand node under the assumption that the left-hand node terminated.

Definition 5.24 (*ITS* rules for split nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be a split node with the split edges (s, s'_{lhs}) and (s, s'_{rhs}).

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned}\mathit{SplitRules}(s) := \{&\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{lhs}) \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{lhs}) \to \mathit{term}_{in}(s'_{rhs}) \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{rhs}) \to \mathit{term}_{out}(s) \mid \mathit{true}\}\end{aligned}$$

■

Example 5.34 (*ITS* rules for split nodes)**.** The termination graph shown in Figure 5.12 only has a single split node, namely the node O. This node is encoded with the following *ITS* rules:

$$\begin{aligned}&f_O^{in}(T_1) \to f_T^{in}(T_1) \mid \mathit{true}\\ &f_T^{out}(T_1) \to f_U^{in}(X_1, Y_1) \mid \mathit{true}\\ &f_U^{out}(T_1, Y_1) \to f_O^{out}(T_1) \mid \mathit{true}\end{aligned}$$

Note that in the rule $f_T^{out}(T_1) \to f_U^{in}(T_1, Y_1) \mid \mathit{true}$ the variable Y_1 is free. At this point the loss of precision by the split is modeled, since, at the application of this rule to some term, any term may be chosen for Y_1. ▲

We could apply the same idea to parallel nodes. However, experiments have shown that this construction makes the resulting *ITS* more complicated and makes the termination analysis harder than it needs to be. Also, while it is strictly necessary to "carry over" information from the run on the left-hand side to the node on the right-hand side at a split node, the evaluations of the children of a parallel node do not influence each other. Hence, an easier construction suffices. The intuition behind this construction is that, if an evaluation of some concretization of a parallel node is nonterminating, then either of its children is nonterminating. We can thus simply use nondeterminism in the *ITS* to pick one of the successor nodes.

Definition 5.25 (*ITS* rules for parallel nodes)**.** Let $G = (V, E)$ be a termination graph and let $s \in V$ be a parallel node. Let (s, s'_{lhs}) and (s, s'_{rhs}) be its outgoing parallel edges.

We then define the *ITS* rules corresponding to s as follows:

$$\begin{aligned}\mathit{ParallelRules}(s) := \{&\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{lhs}) \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{lhs}) \to \mathit{term}_{out}(s) \mid \mathit{true},\\ &\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_{rhs}) \mid \mathit{true},\\ &\mathit{term}_{out}(s'_{rhs}) \to \mathit{term}_{out}(s) \mid \mathit{true}\}\end{aligned}$$

■

Example 5.35 (*ITS* rules for parallel nodes)**.** There are two parallel nodes in the termination graph shown in Figure 5.12, namely the nodes B and D. These are encoded as *ITS* rules using the definition above as follows:

$$\begin{array}{ll}
f_B^{in}() \to f_C^{in}() \mid \mathit{true} & \qquad f_C^{out}() \to f_B^{out}() \mid \mathit{true} \\
f_B^{in}() \to f_D^{in}() \mid \mathit{true} & \qquad f_D^{out}() \to f_B^{out}() \mid \mathit{true} \\
f_D^{in}() \to f_G^{in}() \mid \mathit{true} & \qquad f_G^{out}() \to f_D^{out}() \mid \mathit{true} \\
f_D^{in}() \to f_H^{in}() \mid \mathit{true} & \qquad f_H^{out}() \to f_D^{out}() \mid \mathit{true}
\end{array}$$

▲

We have now defined *ITS* rules for all types of nodes that can occur in a termination graph. The construction of the *ITS* corresponding to a termination graph is now quite simple, as we simply encode each node individually.

Definition 5.26 (*ITS* corresponding to termination graphs)**.** Let $G = (V, E)$ be a Termination Graph. We define the *ITS* S **corresponding to** G as follows:

$$S = (\{\mathit{Rules}(v) \mid v \in V\},$$

where $\mathit{Rules}(v)$ is a function that performs a case distinction based on the node type of v and returns $\mathit{SuccessRules}(v)$, $\mathit{SimpleRules}(v)$, $\mathit{UnificationRules}(v)$, $\mathit{ArithmeticRules}(v)$, $\mathit{ArithmeticAssignmentRules}(v)$, $\mathit{InstanceRules}(v)$, $\mathit{SplitRules}(v)$, or $\mathit{ParallelRules}(v)$, depending on which rules are defined for v. ■

Using this definition, we can now construct the *ITS* corresponding to our previous example of a termination graph.

Example 5.36 (Constructing an *ITS* for Program 5.7)**.** The *ITS* corresponding to the termination graph shown in Figure 5.12 consists of all the rules given in Examples 5.28 through 5.35. ▲

We have seen how to construct an *ITS* from any given termination graph. In the next section we are going to show that this construction is actually sound.

5.2.3. Soundness of the Reduction

It remains to show that the construction presented in the previous section is sound with respect to termination. For this we first state that the translation from termination graphs to *ITS* is sound in Lemma 5.4. The full proof of this lemma can be found in Appendix A. We then use this lemma to show the soundness of the complete construction in Theorem 5.1.

Lemma 5.4. *Let G be a termination graph, let s be a state in G and let $\mathfrak{s}$ be a nonterminating concrete state that is represented by s with the concretizing substitution γ. Then there exists a successor s' of s in G that represents a concrete state $\mathfrak{s}'$ with the concretizing substitution γ', such that $\mathfrak{s}'$ is nonterminating. Furthermore, either $\mathit{term}_{in}(s)\gamma$ is nonterminating or $\mathit{term}_{in}(s)\gamma \to_I^+ \mathit{term}_{in}(s')\gamma'$ holds true.* ■

As stated previously, we now show that this local soundness of the translation of termination graphs to *ITS* implies the soundness of the complete termination analysis.

Theorem 5.1 (Soundness of termination analysis). *Let P be a program and let q be a query. Furthermore, let G be a termination graph constructed from P and q with Algorithm 5.1. Finally, let I be the ITS corresponding to G. If the query q is nonterminating on P, then $term_{in}(s^q_{init})$ is nonterminating on I.* ■

Proof. Due to the definition of nonterminating queries and due to the construction of termination graphs, we know that the root of G is an abstract state s^q_{init} that represents a nonterminating concrete state $\mathfrak{s}$ with the concretizing substitution γ.

Due to Lemma 5.4 we know that there exists a state s' in G such that s' is a successor of s^q_{init}, and such that the abstract state s' represents a nonterminating concrete state $\mathfrak{s}'$ with the concretizing substitution γ'. The Lemma furthermore states that $term_{in}(s^q_{init})\gamma$ is either nonterminating or it evaluates to $term_{in}(s')\gamma'$. If it is nonterminating, the proof is finished.

Otherwise we can apply Lemma 5.4 again to the abstract state s' and the nonterminating concrete state $\mathfrak{s}'$ represented by it and receive another abstract state s'' that is a successor of s' such that s'' is nonterminating. In addition to this, either $term_{in}(s')\gamma'$ is nonterminating or $term_{in}(s')\gamma' \to_I^+ term_{in}(s'')\gamma''$ holds true, where γ'' is the substitution that concretizes s'' to some nonterminating concrete state $\mathfrak{s}''$. In the former case, the proof is finished, since $term_{in}(s^q_{init})\gamma$ evaluates to the nonterminating $term_{in}(s')\gamma'$, whence $term_{in}(s^q_{init})\gamma$ itself is nonterminating.

Since both $term_{in}(s^q_{init})\gamma \to_I^+ term_{in}(s')\gamma'$ and $term_{in}(s')\gamma' \to_I^+ term_{in}(s'')\gamma''$ hold true, $term_{in}(s^q_{init})\gamma \to_I^+ term_{in}(s'')\gamma''$ holds true as well. We can continue this construction by repeated application of Lemma 5.4 and receive either some nonterminating term that $term_{in}(s^q_{init})\gamma$ evaluates to, or an infinite chain of terms that $term_{in}(s^q_{init})\gamma$ evaluates to. In both cases, $term_{in}(s^q_{init})\gamma$ is nonterminating on I. □

We have thus shown the soundness of the termination analysis presented in this chapter. It remains to discuss the termination analysis of *ITS*. We do so in the next section and give pointers to literature on this topic.

5.2.4. Termination Analysis of ITSs

The formalism of Integer Transition Systems is well known and has been studied in depth. Even though the halting problem for these systems is undecidable, there are algorithms that decide termination for large classes of systems. [PR04] describes one such method.

For our implementation we use the algorithms implemented in the termination prover AProVE as a backend for the resulting *ITS*. These algorithms are sufficient for handling the integer transition systems resulting from our construction, especially since they are comparatively simple. The *ITS*s resulting from our construction usually do not pose a challenge to most termination provers for this formalism.

Example 5.37 (Termination analysis of *ITS* corresponding to Program 5.7)**.** Consider the *ITS* for Program 5.7 that we constructed in Example 5.36. The termination prover that we use for showing termination of the term $f_A^{in}()$ first simplifies that *ITS* to the *ITS* consisting of the single rule:

$$f(X) \to f(X-1) \mid X > 0$$

and proceeds to show termination of the term $f(X)$, which is very simple. ▲

Note that the simplified integer transition system in the previous example is precisely the argument that a human prover would use to argue about the termination of the program. It could be described in plain English as "we subtract 1 from X as long as X is larger than 0". The fact that we can only perform this subtraction finitely often for any fixed X is exactly the reason why Program 5.7 is terminating.

5.3. Practical Implementation

While we have described the termination analysis formally in the two previous sections, there are still some roadblocks to an actual implementation. In this section, we briefly describe the most major of these hindrances and how we handled them in our practical implementation.

