Rheinisch-Westfälische Technische Hochschule Aachen

Lehr- und Forschungsgebiet Informatik 2
Programmiersprachen und Verifikation

Towards Termination Analysis of Real Prolog Programs

Thomas Ströder

Diplomarbeit
im Studiengang Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen
im Februar 2010

Erstgutachter: Prof. Dr. Jürgen Giesl
Zweitgutachter: Prof. Dr. Peter Schneider-Kamp

Abstract

When developing programs it is of great practical interest to verify that the resulting programs have the desired properties. One of the most fundamental properties of programs is termination, i.e., that the program will not run forever, but compute some result in finitely many computation steps. The corresponding decision problem in computer science is the *halting problem*, i.e., given a description of a program and an input, decide whether the program terminates after finitely many steps or runs forever on that input.

Unfortunately, Turing showed this problem to be undecidable in general. Nevertheless, a huge number of analysis techniques which can automatically prove termination for many pairs of programs and inputs have been developed during the last decades. Nowadays, there are fully-automated tools that try to prove termination of a given program w.r.t. a given class of inputs.

However, most approaches for proving termination of programs are restricted to artificial programming languages having a comparatively simple mathematical definition and cannot handle essential features of programming languages used for real applications where exact mathematical definitions are very complex. This is especially true for logic programming, where most techniques for termination analysis are restricted to definite logic programs in contrast to real applications mostly written in the programming language Prolog, the main language for expert systems and applications from the artificial intelligence domain.

In this thesis, we extend the only existing approach known to be capable of handling logic programs with cuts to cover most of the features of real Prolog applications.

The contributions developed in this thesis are implemented in our fully automated termination prover AProVE. AProVE has reached the highest score for logic programming at the annual international Termination Competition, where the leading automated tools try to analyze termination of programs from different areas of computer science, in all years since 2004. In 2009, AProVE also was the only tool capable of successfully analyzing logic programs with cuts. The significance of our results is demonstrated by the empirical improvement AProVE shows on real Prolog applications used in the Termination Competition.

Acknowledgments

First, I would like to thank my supervisors Jürgen Giesl and Peter Schneider-Kamp for their great support. This thesis would hardly ever have become that extensive without the many fruitful discussions in person and on Skype. Despite paper deadlines, travels and accidents there was always time for my questions.

Second, I would also like to thank the members of the AProVE team for some nice chats - especially after 2pm - and more than one long hour at the Guinness House. Special thanks go to Carsten Fuhs and Fabian Emmes for having a look at preliminary versions of this thesis and giving useful feedback or just listening to crazy ideas.

I am grateful to my friends who tried to compensate my lack of sleep with funny messages and pictures keeping me in good spirits.

Finally, my thanks go to my family for being there, supplying me with too much good food and keeping most troubles away from me during the last months.

Thomas Ströder

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

Thomas Ströder

Contents

1 Introduction

Termination analysis of programs is a widely studied field. It deals with the most fundamental decision problem in computer science: the *halting problem*. Given a program and an input, we are interested in the question whether the program terminates after finitely many steps or whether it runs forever on that input. Unfortunately, [Tur36] showed this problem to be undecidable in general. Nevertheless, a huge number of analysis techniques has been developed to decide this problem for many pairs of programs and inputs. So even if we cannot decide the problem for all such pairs, it is of great practical interest to find fully automatable methods which can prove termination of practically relevant classes of programs and inputs.

Automatic termination analysis is an essential part of various verification techniques. As an example, consider theorem proving [BM79] which requires frequent termination proofs to operate correctly. The same is true for Knuth-Bendix completion [KB70] (see [MV06, WSW06, BKN07, SKW+09] for recent work on applying termination analysis in these areas). Another area where termination analysis is used is process verification. The liveness problems encountered there can often be reduced to termination problems [GZ03]. Furthermore, Microsoft uses termination analysis to statically verify their device drivers [CPR05]. Recently, techniques from termination analysis have even been adapted to solve synthesis problems in the area of behavior composition [SP09].

While termination analysis is widely used for artificial programming languages which correspond to a comparatively simple mathematical definition (e.g., term rewriting systems, lambda calculus, definite logic programs) it is also very important for programming languages used for real applications. Prolog is such a programming language especially used for expert systems and applications in the area of artificial intelligence. Prolog is based on logic programming and definite logic programs can be seen as a subset of all Prolog programs. Unfortunately, logic programming adheres to a lack of direction in the computation and this virtually guarantees that any non-trivial program terminates only for certain classes of inputs. So termination analysis is of particular interest here. Thus, termination of logic programs is widely studied (see e.g., [DD94] for an overview and [BCG+07, CLS05, CLSS06, DS02, LMS03, MR03, MS07, ND05, ND07, NGSD08, SD05, Sma04, Sch08, SGN09] for more recent work) and significant advances have been made during the last decades. Nowadays, there are fully-automated tools [LSS97, MB05, TGC02, SD03, GST06, ND07, OCM00] that try to prove termination of

a given logic program w.r.t. a given class of inputs. Nevertheless, there still remain many features of Prolog which cannot be successfully analyzed by any of these tools.

The state of the art in automated termination proving is assessed through the annual international Termination Competition [MZ07], where the leading automated tools try to analyze termination of programs from different areas of computer science gathered in the *termination problem database (TPDB)* [TPD09].

Among these tools, our fully automated termination prover AProVE [GST06] has reached the highest score for logic programming in the years 2004, 2005, 2006, 2007, 2008 and 2009.

In addition to that AProVE was the only tool capable of analyzing logic programs with cuts in 2009. The cut operator influences the control flow in Prolog and does not belong to the standard definition of definite logic programs. So this was a first step into the direction of analyzing real Prolog applications. In this thesis we will extend the methods used for termination analysis of logic programs with cuts to cover a large set of real Prolog applications.

From definite logic programs to Prolog

The fact that termination analysis of logic programs is widely studied is mostly due to the importance of termination analysis when one is developing and using Prolog programs.

Still, most techniques for termination analysis are limited to *definite* logic programs. There are several major differences between this notion and Prolog programs:

(i) In definite logic programs, the only method of computation is left-to-right, depth-first search (SLD resolution). In practice, virtually all Prolog programs make use of additional extra-logical constructs to **cut** the search space (! operator) or implement some kind of negation (\+ operator). Currently, the only approach for termination analysis of logic programs with cuts is given in [Sch08].

(ii) When speaking of termination, one might be interested in either universal termination (finiteness of the SLD tree) or existential termination (failure or first answer after a finite number of derivation steps). With very few exceptions (cf. [Mar96, Sch08]), only universal termination is analyzed.

(iii) In definite logic programs there is a clear distinction between predicate symbols and function symbols and, consequently, between atoms and terms. In Prolog there is no such distinction and when one is using so-called meta-programming, "atoms" may well be arguments of other atoms or terms. Using meta-programming and cuts, negation-as-failure [Cla78] can, e.g., be expressed by the two clauses $\backslash+(X) \leftarrow \mathsf{call}(X), !, \mathsf{fail}$ and $\backslash+(X)$. For an atomic query Q, $\backslash+(Q)$ can be proved if, and only if, Q fails.

(iv) SLD resolution uses unification with occurs-check. For efficiency, most Prolog implementations do not make use of the occurs-check. Except for the transformational approaches from [Sch08, SKW+09], all methods for termination analysis of logic programs assume unification with occurs-check.

(v) While there are no pre-defined predicates in definite logic programs, Prolog knows a huge number of so-called built-in predicates, whose definitions are not part of the program to analyze. Instead, their definitions are part of the interpreter or compiler and they may even have side effects which do not correspond to the mathematical background of definite logic programs at all. While different Prolog implementations may know different built-in predicates, the ISO standard for Prolog [DEC96] defines a list of built-in predicates which have to be implemented by all standard conforming implementations. Up to now, there is no approach capable of successfully analyzing termination of Prolog programs using built-in predicates.

The only existing approach known to handle (i), (ii) and, to some extent, (iii) together is the non-termination-preserving pre-processing step for logic programs with cuts based on symbolic evaluation given in [Sch08]. This approach constructs cut-free definite logic programs for a given logic program with cuts where termination of the cut-free program implies termination of the original one. In this thesis we extend this approach to a transformation from Prolog programs to dependency triple problems [SGN09] to obtain a more precise method which can fully handle (iii) and (iv), too. We will also handle 26 commonly used built-in predicates defined in [DEC96] and, thus, even handle (v) to some extent. Hence, we will be able to cover a large set of real Prolog applications.

Contributions of this thesis

While this thesis is based on the pre-processing step from [Sch08], it contributes various extensions to this method:

C1: We extend the pre-processing to handle the essential built-in predicate call/1 and, thus, full meta-programming as described in [DEC96].

C2: We additionally handle errors due to undefined predicate or uninstantiated variable calls during the execution of a Prolog program.

C3: We extend the definitions and proofs from [Sch08] to rational terms as opposed to finite terms. Thus, we can handle unification without occurs-check.

C4: The symbolic evaluation from [Sch08] uses an operation which is not fully automatable in its current form. We give a fully automatable alternative operation which can be used for the same purpose instead.

C5: The approach from [Sch08] uses a special representation of the evaluation process of Prolog for a given query and abstracts from this representation by simulating the evaluation of classes of queries at once. We improve the precision and speed of the operations used for this simulation.

C6: We handle the effects of 24 commonly used built-in predicates other than call/1 and !/0 as defined in [DEC96].

C7: We prove that our representation of the evaluation process of Prolog corresponds to the operational semantics of Prolog as defined in [DEC96].