The most major obstacle was the treatment of arithmetic states. In our formalization, these states are an infinite set of functions. Since it is not possible to directly represent such an infinite set in a programming language, we instead store the relations R that define the arithmetic state $\mathcal{A}_R$. Using this representation, the check whether or not the statement $\mathcal{A}_R \models r$ holds true is reduced to checking whether or not $\bigwedge R \Rightarrow r$ holds true. This representation also allows us to check whether or not the abstract state s is an instance of the abstract state s' by checking whether or not the statement $\bigwedge R \Rightarrow \bigwedge R'$ holds true. Both checks can easily be carried out using an SMT solver.

Recent work furthermore allowed us to modularize the representation of the arithmetic state, so that we can easily switch between the representation as a set of relations and a representation using other abstract arithmetic domains. It would, for example, be possible to represent arithmetic states using the well-known interval domain or the octagon domain from [Min01]. These arithmetic domains both have their own decision procedures for checking the aforementioned statements.

Another hurdle for the practical implementation is the extension of the functions PickEdges and GeneralizationCandidate from [Str10, Section 6.2] to take the arithmetic state into account. The standard heuristic for PickEdges decides whether to use an abstract evaluation rule to evaluate a state, or whether to use an instance-, generalization-, split- or parallel edge. We simply extended that heuristic to pick an arithmetic evaluation edge or arithmetic assignment edge if it chooses to use an abstract evaluation rule and such an edge is defined for the current state. We just extended the abstract semantics by additional inference rules, but we did not change the rules that govern when to pick edges that do not correspond to abstract evaluation. Since the choice of these rules is responsible for nontermination of the graph construction, he termination argument from [Str10, Section 6.3] still holds true for this adapted heuristic.

We also had to change the implementation of GENERALIZATIONCANDIDATE slightly, as this function did not take the arithmetic state into account. Recall that GENERALIZATIONCANDIDATE takes two abstract states s and s' as input and tries to construct a state s_{gen} such that both s and s' are instances of s_{gen}. Since we represent arithmetic states by sets of relations, we can simply intersect both sets of relations to receive a new arithmetic state that is more general than the two existing arithmetic states. We employ heuristics to perform a kind of relaxed intersection.

Example 5.38 (Relaxed intersection of sets of relations)**.** Consider the two sets of relations $R_1 := \{X > 1\}$ and $R_2 := \{X > 2\}$. The traditional intersection yields $R_1 \cap R_2 = \emptyset$.

In the context of arithmetic states, however, we see that the $\mathcal{A}_{\{X>1\}}$ is a strict superset of $\mathcal{A}_{\{X>2\}}$. Hence, the relaxed intersection of R_1 and R_2 yields the set $\{X > 1\}$. ▲

This relaxation of the intersection increases the power of our analysis, since we do not "lose" relations unnecessarily during this process of generalizations. Using these ideas, which mainly build on the implementation of the termination analysis from [Str10], we were able to implement this termination analysis in the termination prover APROVE. We evaluate this implementation in the following chapter.

6. Evaluation

In the previous chapter, we have defined a technique for termination analysis of programs written in our fragment of PROLOG. We have also shown its correctness.

In addition to this theoretical proof of soundness, we have implemented this method and performed a number of experiments using examples from a multitude of different sources. We present the results of these experiments and evaluate our approach in comparison to other approaches to program termination in this chapter.

We first present the approaches we compared to our approach in Section 6.1 and present the types of examples we performed our experiments on in Section 6.2. In Section 6.3 we present a summary of our experiments. A combination of the existing approaches is evaluated in Section 6.4. We discuss the advantages and limitations of our technique in Section 6.5.

6.1. Compared Approaches

There are multiple approaches to termination analysis of PROLOG-programs or fragments thereof. Some of these approaches are implemented in APROVE as well, while others are implemented in separate tools. We first give an overview over the techniques that are implemented in APROVE, before we discuss the stand-alone tools that we were able to obtain. A concise overview over all approaches mentioned in this section can be found in Table 6.1.

Tool name	**Fragment**	**Analysis**	**Compared**	**Reason**
APROVE (to PiTRS)	Cut + Arithmetic	Termination	Yes	
APROVE (to DT)	Cut	Termination	Yes	
APROVE (to TRS)	Cut	Termination	Yes	
CIAO	Cut + Arithmetic	Termination	No	No analysis
POLYTOOL	Logic	Termination	No	Limited Fragment
HASTA-LA-VISTA	Arithmetic	Termination	No	Non-runnable code
TERMINWEB	Logic	Termination	No	Non-runnable code
pTNT	Cut	Nontermination	Yes	
TALP	Logic	Termination	No	Limited Fragment
cTI	Logic	Termination	No	Limited Fragment
CASLOG	Arithmetic	Complexity	No	Explicit moding
NTI	Logic	Nontermination	No	Limited Fragment
LPTP	Logic	Theorems	No	Not automatic

Table 6.1.: Approaches to termination analysis

6.1.1. Strategies Implemented in AProVE

There are three different strategies that are used by AProVE for showing the termination of Prolog programs. We give a brief summary of each approach and point to the relevant literature for further reading.

Conversion to Term Rewrite System with Argument Filtering (to PiTRS) One approach to showing termination of logic programs is to transform them into a term rewrite system whose termination implies termination of the original program. Such a transformation is described in [SKGST09].

In addition to this conversion, this method also includes a rewriting of arithmetic programs to pure logic programs. It does so by using Peano notation to encode natural numbers as uninterpreted constants. It furthermore adds definitions of the predefined arithmetic operators and comparisons in terms of Peano notation, thus transforming an arithmetic program into a logic program. This transformation was implemented in AProVE. It is one of the techniques we will compare our approach to.

Conversion to Dependency Triples (to DT) In order to combine direct and transformational approaches to the termination analysis of programs, the dependency triple framework was introduced in [SKGN10] and refined in [SSKG11]. This approach uses the same construction of an abstract program graph as we do in this work. Even though this construction does not support arithmetic, we compare our method to it, since it is, to the best of our knowledge, the most powerful technique for showing termination of Prolog programs with cut.

Conversion to TRS (to TRS) Another approach was presented in [GSSK$^+$12]. This method first constructs an abstract program graph that is similar to our construction and which was the starting point for our construction. It then transforms the graph into a term rewrite system, whose termination implies termination of the original program. While this construction takes cuts into account during the construction of the graph, it simply ignores arithmetic comparisons as well as the is-predicate. Nevertheless, we compare it to our method.

6.1.2. Other Tools for Termination

There are a number of other tools that also prove termination of logic programs, with varying degrees of handling for built-in predicates of Prolog. In this section we give an overview over these tools and discuss whether or not we compare them to the approach presented in this thesis.

Ciao This is a complete Prolog implementation, which includes proprietary extensions. Among these extensions is a predicate `terminates\1` which takes a term and succeeds if all calls of the form described by the term terminate. According to [BCC$^+$05, Section 56], this predicate uses the non-termination analysis presented in [DLGH97].

However, experiments on our set of benchmarks have shown that the implementation apparently only tries to evaluate the predicate and does not perform any more complicated analysis of it. In particular, calls of non-terminating predicates lead to a nonterminating analysis. Hence, we exclude CIAO from the comparison with our approach.

PolyTool This is another approach to show termination of definite logic programs. It uses a conversion of programs to polynomials, where each function symbol and each predicate is associated with a fixed polynomial. The system is described in [NDS07]. An explanation of the theoretical background is given in [NDS05]. Since the tool only supports definite logic programs, but neither programs containing cuts nor any other built-in predicates of PROLOG, we exclude it from the comparison with our approach.

Hasta-La-Vista HASTA-LA-VISTA uses a transformation of a program into a set of constraints to show its termination, as described in [SDS03]. Even though we were able to obtain a copy of this tool, we were not able to execute the tool on a modern computer. The tool was originally written using a PROLOG implementation manufactured by a Belgian company called BIM. Since the company went bankrupt in 1996, we were unable to obtain a copy of this implementation. As HASTA-LA-VISTA makes heavy use of the proprietary extensions of PROLOG by BIM, it is not possible to run this tool on modern computers. We therefore exclude HASTA-LA-VISTA from the comparison with our approach.

TerminWeb TERMINWEB implements semantics for PROLOG that allow reasoning about the termination behavior of the program, which were originally published in [GC03, CT99]. Similar to HASTA-LA-VISTA, we were able to obtain a copy of the tool. However, the tool was written using an outdated version of SICStus PROLOG, for which neither the implementation, nor the documentation is available anymore. Thus, it was not possible to execute the tool on modern computers. For this reason, we exclude TERMINWEB from the comparison with our approach.

pTNT This tool infers non-termination in two phases [VDS11]. In the first phase, non-terminating queries are identified, while it is assumed that all arithmetic comparisons succeed. These comparisons are then transformed to a constraint satisfaction problem and solved in the second phase. We were able to acquire the code for this tool and could run it on a modern machine. Hence, we are going to compare PTNT with our approach.

TALP Another approach to termination analysis was described in [OCM00] and implemented in the tool TALP. In this approach, the program is first transformed into a term rewrite system with conditions, which is then in turn translated to a standard term rewrite system. Similar to our approach, the termination of the standard TRS implies the termination of the original program. However, this transformation ignores cuts and does not take arithmetic predicates into account. It is therefore not comparable to our approach and not included in the benchmarks.

cTI The approach of cTI is to perform a bottom-up analysis of the program in order to infer termination conditions for predicates [MB05]. However, this tool does not support cuts or integer arithmetic. We therefore exclude it from the comparison with our approach.

Caslog A complexity analysis of the evaluation of predicates can be carried out using Caslog [DL93]. Even though complexity analysis is not the same as termination analysis, the existence of a finite upper bound on the runtime of the inference of a predicate implies the termination of its inference. Thus, we considered including Caslog for comparison with our approach. However, Caslog needs explicit annotations about the so-called mode of each predicate that may occur during the evaluation of a given predicate, which is not included in our set of benchmarks. We thus exclude Caslog from the comparison with our approach.