C8: The symbolic evaluation from [Sch08] is non-deterministic. We present a heuristic to make this approach deterministic and, thus, fully automatable. We also show that the presented heuristic is successful on a large set of example programs.

C9: Instead of synthesizing cut-free logic programs we turn the pre-processing step into a transformation from Prolog programs to dependency triple problems. We exploit special properties of dependency triple problems to obtain a more powerful method for termination analysis of Prolog programs.

C10: The theoretical contributions of this thesis except for the handling of rational terms have been implemented in the fully automated termination prover AProVE and tested on a set of about 400 example programs. Some of these contributions were already used in the international Termination Competition 2009.

C11: The new possibilities and problems with our method gave rise to a great number of new examples which have been submitted to the TPDB and where already 76 examples were accepted and used for the international Termination Competition 2009.

Structure of the Thesis

In Chapter 2 we establish some basic notions about termination, terms, logic programming, Prolog and dependency triples which are used throughout this thesis.

We continue in Chapter 3 by recapitulating the basics for the pre-processing step from [Sch08] and extending it to also handle full meta-programming (C1), errors caused by undefined predicate or uninstantiated variable calls (C2), and unification without occurs-check together with rational terms which may be present due to unification without occurs-check (C3). Additionally, we present some extensions to improve the precision, speed and automation of the pre-processing step (C4) and (C5). Thus, this chapter describes a method for termination analysis of Prolog programs capable of handling (i) – (iv).

In Chapter 4 we present further extensions to the pre-processing method to additionally handle 24 built-in predicates other than call/1 and !/0 which are commonly used in real Prolog applications (C6). Therefore, our method can (to some extent) handle (v), too. We also discuss the problems which need to be solved in order to handle more built-in predicates. Moreover, we adapt the results from Chapter 3 to the extended set of handled built-in predicates.

Furthermore, we prove in Chapter 5 that the representation of the evaluation process of Prolog introduced in Chapter 3 and Chapter 4 corresponds to the operational semantics for Prolog programs as defined in the ISO standard for Prolog [DEC96] (C7).

We give a heuristic for the order in which the presented operations can be applied (C8) in Chapter 6 and prove that our approach is always terminating using this heuristic.

Afterwards, we turn the pre-processing step from logic programs with cuts to cut-free logic programs into a transformation from Prolog programs to dependency triple problems (C9) in Chapter 7 where termination of the resulting dependency triple problem implies termination of the original Prolog program.

Finally, in Chapter 8 we summarize the theoretical results of this thesis and state the empirical results evaluated with our fully-automated termination prover AProVE (C10).

Throughout the whole thesis we will illustrate our definitions with examples which have to a great extent been submitted to the TPDB used for the annual international Termination Competition (C11). In particular, Chapter 7 has one section dedicated to examples only.

2 Preliminaries

In this chapter we establish the basic definitions for termination, terms, logic programming, Prolog and dependency triples used throughout this thesis.

Structure of the Chapter

First, we state the standard definitions for termination and logic programming in Section 2.1. While these definitions are only rarely used explicitly in this thesis, they are the underlying basis for the following work.

Afterwards, in Section 2.2 we describe the basic elements and notations used for Prolog programs according to the ISO standard for Prolog [DEC96].

Finally, we give the definitions and the central theorem for the dependency triple framework as given in [SGN09, Sch08] in Section 2.3.

2.1 Termination and Standard Logic Programming

First, we state the standard definitions for termination and logic programming as given in [Sch08]. Although we will not use all of these definitions explicitly in this thesis, they are the underlying basis for the following work.

Abstract Reductions, Termination

To model computation steps of a program, we use the concept of abstract reductions. An *abstract reduction system* is a pair $(\mathcal{A}, \rightarrow)$ where $\rightarrow$ is a binary relation over $\mathcal{A}$, i.e., $\rightarrow \subseteq \mathcal{A} \times \mathcal{A}$. For the sake of brevity, instead of $(a, b) \in \rightarrow$ we write $a \rightarrow b$.

Now, we model a computation by defining $\mathcal{A}$ to be a superset of the program states and $\rightarrow$ to be the transition relation from a certain program state to its successor state.

For a given state $a \in \mathcal{A}$, we say that that $\rightarrow$ is *terminating* w.r.t. a if and only if there is no infinite reduction $a \rightarrow a_0 \rightarrow a_1 \rightarrow \ldots$ Furthermore, we say that $\rightarrow$ is terminating if, and only if, it is terminating for all $a \in \mathcal{A}$.

In the remainder of this thesis, the set of program states $\mathcal{A}$ and the relation $\rightarrow$ will typically be queries and the left-to-right resolution used in logic programs.

Terms, Atoms and Substitutions

Logic programming relies on the basic concepts of (possibly infinite) terms and substitutions built over sets of function symbols and variables. We additionally need the notion of atoms built from predicates and terms. This leads to the following formal definition of sets of function and predicate symbols.