NTI Another idea for showing nontermination of programs is to compute a set of looping queries. An iterative algorithm for this is presented in [PM06] and implemented in NTI. However, this tool only works on definite logic programs and ignores cuts and arithmetic comparisons. Therefore, we exclude NTI from the comparison with our approach as well.

LPTP Finally, there exists a tool called LPTP that implements a theorem prover over the domain of programs [Stä98]. However, since the verification offered by LPTP is interactive instead of automatic, this tool is not comparable to automatic (non-)termination provers. Therefore, we exclude LPTP from the set of compared tools.

6.2. Types of Examples

In this section we give a short overview over the examples that we used to evaluate our approach. The examples can be separated into two categories: logic and arithmetic examples. Logic examples are those whose (non-)termination-argument does not rely on arithmetic, but only on pure logic and the cut. These examples are taken from the Termination Problems Database[1], or TPDB for short [tpd15]. Whereas the TPDB separates these examples into three subcategories, based on what features of Prolog they use, we just use this set of 477 examples without further distinction.

Arithmetic examples are those examples whose argument for (non-)termination depends on the arithmetic properties. These examples are taken from a multitude of sources. We have taken 14 examples from [SS86, Chapter 8]. 12 more examples are taken from the solutions to the exercises to this chapter, provided by [Bar15]. Another 26 examples were taken from [Het15], as well as nine additional problems from [Lam15].

Furthermore, we have implemented five solutions to problems from [Hug15]. We implemented 28 examples to check our work for correctness during the implementation.

[1]`http://termination-portal.org/wiki/TPDB`

Finally, the largest part of our examples consists of ports of examples from [tpd15], which were originally written in the C-language. We ported the 68 examples from the category `AProVE_numeric` to PROLOG and used them in our set of examples. Thus, we have a set of 162 examples whose (non-)termination argument relies on arithmetic comparison and evaluation.

Of these examples, 156 have been added to the TPDB. The six examples which were not added consisted of very simple nonterminating examples that served merely to test the correctness of the implementation.

6.3. Discussion of Results

In this section we summarize the results of our experiments and discuss the comparison between our approach and existing ones. We provide and discuss a summary of the actual results of these runs. The major statistical properties of the results are shown in Figure 6.1 and Figure 6.2.

6.3.1. Performance on Logic Examples with Cut

We show the results of running the four compared approaches on the 477 logic examples in Figure 6.3. The runtime of these approaches is shown in Figure 6.4. Note that the y-axis is scaled logarithmically in the latter figure.

We first notice that none of the approaches implemented in APROVE are able to show nontermination of programs. This stands in contrast to PTNT, which can only show nontermination of programs, but not their termination. Out of the four methods for showing termination, the transformation to dependency triples clearly outperforms the other methods. The transformation to a PiTRS is the weakest previously existing approach. Our approach is slightly weaker than this transformation, but not by a large margin. More specifically, whereas transformation to a PiTRS could show the termination of 269 examples, our approach could still prove termination of 223 examples. Moreover, the examples for which we can show termination are not a subset of those that can be shown to be terminating by the transformation to PiTRS.

Figure 6.4 shows that even though our approach is slightly less powerful than existing approaches, it offers an improvement in average runtime over at least the two next

	Term Rewriting with Argument Filter	Dependency Triples	Term Rewriting	Integer Transition System	Two-Step
Yes	269	345	322	223	0
Maybe	208	132	155	254	421
No	0	0	0	0	56
Average [s]	10.37	3.70	7.29	6.19	7.29
Median [s]	2.14	1.97	2.15	3.05	0.46

Figure 6.1.: Statistical evaluations of benchmarks on 477 logic programs

powerful methods. The average runtime of our approach on the 477 examples in this collection is 6.19s, compared to 10.37s for the conversion to dependency triples and compared to an average runtime of 7.29s for the conversion to a term rewrite system with argument filtering. Hence we see that our approach is neither as powerful nor as fast as other methods for showing termination of logic programs with cut. This is especially visible when comparing median runtimes. The median runtime of our approach is 3.05s, which translates to a slowdown of 40% in comparison to the transformation to term rewriting, which is more powerful by far.

The evaluation of the five approaches on logic programs has shown that our approach is both weaker and slower than the existing approaches for termination analysis. However, even though our method is weaker than the existing ones, it can still show termination of more than half of the benchmarks that we know to be terminating.

6.3.2. Performance on Arithmetic Examples

The benefits of our approach are clearly visible when we use examples whose termination argument relies on arithmetic properties. We show the results of running our method on a set of 162 arithmetic programs. The termination results are shown in Figure 6.5, whereas the runtime of the benchmarks can be seen in Figure 6.6.

It is easy to see that our approach beats the existing approaches by a wide margin. This was to be expected, since none of the existing approaches take arithmetic evaluations and comparisons into account natively. These comparisons, however, are an integral part of the termination argument for the examples in this set.

The only method that handles these comparisons and evaluations at all is the conversion to a term rewrite systems with argument filtering. It does so by converting these comparisons to standard logic predicates using Peano notation. Thus, it is able to show termination for 67 out of the 162 examples in this set. Our approach does not rely on such a transformation and can show termination of 110 examples out of this set.

The positive answers of the other two approaches occur for some very simple examples, for which it apparently suffices to argue over the structure of the logical fragment of the program in order to show termination. This also holds for the five cases in which pTNT was able to show nontermination in this case.

An additional drawback of the conversion of arithmetic predicates to Peano notation

	Term Rewriting with Argument Filter	Dependency Triples	Term Rewriting	Integer Transition System	Two-Step
Yes	67	30	11	110	0
Maybe	85	132	151	52	157
No	0	0	0	0	5
Average [s]	54.66	15.12	13.09	13.93	0.63
Median [s]	71.26	2.23	1.73	1.85	0.66

Figure 6.2.: Statistical evaluations of our benchmarks on 162 arithmetic programs

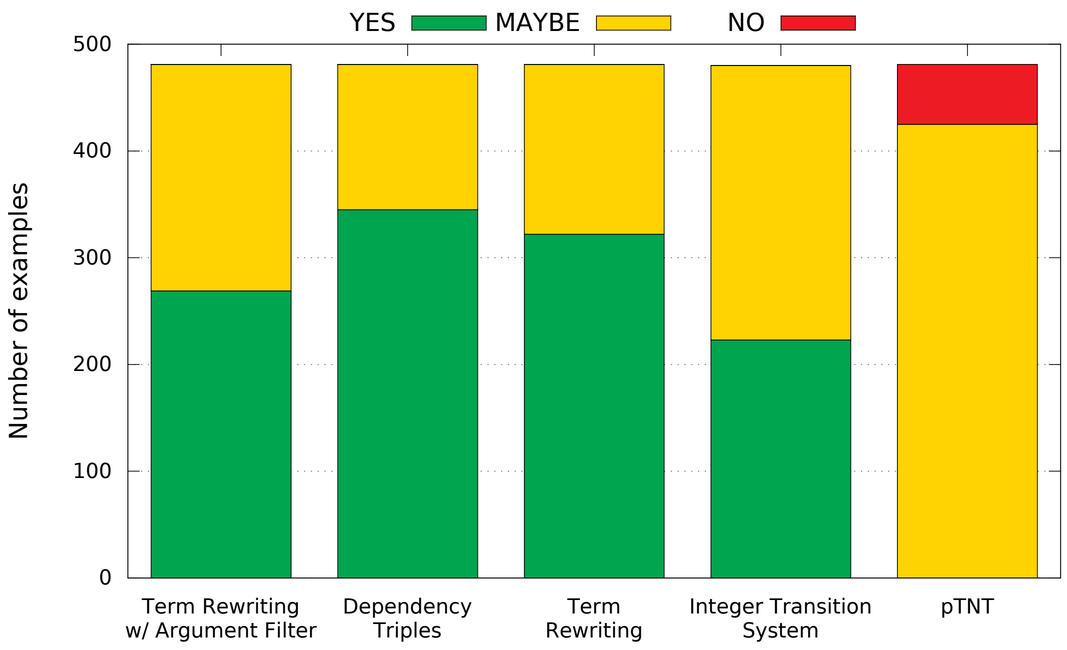


Figure 6.3.: Results of termination analysis on logic benchmarks

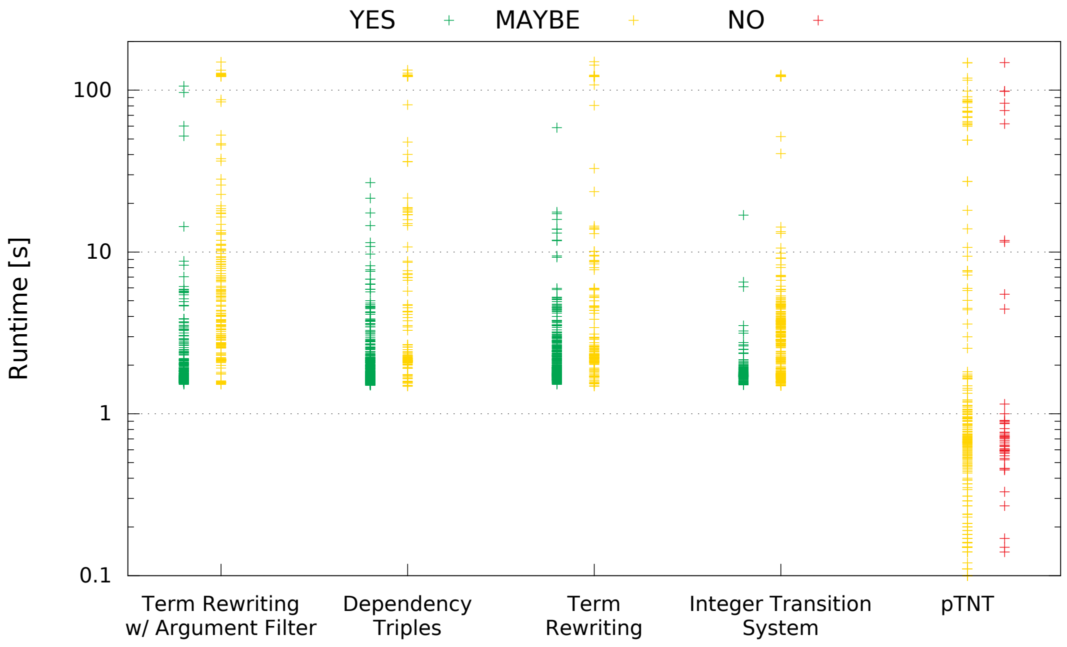


Figure 6.4.: Runtimes of termination analysis on logic benchmarks

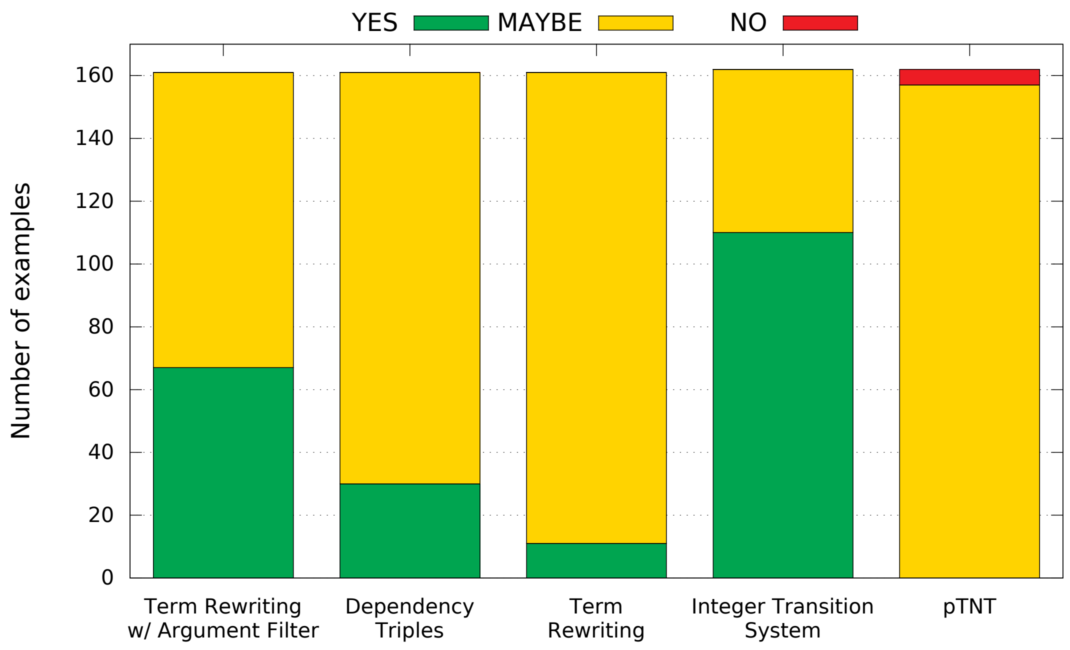


Figure 6.5.: Results of termination analysis on arithmetic benchmarks

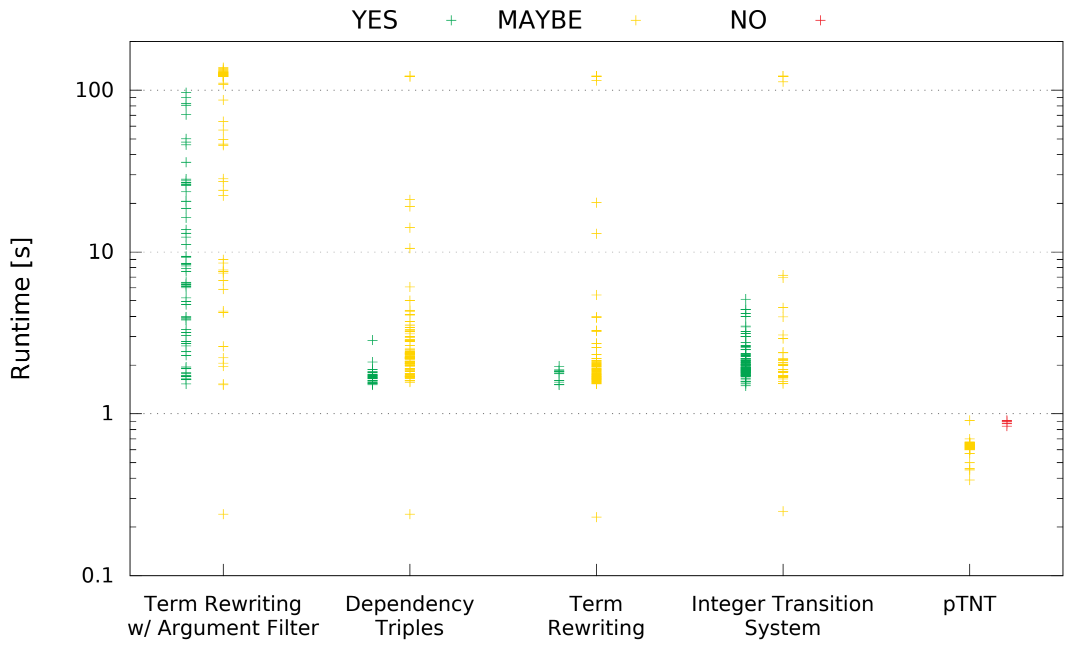


Figure 6.6.: Runtimes of termination analysis on arithmetic benchmarks

becomes obvious in Figure 6.6. Since this conversion results in a huge blowup of the program size, the runtime of the conversion to PiTRS are far greater than those on logic examples. This also shows itself in the average and median runtime of 58.26s and 27.40s, respectively, for this approach. When compared to the average and median runtime of 13.93s and 1.84s of our approach, respectively, it is clear that our approach does not only have a much greater power than even the most powerful existing approach, but also provides a clear speedup.

6.4. Combining the Approaches

None of the approaches implemented in AProVE handle a subset of the examples handled by any other approach. Thus, we also tested a version of AProVE in which all previously existing approaches and the new one presented in this thesis were executed in parallel.

Using this version, we could show termination of 353 logic benchmarks with an average and median runtime of 8.61s and 2.27s, respectively. This shows that this combined approach is more powerful than each individual approach on its own, even though the parallel execution of these approaches comes at a slight runtime penalty.

With regards to arithmetic examples, the combined version could show termination of 118 benchmarks with an average and median runtime of 16.39s and 2.36s, respectively. This shows that even though our new approach is more powerful than all existing approaches on arithmetic programs, not all other approaches handle a subset of those examples handled by our approach. Combining the existing approaches allows us to show more examples terminating than just running our approach in isolation. Similar to the logic benchmarks, this increase in power comes with a slight runtime penalty.

6.5. Advantages and Limitations

We have shown that our approach is both more powerful and faster than all existing and still available approaches to showing termination of arithmetic programs by far. The conversion into an integer transition system is both fast and retains the properties of the program which are necessary for showing termination. Also, the decision procedure for termination of integer transition systems, which we use as a backend, is powerful enough to show termination of the resulting systems.

However, our method does not perform as well on logic programs. For these programs there exist more powerful and faster decision procedures that partially decide termination of programs. Thus, we clearly see that our encoding does not carry over the logical properties of programs quite as well as, for example, the encoding into dependency triples does.

This shows that a combined approach would probably work best for the analysis of general programs. Such an approach would run our method if the program contains arithmetic predicates at all. Otherwise it would construct dependency triples from the program and show termination that way.

7. Conclusion

We have started this thesis with the introduction of a fragment of the programming language Prolog that contains both the built-in arithmetic capabilities of that language as well as the cut in Chapter 2. In that chapter we have given an intuitive explanation of the behavior of a program. For the official semantics of the language, we have referred to [ISO95] and have given a brief overview over the features that ISO-Prolog contains, but that are not present in our fragment.

In Chapter 3 we have recapitulated the concrete semantics first published in [SESK$^+$12]. This semantics is an alternative semantics for ISO-Prolog, which is equivalent to the "official" semantics both in regards to termination and with regards to complexity of evaluations. We have presented those parts of the concrete state-based semantics of Prolog from [SESK$^+$12] that are relevant to our fragment of Prolog.

Following that we have defined an abstract semantics for this fragment of Prolog in Chapter 4. Parts of this semantics were first published in [SESK$^+$12] as part of a termination analysis for the logic fragment of Prolog with the cut. We have separated those rules from the termination analysis that are general purpose and have extended this set by rules that handle arithmetic comparison and evaluation. In this chapter we have also showed that these extended abstract semantics provide a sound abstraction of the concrete semantics.

In Chapter 5 we have built a termination analysis for our fragment of Prolog. For this we have first constructed a termination graph from a program and a query, which we have then transformed into an integer transition system. We have also showed that this reduction of a program and a query to an *ITS* is sound with respect to termination, meaning that termination of the resulting *ITS* implies termination of the original program. The resulting *ITS* can then be analyzed for termination using existing approaches.

Finally, we have reported on our implementation of the termination analysis from the preceding chapter in Chapter 6. We have also compared it to existing methods for showing program termination on two sets of examples, namely a set of examples that only used logic features and the cut, and a set of examples that used the full fragment of Prolog defined in an earlier chapter.

We found that our approach is slightly less powerful than existing approaches on the former set, but is competitive to those methods in terms of runtime. It also is still powerful in terms of absolute numbers, showing termination for more than half of the terminating examples. However, it outperformed all existing methods for showing termination of Prolog programs on the latter set both in terms of runtime and in terms of power.