Definition 2.1 (Signature)**.** *A* signature *is a pair* (Σ, Δ) *where* Σ *and* Δ *are finite sets of* function *respectively* predicate symbols*. If* $\Delta = \varnothing$*, we often just write* Σ *instead of* $(\Sigma, \varnothing)$*.*

Each symbol in $f \in \Sigma \cup \Delta$ *has an* arity $n \geq 0$ *and we often write* f/n *instead of* f *to denote that* f *has arity* n*.*

In the following, we always assume that Σ contains at least one constant $f/0$. This is not a restriction, since enriching the signature by a fresh constant does not change the termination behavior. This assumption is useful to ensure that we can always build finite ground terms over a given signature.

Definition 2.2 (Terms)**.** *A* term *over* Σ *is a tree where every node is labeled with a function symbol from* Σ *or with a* variable *from* $\mathcal{V} = \{X, Y, \ldots\}$*. Every inner node with* n *children is labeled with some* $f/n \in \Sigma$*, while leaves are labeled with a variable* $X \in \mathcal{V}$ *or with* $f/0 \in \Sigma$*. We write* $f(t_1, \ldots, t_n)$ *for the term* t *with* root f *(denoted* $root(t) = f$*) and direct subtrees* $t_1, \ldots, t_n$*. A term* t *is called* finite *if all paths in the tree* t *are finite, otherwise it is* infinite*. A term is* rational *if it only contains finitely many different subterms. The sets of all finite terms, all rational terms, and all (possibly infinite) terms over* Σ *are denoted by* $\mathcal{T}(\Sigma, \mathcal{V})$*,* $\mathcal{T}^{rat}(\Sigma, \mathcal{V})$*, and* $\mathcal{T}^{\infty}(\Sigma, \mathcal{V})$*, respectively. If* $\vec{t}$ *is the sequence* $t_1, \ldots, t_n$*, then* $\vec{t} \in \vec{\mathcal{T}}^{\infty}(\Sigma, \mathcal{V})$ *means that* $t_i \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$ *for all* i*.* $\vec{\mathcal{T}}(\Sigma, \mathcal{V})$ *is defined analogously. We write* $\mathcal{T}(\Sigma)$ *instead of* $\mathcal{T}(\Sigma, \varnothing)$ *to denote* ground terms*, i.e., finite variable-free terms.*

Finally, for any set of variables $\mathcal{V}' \subseteq \mathcal{V}$ *and any term* $t \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$*, let* $\mathcal{V}'(t)$ *be the set of all variables from* $\mathcal{V}'$ *occurring in* t*, i.e.,* $\mathcal{V}'(X) = \{X\}$ *for* $X \in \mathcal{V}'$*,* $\mathcal{V}'(X) = \varnothing$ *for* $X \notin \mathcal{V}'$*, and* $\mathcal{V}'(f(t_1, \ldots, t_n)) = \bigcup_{1 \leq i \leq n} \mathcal{V}'(t_i)$*.*

Given the above definition, the formal definition for atoms is straightforward.

Definition 2.3 (Atom)**.** *An* atom *over* (Σ, Δ) *is a tree* $p(t_1, \ldots, t_n)$*, where* $p/n \in \Delta$ *and* $t_1, \ldots, t_n \in \mathcal{T}^{\infty}(\Sigma, \mathcal{V})$*.* $\mathcal{A}^{\infty}(\Sigma, \Delta, \mathcal{V})$ *is the set of atoms and* $\mathcal{A}^{rat}(\Sigma, \Delta, \mathcal{V})$ *(and* $\mathcal{A}(\Sigma, \Delta, \mathcal{V})$*, respectively) are the atoms* $p(t_1, \ldots, t_n)$ *where* $t_i \in \mathcal{T}^{rat}(\Sigma, \mathcal{V})$ *(and* $t_i \in \mathcal{T}(\Sigma, \mathcal{V})$*, respectively) for all* i*. We write* $\mathcal{A}(\Sigma, \Delta)$ *instead of* $\mathcal{A}(\Sigma, \Delta, \varnothing)$*.*

To address or replace certain subterms, we introduce the notion of a position. The intuition is that a position describes a path from the root of the term to the subterm.

Definition 2.4 (Position)**.** *For a term t we define the set of* positions $Occ(t)$ *as the least subset of* $\mathbb{N}^*$ *such that* $\varepsilon \in Occ(t)$ *and* $i\,pos \in Occ(t)$*, if* $t = f(t_1, \ldots, t_n)$*,* $1 \leq i \leq n$*, and* $pos \in Occ(t_i)$*. We denote the subterm of* t *at position* pos *as* $t|_{pos}$ *where* $t|_\varepsilon = t$ *and* $f(t_1, \ldots, t_n)|_{i\,pos} = t_i|_{pos}$*. For a position* $pos \in Occ(t)$ *we denote the replacement of* $t|_{pos}$ *in* t *with a term* s *by* $t[s]_{pos}$ *where* $t[s]_\varepsilon = s$ *and* $f(t_1, \ldots, t_n)[s]_{i\,pos} = f(t_1, \ldots, t_i[s]_{pos}, \ldots, t_n)$*. For two positions* pos_1 *and* pos_2 *we write* $pos_1 \lhd pos_2$ *iff* pos_1 *is a prefix of* pos_2 *and* $pos_1 \neq pos_2$*.*

A common operation on terms is the instantiation of a term by a substitution, i.e., the replacement of all occurrences of certain variables by certain terms.