This is mainly due to the fact that existing approaches either only apply a transformation to Peano notation to arithmetic programs or simply treat arithmetic predicates as built-in ones without any result on the program state. While the former method results in a strong increase of the size of the program, the latter method cannot show termination of programs whose termination argument relies on arithmetic properties.

Our main theoretical contributions consist of the separation of an abstract semantics from the termination analysis performed in [SESK$^+$12] and the extension of that termination analysis to the fragment of Prolog considered in this work. On a practical side of things, we have implemented the aforementioned termination analysis and shown that this method is quite powerful to show termination of programs written in our fragment of Prolog. To the best of our knowledge, this is the first termination analysis for this fragment of the language. However, there remain some open research questions. We are going to discuss these in the next section.

7.1. Future Work

Even though this work is a significant improvement over existing methods for automated analysis of Prolog programs, there are still open questions for further research. We discuss some of these questions in this section. Some of these questions were already posed in [Str10, Chapter 8].

First and foremost, even though we extended the fragment of Prolog that our analysis handles beyond that handled by other existing approaches, there are still a lot of features of ISO-Prolog that are ignored by our analysis. These include, for example, user interaction, file input and output as well as predicates that change the program at runtime. A more thorough listing of the built-in predicates that are missing from our fragment is given in Section 2.3. One avenue of research would be the inclusion of these predicates in the abstract semantics defined in Section 4. It would then be possible to extend the termination analysis of Section 5 to programs containing these predicates.

The termination analysis rests in large parts on the choice of the function PickEdges for Algorithm 5.1. In our implementation we used a function that proved to work well on our set of examples, but that does not have a solid theoretical foundation. We are nearly certain that there exist other choices of this function that improve the termination analysis. Finding such a function would improve both the power and the runtime of the termination analysis.

Another interesting path would be the use of the abstract semantics in other types of analysis. The automated analysis of the complexity of the evaluation of a query as well as its determinacy are analyses that come to mind. Both analyses have been discussed in [GSSK$^+$12]. This work implemented these analyses using the same graph construction that it used for termination analysis, and which we in turn used in this work. Hence, it should be easy to use the graph constructed in Section 5.1 for these kinds of analyses.

We have seen in the empirical evaluation of our method that even though our approach performs well on arithmetic programs, it is outperformed by other, previously existing methods on logic programs with cut. It would be interesting to find a way to combine the power of our method on arithmetic programs with the power of existing approaches on logic programs with cut.

Finally, both our abstract semantics and the termination analysis build on the notion of abstract states as defined in Section 4.1. Although the information contained in these states is sufficient to show termination of our benchmarks, we are unable to construct a complete abstraction using these states, but only a sound one. For example, we only store information about non-unifiability of terms in the state, but no information about the inequality of terms. We are certain that the storage of additional information in these states would sharpen the analysis.

Even though there are methods for showing termination for complete other languages and not just fragments thereof, analysis of Prolog has always concentrated on fragments of this language. This is in spite of the fact that the semantics of Prolog are more precisely defined and simpler than those of most other languages, while it is still possible to write programs with a real-world use in it with considerable ease. It is our hope that this work builds the foundation for a method of analysis of the complete Prolog language.

A. Supplementary Proofs

Lemma A.1 (Reprint of Lemma 5.3 on page 85)**.** *Let $s = (\langle t, T\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ be an abstract state, let NextG be a sound groundness analysis function and let σ be a function that replaces all variables in $\langle t, T\rangle$ with fresh term variables. Furthermore, let γ be some concretization of s. If $\langle t\gamma, T\gamma\rangle$ evaluates to $\langle T\gamma\gamma'\rangle$, then $\sigma^{-1}\gamma\gamma'$ conforms to $s' = (\langle T\sigma\rangle, \mathcal{G}\sigma \cup \mathit{NextG}(t, \mathcal{G})\sigma, \mathcal{U}\sigma, \mathcal{E}\sigma, \mathcal{A}\sigma)$.* ■

Proof. It is clear that $\gamma\gamma'$ conforms to the knowledge base $(\mathcal{G}\sigma, \mathcal{U}\sigma, \mathcal{E}\sigma, \mathcal{A}\sigma)$, since $\sigma\sigma^{-1} = id$ and since γ already is a concretization, whence $\mathcal{T}(T\sigma\sigma^{-1}\gamma) = \emptyset$. Since γ' is another concretization, i.e., the domain of γ contains only term variables, we have $T\sigma\sigma^{-1}\gamma\gamma' = T\gamma$. As γ conforms to s, we can thus infer that $\sigma^{-1}\gamma\gamma'$ fulfills the criteria for conformity regarding $\mathcal{U}\sigma$, $\mathcal{E}\sigma$ and $\mathcal{A}\sigma$.

It remains to show that $\mathcal{V}(X\sigma\sigma^{-1}\gamma\gamma')$ is empty for all $X \in \mathit{NextG}(t, \mathcal{G})$, which amounts to showing that $\mathcal{V}(X\gamma\gamma')$ is empty for these X. This follows directly from the assumed soundness of *NextG*. Since γ' is the answer substitution found by evaluating $t\gamma$, we know by this soundness criterion that γ' maps all variables in $\mathit{NextG}(t, \mathcal{G})$ to ground terms. Hence, the statement $\mathcal{V}(X\sigma\sigma^{-1}\gamma\gamma') = \emptyset$ holds true for all $X \in \mathit{NextG}(t, \mathcal{G})$.

Thus, $\sigma^{-1}\gamma\gamma'$ conforms to s' and Lemma A.1 holds true. □

In order to show Lemma 5.4, we first need an additional lemma that proves the intended relation between $\mathit{term}_{in}(s)$ and $\mathit{term}_{out}(s)$ of any given state s. The idea is that we can go from $\mathit{term}_{in}(s)$ to $\mathit{term}_{out}(s)$ if s is terminating and succeeds. On the way we collect the succeeding substitution.

Lemma A.2. *Let G be a termination graph, let s be an abstract state and let γ be a substitution conforming to s. Furthermore let I be the ITS corresponding to G. If $s\gamma$ succeeds and if $\mathit{term}_{in}(s)\gamma$ is terminating, then $\mathit{term}_{in}(s)\gamma \to_I^+ \mathit{term}_{out}(s)\gamma\theta$ holds true for some substitution θ and $\gamma\theta$ conforms to s.* ■

Proof. If $s\gamma$ succeeds, then there is a finite sequence of concrete states $\mathfrak{s}_1, \mathfrak{s}_2, \ldots, \mathfrak{s}_k$ such that $\mathfrak{s}_i \to \mathfrak{s}_{i+1}$ for all $i \in [1; n-1]$, $\mathfrak{s}_1 = s\gamma$ and such that $\mathfrak{s}_k = \langle\square\rangle$.

If s is an instance node, we follow its instance edge to some other node and continue doing so until we reach either a node that is not an instance node or we arrive back at s. We collect all the instance-substitutions along this path and receive the combined substitution σ. If we arrive back at s, then $\mathit{term}_{in}(s)\gamma$ evaluates to $\mathit{term}_{in}(s)\gamma\sigma$ and thus, $\mathit{term}_{in}(s)\gamma$ is nonterminating. This is a contradiction to our assumption. Hence, this cannot occur.

Thus, we assume that we arrive at some node s' which is not an instance node, but which is reachable from s by only following instance edges. We call the combined substitution along these edges σ and perform a case distinction on the type of s' to show that, if $\mathit{term}_{\mathit{in}}(s')\gamma\sigma$ is terminating in I, then it evaluates to $\mathit{term}_{\mathit{out}}(s')\gamma\sigma\theta$ for some θ. Thus, for the remainder of this proof we assume that $\mathit{term}_{\mathit{in}}(s')\gamma\sigma$ is terminating in I.

We show the claim by induction over k.

Base case: $k = 1$ If $k = 1$, then $s'\sigma\gamma$ is directly terminating, then it must be of the form $\langle\square\rangle$, whence s' is of the form $(\langle\square\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$. Thus, $(\langle\square\rangle, \mathcal{G}, \mathcal{U}, \mathcal{E}, \mathcal{A})$ is a success node and we have $\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{out}}(s')\sigma\gamma$, according to Definition 5.17. Hence, Lemma A.2 holds in this case.

Induction step: $k > 1$ Assume that $s'\sigma\gamma$ reaches a success state in $k-1$ steps and assume that the claim holds for all succeeding states that succeed in less than $k-1$ steps.

Case 1: s' **is a simple node** If s' is a simple node, it has a single successor s'', where s' evaluates to s'' according to the abstract semantics and, more precisely $s'\sigma\gamma$ evaluates to $s''\sigma\gamma$ according to the concrete semantics. Since $s'\sigma\gamma$ is succeeding in k steps and evaluates to $s''\sigma\gamma$ in one step, the latter state succeeds in $k-1$ steps. Furthermore, if $\mathit{term}_{\mathit{in}}(s'')\sigma\gamma$ is nonterminating, then $\mathit{term}_{\mathit{in}}(s)\gamma$ is nonterminating as well, since

$$\mathit{term}_{\mathit{in}}(s)\gamma \to_I^+ \mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{in}}(s'')\sigma\gamma$$

holds true. This is a contradiction to our assumption that $\mathit{term}_{\mathit{in}}(s)\gamma$ is terminating, hence $\mathit{term}_{\mathit{in}}(s'')\sigma\gamma$ is terminating. Thus, we can apply the induction hypothesis to s'' and receive that $\mathit{term}_{\mathit{in}}(s'')\sigma\gamma$ evaluates to $\mathit{term}_{\mathit{out}}(s'')\sigma\gamma\theta$ for some θ. Since s' is a simple node, I contains the rule $\mathit{term}_{\mathit{out}}(s'') \to \mathit{term}_{\mathit{out}}(s') \mid \mathit{true}$ whence the statement

$$\mathit{term}_{\mathit{out}}(s'')\sigma\gamma\theta \to_I \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true.