Definition 2.5 (Substitution)**.** *A* substitution *is a function* $\theta : \mathcal{V} \to \mathcal{T}^\infty(\Sigma, \mathcal{V})$*. We define the* domain *of a substitution* θ *as* $Dom(\theta) = \{X \in \mathcal{V} \mid \theta(X) \neq X\}$ *and similarly the* range *of a substitution* θ *as* $Range(\theta) = \{\theta(X) \mid X \in Dom(\theta)\}$*.*

By abuse of notation, we extend substitutions homomorphically to work on terms, atoms, etc. by applying them to all variables occurring in these expressions. If θ *is a variable renaming (i.e., a one-to-one correspondence on* $\mathcal{V}$*), then* $\theta(t)$ *is a* variant *of* t*.*

Instead of $\theta(X)$ *we often write* $X\theta$*. We write* $\theta\sigma$ *to denote that the application of* θ *is followed by the application of* σ*, i.e.,* $X\theta\sigma = \sigma(\theta(X))$*. For* $Dom(\theta) = \{X_1, \ldots, X_n\}$ *with* $X_i\theta = t_i$*, we often write* $[X_1/t_1, \ldots, X_n/t_n]$*.*

The set of all substitutions over Σ *and* $\mathcal{V}$ *is denoted* $Subst(\Sigma, \mathcal{V})$*.*

Logic Programming

A *clause* c is a formula $H \leftarrow B_1, \ldots, B_k.$ with $k \geq 0$ and $H, B_i \in \mathcal{A}(\Sigma, \Delta, \mathcal{V})$. H is called the *head* of the clause c and $B_1, \ldots, B_k$ is called its *body*. A finite set of clauses $\mathcal{P} = \{c_1 \ldots c_n\}$ over (Σ, Δ) is a *definite logic program* (LP). A clause with empty body is a *fact* and a clause with empty head is a *query*. We usually omit "$\leftarrow$" in queries and just write "$B_1, \ldots, B_k$". The empty query is denoted $\Box$.

To define the evaluation mechanism of resolution, we need the concept of a (most general) unifier of two atoms or terms.

Definition 2.6 (Most General Unifier)**.** *A substitution* θ *is a* unifier *of two atoms or terms* s *and* t *if and only if* $s\theta = t\theta$*. We write* $s \sim t$ *if there is a unifier of* s *and* t*. We call* θ *a* most general unifier (mgu) *of* s *and* t *if and only if* θ *is a unifier of* s *and* t *and for every unifiers* σ *of* s *and* t *there is a substitution* μ *such that* $\sigma = \theta\mu$*.*

We briefly present the procedural semantics of logic programs based on SLD-resolution using the left-to-right selection rule implemented by most Prolog systems. More details on logic programming can be found in [Apt97], for example.

Definition 2.7 (Derivation)**.** *Let Q be a query $A_1, \ldots, A_m$, let c be a clause $H \leftarrow B_1, \ldots, B_k$. Then Q' is a* resolvent *of Q and c using θ (denoted $Q \vdash_{c,\theta} Q'$) if θ is the mgu[1] of A_1 and H, and $Q' = (B_1, \ldots, B_k, A_2, \ldots, A_m)\theta$. We call A_1 the* selected atom*.*

A derivation *of a program $\mathcal{P}$ and Q is a possibly infinite sequence $Q_0, Q_1, \ldots$ of queries with $Q_0 = Q$ where for all i, we have $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$ for some substitution θ_{i+1} and some fresh variant c_{i+1} of a clause of $\mathcal{P}$. For a derivation $Q_0, \ldots, Q_n$ as above, we also write $Q_0 \vdash^n_{\mathcal{P},\theta_1 \ldots \theta_n} Q_n$ or $Q_0 \vdash^n_{\mathcal{P}} Q_n$, and we also write $Q_i \vdash_{\mathcal{P}} Q_{i+1}$ for $Q_i \vdash_{c_{i+1},\theta_{i+1}} Q_{i+1}$. The query Q* terminates *for $\mathcal{P}$ if all derivations of $\mathcal{P}$ and Q are finite, i.e., if $\vdash_{\mathcal{P}}$ is terminating for Q.*

If we restrict the substitutions used in derivations to functions from $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$, i.e., to substitutions for which the range contains only finite terms, the above notion of derivation corresponds to SLD-resolution using the left-to-right selection rule typically found in logic programming. Without this restriction, the above notion of derivation coincides with logic programming without an occurs-check [Col82] as implemented in common Prolog systems such as SICStus or SWI.

Next, we need the concept of answer substitutions that define the results of the computation of a logic program.

Definition 2.8 (Answer Set)**.** *The* answer set *$Answer(\mathcal{P}, Q)$ for a logic program $\mathcal{P}$ and a query Q is defined as the set of all substitutions $\theta|_{\mathcal{V}(Q)}$ such that $Q \vdash^n_{\mathcal{P},\theta} \square$ for some $n \in \mathbb{N}$.*

Example 2.9. Consider the predicate $\mathsf{minus}/3$ that is true when the third argument is the first argument minus the second argument. The following two clauses constitute a logic program $\mathcal{P}$ over $(\{\mathsf{0}, \mathsf{s}\}, \{\mathsf{minus}\})$ where $\mathsf{0}$ and the successor function s are used to represent natural numbers:

$$\mathsf{minus}(X, \mathsf{0}, X) \leftarrow \square. \tag{1}$$

$$\mathsf{minus}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{minus}(X, Y, Z). \tag{2}$$

For the query $Q = \mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z)$ we obtain the following derivation:

$$\mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z) \vdash_{(2),[X/\mathsf{0},Y/\mathsf{0}]} \mathsf{minus}(\mathsf{0}, \mathsf{0}, Z) \vdash_{(1),[Z/\mathsf{0}]} \square$$

The set of answer substitutions is $Answer(\mathcal{P}, Q) = \{[Z/\mathsf{0}]\}$.

[1] Note that for finite sets of *rational* atoms or terms, unification is decidable, the mgu is unique modulo renaming, and it is a substitution with *rational* terms [Hue76].

Finally, we introduce the *call set* for a logic program w.r.t. a class of queries. This concept is especially used in the dependency triple framework.

Definition 2.10 (Call Set)**.** *Let $\mathcal{Q}$ be a set of queries and $\mathcal{P}$ be a logic program. Then the* call set *for $\mathcal{Q}$ w.r.t. $\mathcal{P}$ is defined as $Call(\mathcal{P}, \mathcal{Q}) = \{G_1 \mid Q \vdash^n_{\mathcal{P}} G_1, \ldots, G_m$ and $Q \in \mathcal{Q}, n \in \mathbb{N}\}$.*

Example 2.11. Consider again the logic program $\mathcal{P}$ for subtraction from Example 2.9 and the query set $\mathcal{Q} = \{\mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z)\}$. Then we have $Call(\mathcal{P}, \mathcal{Q}) = \{\mathsf{minus}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z), \mathsf{minus}(\mathsf{0}, \mathsf{0}, Z), \square\}$.

2.2 Prolog

Now we introduce the notions we need for termination analysis of Prolog programs as opposed to standard logic programming. For most of our definitions we refer to the ISO standard for Prolog [DEC96]. But since this standard leaves some features of Prolog undefined, we consider the behavior of some common implementations of Prolog such as SICStus or SWI in such cases.

As Prolog does not distinguish between predicate symbols and function symbols as in definite logic programs, we will use only one signature Σ containing all "predicate" and "function" symbols. Instead of atoms and terms we will just consider terms from $\mathcal{T}^{rat}(\Sigma, \mathcal{V})$.[2]

Another small change from standard logic programming to Prolog is the possibility to use *anonymous variables*. Instead of giving fresh names to variables only used once, one can use the underscore (_) at each position where a fresh singleton variable should be used. Thus, by replacing all underscores in a Prolog program by pairwise different fresh variables we obtain an equivalent program. We assume that such a replacement is done for each program and each query before we start our analysis.

According to the execution model of Prolog as defined in [DEC96], there are some special positions inside the terms of clause bodies or goals which may be executed. When referring to such a term, these positions are exactly those reachable from the root of the term by a path having only function symbols from the set $GoalJunctors = \{$,/2, ;/2, ->/2$\}$ except for the position itself. For the clause body or goal, these positions are all such positions in the terms belonging to the clause body or goal respectively.

[2] In spite of the fact that Prolog also uses numbers (especially unbounded integers, cf. [DEC96]) and, therefore, an infinite signature, we may still assume a finite signature as we do not handle the arithmetical features of Prolog. Instead, we will treat numbers as constants. We leave the analysis of arithmetical features of Prolog to future work.

Definition 2.12 (Predication Position, Predication)**.** *Given a term $t \in \mathcal{T}^{rat}(\Sigma, \mathcal{V})$ and a position $pos \in Occ(t)$, we call pos a* predication position *w.r.t. t iff for all positions $pos' \in Occ(t)$ with $pos' \lhd pos$ we have $root(t|_{pos'}) \in GoalJunctors$. Furthermore, we call $t|_{pos}$ a* predication *w.r.t. t. For a finite list L of terms $t_1, \ldots, t_k$ we also call every predication position $pos_i \in Occ(t_i)$ w.r.t. t_i a predication position w.r.t. L and $t_i|_{pos_i}$ a predication w.r.t. L.*

Example 2.13. Consider the term $t = ,(\mathsf{p}, ;(\mathsf{f}(,(\mathsf{r}, \mathsf{r})), \mathsf{q}))$. The predication positions w.r.t. t are:

- ε
- 1
- 2
- $2 \mid 1$
- $2 \mid 2$

The predications w.r.t. t in the same order are, therefore, given as follows.