Case 2: s' **is a unification node** If s' is a unification node, it has two successors, which we call s'_1 and s'_2 and which represent the case that the unification succeeds or fails, respectively.

Assume that the unification succeeds in $s'\sigma\gamma$ with the most general unifier σ'. Then $s'_1\sigma\gamma\sigma'$ succeeds in $k-1$ steps. Furthermore, since s' is a unification node, I contains the rule $\mathit{term}_{\mathit{in}}(s')\sigma' \to \mathit{term}_{\mathit{in}}(s'') \mid \mathit{true}$ and hence,

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma\sigma'$$

holds true. Using the same reasoning as in case 1, we find that $s'_1\sigma\gamma\sigma'$ must be terminating and we can apply the induction hypothesis to s'_1 to see that

$$\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma\sigma' \to_I^+ \mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\sigma'\theta$$

holds true. We can then apply the rule $\mathit{term}_{\mathit{out}}(s'_1)\sigma' \to \mathit{term}_{\mathit{out}}(s') \mid \mathit{true}$, which is part of I since s' is an evaluation node and receive that

$$\mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\sigma'\theta \to \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true and hence, we have $\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$.

Now assume that the unification fails in $s'\sigma\gamma$. We then have the same situation as we had in case 1 if the backtrack rule was applied to s'. Hence, we can copy the proof from this case with s'_2 as the successor node and receive that $\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$ holds true.

Case 3: s' **is a comparison node** If s' is a comparison node and $s'\sigma\gamma$ succeeds, then the comparison evaluated in s' must either be true or false, but the evaluation of the expressions on either side must succeed without an error.

Assume that the comparison in s' succeeds. Then there is a successor node s'_1 of s' that models this as well as a rule $\mathit{term}_{\mathit{in}}(s') \to \mathit{term}_{\mathit{in}}(s'_1) \mid t_1 \bowtie t_2$ in I, where $t_1 \bowtie t_2$ is the comparison that succeeds in $s'\sigma\gamma$. Since this comparison succeeds, we see that

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma$$

holds true. Using the same reasoning as in case 1, we infer that $\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma$ is terminating. We furthermore see that $s'_1\sigma\gamma$ succeeds in $k-1$ steps, whence we can apply the induction hypothesis to it and receive the fact that

$$\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\theta$$

holds true. Since we already know that $(t_1 \bowtie t_2)\sigma\gamma$ is a tautology, $(t_1 \bowtie t_2)\sigma\gamma\theta$ is as well. Hence, we can apply the rule $\mathit{term}_{\mathit{out}}(s'_1) \to \mathit{term}_{\mathit{out}}(s') \mid t_1 \bowtie t_2$ and find that

$$\mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\theta \to_I \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true.

Now assume that the comparison in s' fails. We can apply the same reasoning as before using s'_2 as the successor node and the knowledge that, if $(t_1 \bowtie t_2)\sigma\gamma$ is false, then $\neg(t_1 \bowtie t_2)\sigma\gamma$ is a tautology. Hence, we receive the fact that

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true for some θ in this case as well.

Case 4: s' **is an arithmetic assignment node** If s' is an arithmetic assignment node and $s'\sigma\gamma$ succeeds, then the evaluation of the expression in the arithmetic assignment succeeds. We perform a case distinction based on whether or not its result unifies with the left-hand side of the assignment predicate.

Assume that the unification succeeds. Then we know that the condition $t_1 \bowtie t_2$ holds true in $s'\sigma\gamma$, i.e., $(t_1 \bowtie t_2)\sigma\gamma$ is a tautology. We furthermore know that $s'\sigma\gamma$ evaluates to $s'\sigma\gamma\sigma'$ for the unification σ' Using the same reasoning as before, we see that we can use the rule $\mathit{term}_{\mathit{in}}(s') \to \mathit{term}_{\mathit{in}}(s'_1)\sigma' \mid t_1 \bowtie t_2$ to find that

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{in}}(s'_2)\sigma\gamma\sigma'$$

holds true. We then apply the induction hypothesis to $\mathit{term}_{\mathit{in}}(s'_2)$ to see that

$$\mathit{term}_{\mathit{in}}(s'_2)\sigma\gamma\sigma' \to_I^+ \mathit{term}_{\mathit{out}}(s'_2)\sigma\gamma\sigma'\theta$$

holds true as well. We can then finally apply the rule $\mathit{term}_{\mathit{out}}(s'_2)\sigma' \to \mathit{term}_{\mathit{out}}(s') \mid t_1 \bowtie t_2$ to infer that

$$\mathit{term}_{\mathit{out}}(s'_2)\sigma\gamma\sigma'\theta \to_I \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true.

The case in which the unification does not succeed is identical to the case of a simple node or that of failing unification in case 2. We can copy the proof from the former case and receive the fact that

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s')\sigma\gamma\theta$$

holds true in this case as well.

Case 5: s' is a split node If s' is a split node, then we know that $s'\sigma\gamma$ only has a single goal of the form $t\sigma\gamma, T\sigma\gamma$. Since $s'\sigma\gamma$ is succeeding, both the term $t\sigma\gamma$ as well as the sequence of terms $T\sigma\gamma$ are succeeding. Furthermore, since both need to be evaluated to success for $s'\sigma\gamma$ to be evaluated, we know that both terms succeed in less than k steps.

Let s'_1 and s'_2 be the successors of s'. We first apply the first split rule $\mathit{term}_{\mathit{in}}(s') \to \mathit{term}_{\mathit{in}}(s'_1) \mid \mathit{true}$ to receive the fact that

$$\mathit{term}_{\mathit{in}}(s')\sigma\gamma \to_I \mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma$$

holds true. We then apply the induction hypothesis to s'_1 with the concretizing substitution $\sigma\gamma$. This implies that if $\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma$ is terminating

$$\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma \to_I^+ \mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\theta$$

holds true. Using the same reasoning as before with the rule $\mathit{term}_{\mathit{in}}(s') \to \mathit{term}_{\mathit{in}}(s'_1) \mid \mathit{true}$ in I, we infer that $\mathit{term}_{\mathit{in}}(s'_1)\sigma\gamma$ is terminating, since otherwise $\mathit{term}_{\mathit{in}}(s')\sigma\gamma$ would not be terminating.

We can then apply the rule $\mathit{term}_{\mathit{out}}(s'_1) \to \mathit{term}_{\mathit{in}}(s'_2) \mid \mathit{true}$ to find that

$$\mathit{term}_{\mathit{out}}(s'_1)\sigma\gamma\theta \to_I \mathit{term}_{\mathit{in}}(s'_2)\sigma\gamma\theta$$

holds true. Following this, we can apply the induction hypothesis again to s'_2 to receive that, if $\mathit{term}_{\mathit{in}}(s'_2)\sigma\gamma$ is terminating, then

$$\mathit{term}_{\mathit{in}}(s'_2)\sigma\gamma\theta \to_I^+ \mathit{term}_{\mathit{out}}(s'_2)\sigma\gamma\theta\theta'$$

holds true. We again infer that $term_{in}(s_2')\sigma\gamma$ is terminating, since its nontermination would imply nontermination of $term_{in}(s')\sigma\gamma$.

Using the final split rule $term_{out}(s_2') \rightarrow term_{out}(s') \mid true$, we can now see that

$$term_{out}(s_2')\sigma\gamma\theta\theta' \rightarrow_I term_{out}(s')\sigma\gamma\theta\theta'$$

holds true.

Case 6: s' is a parallel node The proof in this case is very similar to the previous case. If s' is a parallel node, it is of the form $s' = \langle g \mid G \rangle$, where either $\langle g\gamma\sigma \rangle$ or $\langle G\gamma\sigma \rangle$ is succeeding in less steps than $\langle g\gamma\sigma \mid G\gamma\sigma \rangle$.

If $\langle g\gamma\sigma \rangle$ is succeeding in less steps than $\langle g\gamma\sigma \mid G\gamma\sigma \rangle$, we apply the induction hypothesis to the left-hand successor of s' and use the rules $term_{in}(s') \rightarrow term_{in}(s_1') \mid true$ and $term_{out}(s_1') \rightarrow term_{out}(s') \mid true$ to argue that, if $term_{in}(s)\gamma$ terminates, then it evaluates to $term_{out}(s)\gamma\theta$. In the other case, namely in the case that $\langle G \rangle$ succeeds in less steps than $\langle g \mid G \rangle$, we apply the induction hypothesis to the right-hand successor of s' and receive the same result.

Hence, in both cases the lemma holds true.

Conclusion We have shown that $term_{in}(s')\sigma\gamma$ evaluates to $term_{out}(s')\gamma\sigma\theta$ for some θ. Hence the induction step holds true in this case.

We can now backtrack through the chain of instance edges that we took at the beginning and see that $term_{out}(s')\sigma\gamma\theta$ evaluates to $term_{out}(s)\gamma\theta$. Thus, under the assumption that $term_{in}(s)\gamma$ terminates, it evaluates to $term_{out}(s)\gamma\theta$.

We have shown that the claim of Lemma A.2 holds for states that terminate immediately as well as for those that take at least a single evaluation to evaluate successfully. Hence, according to the principle of induction, Lemma A.2 holds true. □

As stated previously, we now use Lemma A.2 to show that Lemma 5.4 holds true.

Lemma A.3 (Reprint of Lemma 5.4 on page 104). *Let G be a termination graph, let s be a state in G and let $\mathfrak{s}$ be a nonterminating concrete state that is represented by s with the concretizing substitution γ. Then there exists a successor s' of s in G that represents a concrete state $\mathfrak{s}'$ with the concretizing substitution γ', such that $\mathfrak{s}'$ is nonterminating. Furthermore, either $term_{in}(s)\gamma$ is nonterminating or $term_{in}(s)\gamma \rightarrow_I^+ term_{in}(s')\gamma'$ holds true.* ■

Proof. We show Lemma A.3 using a case distinction on the type of s.