- t
- p
- $;(\mathsf{f}(,(\mathsf{r}, \mathsf{r})), \mathsf{q})$
- $\mathsf{f}(,(\mathsf{r}, \mathsf{r}))$
- q

Although we do not distinguish between predicate and function symbols, we do make a distinction between individual cuts to make their scope explicit. However, this distinction is only necessary and correct if the cuts in question are predications w.r.t. the goal to execute. Concerning comparison or unification of terms, we must not make such a distinction. So we define a set of labeled cut operators $\mathsf{Cuts} = \bigcup_{m \in \mathbb{N}} \{!_m/0\}$ which we will use in the following definitions of goals and their transformation used in the ISO standard. Thus, we have to deal with terms not only containing function symbols from Σ, but also from Cuts. However, the latter may only occur in predication positions. For this reason we define special sets of terms we use in Prolog.

Definition 2.14 (Terms in Prolog)**.** *The terms we consider in Prolog are from the set $PrologTerms(\Sigma, \mathcal{V}) = \{t \in \mathcal{T}^{rat}(\Sigma \cup \mathsf{Cuts}, \mathcal{V}) \mid \forall pos \in Occ(t) : t|_{pos} \in \mathsf{Cuts} \implies pos$ is a predication position$\}$. The definition of predication positions and predications is therefore extended to work also on terms from $PrologTerms(\Sigma, \mathcal{V})$. Analogously, we define the set of ground terms for Prolog as $GroundTerms(\Sigma) = \{t \in \mathcal{T}(\Sigma \cup \mathsf{Cuts}, \varnothing) \mid \forall pos \in Occ(t) : t|_{pos} \in \mathsf{Cuts} \implies pos$ is a predication position$\}$. Finally, we also define the set of finite Prolog terms by $FinitePrologTerms(\Sigma, \mathcal{V}) = \{t \in \mathcal{T}(\Sigma \cup \mathsf{Cuts}, \mathcal{V}) \mid \forall pos \in Occ(t) : t|_{pos} \in \mathsf{Cuts} \implies pos$ is a predication position$\}$.*

Example 2.15. Clearly, we have $\mathcal{T}^{rat}(\Sigma, \mathcal{V}) \subseteq PrologTerms(\Sigma, \mathcal{V})$. Consider the terms $t_1 = ,(!_1, \mathsf{p})$ and $t_2 = ,(\mathsf{q}(!_2), \mathsf{p})$. While $t_1 \in PrologTerms(\Sigma, \mathcal{V})$, we have $t_2 \notin PrologTerms(\Sigma, \mathcal{V})$ as $1 \mid 1 \in Occ(t_2)$ is no predication position, but $t_2|_{1|1} = !_2 \in \mathsf{Cuts}$.

The following definition describes the main structures for Prolog programs which we will use in this thesis.

Definition 2.16 (Prolog)**.** *A* Prolog program[3] *is a finite list of* clauses $H \leftarrow B$ *where the head* H *is a finite term over* Σ *and* $\mathcal{V}$ *and the body* B *is a finite list of finite terms over* Σ *and* $\mathcal{V}$*. We call such lists* goals over Σ and $\mathcal{V}$*. While goals in clauses of* Prolog *programs may only contain finite terms, they may also contain infinite rational terms as well as labeled cut operators in general. The set of all goals over* Σ *and* $\mathcal{V}$ *is* $Goal(\Sigma, \mathcal{V}) = PrologTerms(\Sigma, \mathcal{V})^*$*. We denote the empty goal by* $\square$ *and the concatenation of two terms* t *and* t' *by* t, t'*. The concatenation of any term* t *with* $\square$ *(i.e.,* $t, \square$*) is again just* t*.*

Furthermore, for a Prolog *program* $\mathcal{P} = \{c_1, \ldots, c_k\}$*,* $Slice(\mathcal{P}, t)$ *denotes the set of all clauses for the root of* t*, i.e.,* $Slice(\mathcal{P}, p(t_1, \ldots, t_n)) = \{c_i \mid c_i = p(s_1, \ldots, s_n) \leftarrow B_i \in \mathcal{P}\}$ *and* $Slice(\mathcal{P}, X) = \varnothing$ *for* $X \in \mathcal{V}$*.*

The branching factor $branchingFactor(f, \mathcal{P})$ *of a function symbol* f *w.r.t. a* Prolog *program* $\mathcal{P}$ *is defined as the number of defining clauses for* f *in* $\mathcal{P}$*, i.e., the number of clauses* $H \leftarrow B \in \mathcal{P}$ *with* $root(H) = f$*.*

Whenever we refer to a clause $H \leftarrow B \in \mathcal{P}$*, we assume that the variables in* H *and* B *are freshly renamed.*

While Prolog syntactically allows for variables as predications, the ISO standard demands a transformation for clauses and goals such that no variable is a predication w.r.t. this clause or goal. Instead, these variables are replaced by applications of the built-in predicate call/1 to the same variables. To perform this transformation (which will also be used to set the scope for cut operators found in predication positions) we define a transformation function *Transformed*.