Case 1: s is a simple node If s is a simple node, then there exists a successor node s' of s such that s' represents the concrete successor $\mathfrak{s}'$ of $\mathfrak{s}$ with the concretizing substitution $\gamma' = \gamma$, according to the soundness of the abstract semantics, which was shown in Lemma 4.8. Also, since s is simple, I contains the rule $term_{in}(s) \rightarrow term_{in}(s') \mid true$, whence the statement

$$term_{in}(s)\gamma \rightarrow_I^+ term_{in}(s')\gamma$$

holds true. Thus, the lemma holds true in this case.

Case 2: s is a unification node If s is a unification node, then the unification that occurs in s either succeeds or fails. Let s_1' and s_2' be the successor nodes of s that model success and failure of the unification, respectively.

Assume that the unification succeeds. In this case, since the unification rule is sound, s_1' represents the concrete state $\mathfrak{s}'$, which is the concrete successor state of $\mathfrak{s}$ with the concretizing substitution $\gamma' = \gamma\sigma$, where σ is the unifier applied in the unification. Also, since s is a unification node, I contains the rule $\mathit{term}_{\mathit{in}}(s) \to \mathit{term}_{\mathit{in}}(s_1')\sigma \mid \mathit{true}$, which we can apply to $\mathit{term}_{\mathit{in}}(s)\gamma$ to receive that

$$\mathit{term}_{\mathit{in}}(s)\gamma \to \mathit{term}_{\mathit{in}}(s_1')\gamma\sigma$$

holds true.

In the case that the unification does not succeed, we copy the proof from case 1 for the successor node s_2' and receive the truth of the statement

$$\mathit{term}_{\mathit{in}}(s)\gamma \to \mathit{term}_{\mathit{in}}(s_2')\gamma\sigma$$

in this case as well.

Case 3: s is a comparison node If s is a comparison node, then there are three possible successor states for $\mathfrak{s}$, namely the one in which the evaluation of either of the expressions results in an error, as well as those in which the comparison succeeds and fails, respectively, which are represented by the abstract states s_1', s_2' and s_3' respectively.

Assume that the evaluation of either expression in $\mathfrak{s}$ fails. Then $\mathfrak{s}$ evaluates to the terminal error state, whence $\mathfrak{s}$ is terminating. This contradicts our assumption, whence this case cannot occur.

Now assume that the comparison at the head of $\mathfrak{s}$ succeeds. Then $\mathfrak{s}$ evaluates to $\mathfrak{s}'$, which is nonterminating as well and represented by s_2'. Since the evaluation in $\mathfrak{s} = s\gamma$ succeeds and since I contains the rule $\mathit{term}_{\mathit{in}}(s) \to \mathit{term}_{\mathit{in}}(s_2') \mid t_1 \bowtie t_2$, where $t_1 \bowtie t_2$ is the comparison in $\mathfrak{s}$, we can apply this rule to $\mathit{term}_{\mathit{in}}(s)\gamma$ and receive that the statement

$$\mathit{term}_{\mathit{in}}(s)\gamma \to_I \mathit{term}_{\mathit{in}}(s_2')\gamma$$

holds true.

If the comparison at the head of $\mathfrak{s}$ fails, we employ the same argument as in the previous case with the successor state s_3' and the rule $\mathit{term}_{\mathit{in}}(s) \to \mathit{term}_{\mathit{in}}(s_3') \mid \neg(t_1 \bowtie t_2)$. This argumentation yields the truth of the fact

$$\mathit{term}_{\mathit{in}}(s)\gamma \to_I \mathit{term}_{\mathit{in}}(s_3')\gamma$$

Case 4: s is an arithmetic assignment node If s is an arithmetic assignment node, then the evaluation of $s\gamma$ falls into one of three cases. Either the evaluation of the expression on the right-hand side of the arithmetic assignment fails, the unification of its result with the left-hand side of the assignment succeeds, or the unification fails. We call the three successor states of s that represent these cases s_1', s_2' and s_3', respectively.

As argued before, it can not be the case that the evaluation of the expression fails, since $s\gamma$ is nonterminating and such a failure would terminate the computation. Hence, we consider the latter two cases.

Assume that the unification of the result of the right-hand side of the assignment with its left-hand side succeeds. Then $s\gamma$ evaluates to $s\gamma\sigma$, where σ is the unifier used in the unification and $s\gamma\sigma$ is represented by s'_2, according to the proof of the soundness of the abstract semantics. We apply the rule $\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_2)\sigma \mid t_1 \bowtie t_2$, which is possible since the unification of t_1 and t_2 succeeds, and receive the truth of the statement

$$\mathit{term}_{in}(s)\gamma \to_I \mathit{term}_{in}(s'_2)\gamma\sigma$$

where $\gamma' := \gamma\sigma$ and $s'_2\gamma\sigma$ is nonterminating.

Now assume that the unification fails. In this case we can simply copy the proof from case 1 or the latter subcase of case 2 using s'_3 as the successor node. This also yields the truth of the statement

$$\mathit{term}_{in}(s)\gamma \to_I \mathit{term}_{in}(s'_3)\gamma$$

where $s'_3\gamma$ is nonterminating.

Case 5: s is a split node Let s be a nonterminating split node and let s'_1 and s'_2 be its left- and right-hand successor, respectively. Also, let $\mathfrak{s}$ be a nonterminating concrete state in $\mathit{Conc}(s)$ with the concretizing substitution γ. Due to Lemma 5.2, we know that either s'_1 or s'_2 is nonterminating.

s'_1 is nonterminating Since the knowledge base of s'_1 is the same as that of s, we know that γ conforms to s'_1. Furthermore we can see from the proof of Lemma 5.2 that $\langle t\gamma\rangle$ is nonterminating. Finally, since s is a split node, I contains the rule $\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_1) \mid \mathit{true}$, whence $\mathit{term}_{in}(s)\gamma \to_I \mathit{term}_{in}(s'_1)\gamma$ holds true.

s'_1 is terminating and the evaluation of $s'_1\gamma$ fails Since $s'_1\gamma = \langle t\gamma\rangle$ fails, the evaluation of $s\gamma = \langle t\gamma, T\gamma\rangle$ fails as well. However, a failing evaluation is terminating by definition, and hence s is terminating. This is a contradiction to our assumption that s is nonterminating. Thus, this case cannot occur.

s'_1 is terminating and the evaluation of $s'_1\gamma$ succeeds We show that there exists a concretization θ such that $\sigma^{-1}\gamma\theta$ conforms to s'_2, that $s'_2\sigma^{-1}\gamma\theta$ is nonterminating and that $\mathit{term}_{in}(s)\gamma \to_I^+ \mathit{term}_{in}(s'_2)\sigma^{-1}\gamma\theta$ holds true. We start by showing the latter statement. For this we show that the following three evaluations are possible in I:

$$\begin{gathered}\mathit{term}_{in}(s)\gamma \to_I^+ \mathit{term}_{in}(s'_1)\gamma \\ \mathit{term}_{in}(s'_1)\gamma \to_I^+ \mathit{term}_{out}(s'_1)\gamma\theta \\ \mathit{term}_{out}(s'_1)\gamma\theta \to_I^+ \mathit{term}_{in}(s'_2)\sigma^{-1}\gamma\theta\end{gathered}$$

Since s is a split node, I contains the rule $\mathit{term}_{in}(s) \to \mathit{term}_{in}(s'_1) \mid \mathit{true}$, whence $\mathit{term}_{in}(s)\gamma \to_I^+ \mathit{term}_{in}(s'_1)\gamma$ obviously holds true. Additionally, since $s'_1\gamma$ is succeeding and terminating, it holds true that either $term_{in}(s'_1\gamma)$ is nonterminating or that there is a substitution θ such that $term_{in}(s'_1)\gamma \to_I^+ term_{out}(s'_1)\gamma\theta$ holds true, due to Lemma A.2. In the former case we know that $term_{in}(s)\gamma$ is nonterminating, as it evaluates to the nonterminating term $term_{in}(s'_1)\gamma$. Otherwise, $term_{in}(s'_1)\gamma \to_I^+ term_{out}(s'_1)\gamma\theta$ holds true.

Finally, due to the construction of split edges, we see that the set of ground variables of s'_1 and s are identical, as well as that the set of ground variables of s'_2 is the same as that of s up to the renaming σ. Since the arguments to $term_{in}(s)$ and $term_{out}(s)$ are the ground variables of s, we thus have

$$term_{in}(s'_2) = f^{in}_{s'_2}(\mathcal{G}_{s'_2}) = f^{in}_{s'_2}(\mathcal{G}_s\sigma) = f^{in}_{s'_2}(\mathcal{G}_{s'_1}\sigma)$$

Hence, it holds true that $term_{out}(s'_1)\gamma\theta$ evaluates to $term_{in}(s'_2)\sigma^{-1}\gamma\theta$, since I contains the rule $term_{out}(s'_1) \to term_{in}(s'_2) \mid true$.

It remains to show that $\sigma^{-1}\gamma\theta$ conforms to s'_2 and that $s'_2\sigma^{-1}\gamma\theta$ is nonterminating. The former statement holds true mainly since $\gamma\theta$ conforms to s'_1 according to Lemma A.2. Hence, $\sigma^{-1}\gamma\theta$ conforms to the knowledge base $(\mathcal{G}\sigma, \mathcal{U}\sigma, \mathcal{E}\sigma, \mathcal{A}\sigma)$. Thus, we only need to show that $\mathcal{V}(\sigma^{-1}\gamma\theta(X)) = \emptyset$ for all $X \in NextG(t, \mathcal{G})\sigma$. Since $\sigma\sigma^{-1} = id$, this is the same as showing that $\mathcal{V}(\gamma\theta(X)) = \emptyset$ holds true for all $X \in NextG(t, \mathcal{G})$. This holds true due to the soundness of $NextG$ that we assumed.

The final statement to show is that $s'_2\sigma^{-1}\gamma\theta$ is nonterminating. This holds true since we assumed that $s = \langle t\gamma, T\gamma\rangle$ is nonterminating, but that $\langle t\gamma\rangle$ is terminating and succeeding. The substitution θ is picked in Lemma A.2 as a succeeding substitution of t, whence $\langle t\gamma, T\gamma\rangle$ evaluates to $T\gamma\theta$. Since we knew that s was nonterminating, $T\gamma\theta = s'_2\sigma^{-1}\gamma\theta$ must be nonterminating.

Thus $term_{in}(s)\gamma \to_I^+ term_{in}(s'_2)\sigma^{-1}\gamma\theta$ holds true, $s'_2\sigma^{-1}\gamma\theta$ is nonterminating and $\sigma^{-1}\gamma\theta$ conforms to s'_2 whence Lemma A.3 holds true in this case.

Case 6: s is a parallel node If s is a parallel node and $s\gamma$ is nonterminating, then $s\gamma$ has more than one goal. Either its first goal is nonterminating or one of its latter ones is. We call the successor nodes of s s'_1 and s'_2, respectively.

Assume that the first goal is nonterminating. Then s'_1 represents $s'_1\gamma$, which is nonterminating as well. Since I contains the rule $term_{in}(s) \to term_{in}(s'_1) \mid true$, we can apply this rule and receive

$$term_{in}(s)\gamma \to_I term_{in}(s'_1)\gamma$$

In the latter case, we can repeat the proof from the previous paragraph and receive that s'_2 represents the nonterminating concrete state $s'_2\gamma$ as well as the fact that

$$term_{in}(s)\gamma \to_I term_{in}(s'_2)\gamma$$

holds true.

Case 7: s is an instance node If s is an instance node, then s has a successor node s' for which $Conc(s) \subseteq Conc(s')$ holds true. Furthermore, we know that $s'\sigma = s$ holds true, where σ is the instantiating substitution associated with the instance edge from s to s'. Hence, $s'\sigma\gamma$ is the same as $s\gamma$, which is nonterminating. Also, since I contains the rule $term_{in}(s) \to term_{in}(s')\sigma \mid true$, we can apply this rule to $term_{in}(s)\gamma$ to receive that

$$term_{in}(s)\gamma \to term_{in}(s')\gamma\sigma$$

holds true.

Conclusion We have shown the claim for all types of nodes s. Thus, Lemma A.3 holds true. □

Bibliography

[Bar15] Colin Barker. Solutions to exercises from [SS86, Chapter 8]. `http://colin.barker.pagesperso-orange.fr/sands.htm`, retrieved June 1, 2015.

[BCC$^+$05] Francisco Bueno, Daniel Cabeza, Manuel Carro, Manuel Hermenegildo, P López-Garcıa, and Germán Puebla. The Ciao Prolog System. Technical report, The Computational logic, Languages, Implementation, and Parallelism (CLIP) Group. School of CS, University of Madrid. CS and ECE Departments, University of New Mexico, July 2005.

[BCG$^+$07] Maurice Bruynooghe, Michael Codish, John P. Gallagher, Samir Genaim, and Wim Vanhoof. Termination Analysis of Logic Programs Through Combination of Type-based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2), April 2007.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, January 1977.

[CC14] Patrick Cousot and Radhia Cousot. Abstract Interpretation: Past, Present and Future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 2:1–2:10. ACM, July 2014.

[CT99] Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, October 1999.

[DEDC96] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: The Standard – Reference Manual.* Springer-Verlag Berlin Heidelberg, 1. edition, 1996.

[DL93] Saumya K. Debray and Nai-Wei Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[DLGH97] Saumya Debray, Pedro López-Garcıa, and Manuel Hermenegildo. Non-Failure Analysis for Logic Programs. In Lee Naish, editor, *Proceedings of* *the Fourteenth International Conference on Logic Programming*. MIT Press, July 1997.

[FGP+09] Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving Termination of Integer Term Rewriting. In Ralf Treinen, editor, *Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 32–47. Springer-Verlag Berlin Heidelberg, 2009.

[GC03] Samir Genaim and Michael Codish. Terminweb: Termination analyzer for logic programs. Sixth International Workshop on Termination, June 2003.

[Gie11] Jürgen Giesl. Termersetzungssysteme. Lecture Notes, RWTH Aachen University, `http://verify.rwth-aachen.de/tes15/TES.pdf`, September 2011.

[GSSK+12] Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs. Symbolic Evaluation Graphs and Term Rewriting – A General Methodology for Analyzing Logic Programs. In *Proceedings of the 14th International Symposium on Principles and Practice of Declarative Programming*, pages 1–12. ACM, September 2012.

[Het15] Werner Hett. P-99: Ninety-nine prolog problems. `https://sites.google.com/site/prologsite/prolog-problems`, retrieved May 28, 2015.

[HK03] Jacob M. Howe and Andy King. Efficient groundness analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.

[Hug15] Colin Hughes. Project Euler. `https://projecteuler.net/`, retrieved June 1, 2015.

[ISO94] ISO/IEC. Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic, December 1994.

[ISO95] ISO/IEC. Information technology – Programming languages – Prolog – Part 1: General core, April 1995.

[KK14] Jael Kriener and Andy King. Semantics for Prolog with Cut – Revisited. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, volume 8475 of *Lecture Notes in Computer Science*, pages 270–284. Springer International Publishing, June 2014.

[Lam15] Margaret Lamb. Examples of Prolog Operators and Arithmetic. `http://research.cs.queensu.ca/home/cisc260/2015w/examples/Arithmetic.pl`, retrieved June 1, 2015.

[LMS03] Vitaly Lagoon, Fred Mesnard, and Peter J. Stuckey. Termination Analysis with Types Is More Accurate. In Catuscia Palamidessi, editor, *Logic*

Programming, volume 2916 of *Lecture Notes in Computer Science*, pages 254–268. Springer-Verlag Berlin Heidelberg, 2003.

[MB05] Fred Mesnard and Roberto Bagnara. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1):243–257, 2005.

[Min01] Antoine Miné. The Octagon Abstract Domain. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings of the Workshop on Analysis, Slicing, and Transformation (AST'01)*, pages 310–319. IEEE CS Press, October 2001.

[NDS05] Manh Thang Nguyen and Danny De Schreye. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, volume 3668 of *Lecture Notes in Computer Science*, pages 311–325. Springer-Verlag Berlin Heidelberg, 2005.

[NDS07] Manh Thang Nguyen and Danny De Schreye. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 210–218. Springer-Verlag Berlin Heidelberg, 2007.

[OCM00] Enno Ohlebusch, Claus Claves, and Claude Marché. TALP: A Tool for the Termination Analysis of Logic Programs. In Leo Bachmair, editor, *Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 270–273. Springer-Verlag Berlin Heidelberg, 2000.

[PM06] Etienne Payet and Fred Mesnard. Nontermination Inference of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 28(2):256–289, March 2006.

[PR04] Andreas Podelski and Andrey Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251. Springer-Verlag Berlin Heidelberg, 2004.

[SDS03] Alexander Serebrenik and Danny De Schreye. Hasta-La-Vista: Termination Analyser for Logic Programs. In Frédéric Mesnard and Alexander Serebrenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments.* Katholieke Universiteit Leuven, Department of Computer Science, December 2003.

[SESK$^+$11] Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, and Carsten Fuhs. A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog, Full Version. Technical Report AIB-2011-08, RWTH Aachen, July 2011.

[SESK$^+$12] Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, and Carsten Fuhs. A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog. In Germán Vidal, editor, *Logic-Based Program Synthesis and Transformation*, volume 7225 of *Lecture Notes in Computer Science*, pages 237–252. Springer-Verlag Berlin Heidelberg, 2012.

[SGB$^+$14] Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp. Proving Termination and Memory Safety for Programs with Pointer Arithmetic. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 208–223. Springer International Publishing, 2014.

[SKGN10] Peter Schneider-Kamp, Jürgen Giesl, and Manh Thang Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In Danny De Schreye, editor, *Logic-Based Program Synthesis and Transformation*, volume 6037 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag Berlin Heidelberg, 2010.

[SKGST09] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic (TOCL)*, 11(1):2:1–2:52, October 2009.

[SLH14] Alejandro Serrano, Pedro López-García, and Manuel V. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming*, 14(4-5):739–754, 2014.

[Sma04] Jan-Georg Smaus. Termination of Logic Programs Using Various Dynamic Selection Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming*, volume 3132 of *Lecture Notes in Computer Science*, pages 43–57. Springer-Verlag Berlin Heidelberg, 2004.

[SS86] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. The MIT Press Cambridge, Massachusetts, London, England, 1986.

[SSKG11] Thomas Ströder, Peter Schneider-Kamp, and Jürgen Giesl. Dependency Triples for Improving Termination Analysis of Logic Programs with Cut. In María Alpuente, editor, *Logic-Based Program Synthesis and Transformation*, volume 6564 of *Lecture Notes in Computer Science*, pages 184–199. Springer-Verlag Berlin Heidelberg, 2011.

[Stä98] Robert F. Stärk. The theoretical foundations of LPTP (a logic program theorem prover). *The Journal of Logic Programming*, 36(3):241–269, September 1998.

[Str10] Thomas Ströder. Towards Termination Analysis of Real Prolog Programs. Diploma Thesis, RWTH Aachen University, February 2010.

[tpd15] Termination Problem Database. `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB`, retrieved March 23, 2015.

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58(5):345–363, 1936.

[VDS11] Dean Voets and Danny De Schreye. Non-termination analysis of logic programs with integer arithmetics. *Theory and Practice of Logic Programming*, 11(4-5):521–536, July 2011.