Definition 2.17 (Transformation of Goals)**.** *The function* $Transformed : Goal(\Sigma, \mathcal{V}) \times \mathbb{N} \rightarrow Goal(\Sigma, \mathcal{V})$ *is recursively defined by*

$$
\begin{array}{lcl}
Transformed(\square, m) & = & \square \\
Transformed((x, L), m) & = & \mathsf{call}(x), Transformed(L, m) \text{ for } x \in \mathcal{V} \\
Transformed((!, L), m) & = & !_m, Transformed(L, m) \\
Transformed((!_{m'}, L), m) & = & !_m, Transformed(L, m) \\
Transformed((f(t_1, t_2), L), m) & = & f(Transformed(t_1, m), Transformed(t_2, m)), \\
 & & Transformed(L, m) \text{ for } f \in GoalJunctors \\
Transformed((s, L), m) & = & s, Transformed(L, m) \text{ for } s \in PrologTerms(\Sigma, \mathcal{V}) \setminus \mathcal{V} \\
 & & \text{with } root(s) \notin GoalJunctors \cup \{!/0\} \cup \mathsf{Cuts}
\end{array}
$$

This function will be used to transform goals occurring in clause bodies or meta-calls during the execution of a Prolog program.

[3] We do not consider the real syntax used in Prolog implementations, but parse real Prolog programs and transform them into the representation stated here. See also [DEC96] for the flattening of conjunctions, disjunctions and if-then-else, interpretation of underscores, scopes for cut operators and the transformation of variable calls into applications of the built-in predicate call/1.

Example 2.18. Consider the query $Q = \mathsf{p}(X), !, X, \mathsf{call}(!)$. Its transformation to the scope m is $\mathit{Transformed}(Q, m) = \mathsf{p}(X), !_m, \mathsf{call}(X), \mathsf{call}(!)$. Note that the second cut is not labeled as it is no predication w.r.t. Q.

Finally, to denote the term resulting from replacing all occurrences of a function symbol f in a term t by another function symbol g, we introduce the notation $t[f/g]$.

Built-in Predicates

The ISO standard for Prolog defines a list of built-in predicates. According to this standard we define the set *BuiltInPredicates* as the set containing exactly the following function symbols:

- abolish/1
- arg/3
- =:=/2
- =\=/2
- >/2
- >=/2
- </2
- =</2
- asserta/1
- assertz/1
- at_end_of_stream/0
- at_end_of_stream/1
- atom/1
- atom_chars/2
- atom_codes/2
- atom_concat/3
- atom_length/2
- atomic/1
- bagof/3
- call/1
- catch/3
- char_code/2
- char_conversion/2
- clause/2
- close/1
- close/2
- compound/1
- ,/2
- copy_term/2
- current_char_conversion/2
- current_input/1
- current_op/3
- current_output/1
- current_predicate/1
- current_prolog_flag/2
- !/0
- ;/2
- fail/0
- findall/3
- float/1
- flush_output/0
- flush_output/1
- functor/3
- get_byte/1
- get_byte/2
- get_char/1
- get_char/2
- get_code/1
- get_code/2
- halt/0
- halt/1
- ->/2
- integer/1
- is/2
- nl/0
- nl/1
- nonvar/1
- \+/1
- number/1
- number_chars/2
- number_codes/2
- once/1
- op/3
- open/3
- open/4
- peek_byte/1
- peek_byte/2
- peek_char/1
- peek_char/2
- peek_code/1
- peek_code/2
- put_byte/1
- put_byte/2
- put_char/1
- put_char/2
- put_code/1
- put_code/2
- read/1
- read/2
- read_term/2
- read_term/3

- repeat/0
- retract/1
- set_input/1
- set_output/1
- set_prolog_flag/2
- set_stream_position/2
- setof/3
- stream_property/2
- sub_atom/5
- @>/2
- @>=/2
- ==/2
- @</2
- @=</2
- \==/2
- throw/1
- true/0
- \=/2
- =/2
- unify_with_occurs_check/2
- =../2
- var/1
- write/1
- write/2
- write_canonical/1
- write_canonical/2
- write_term/2
- write_term/3
- writeq/1
- writeq/2

We will give explanations for these built-in predicates and either special inference rules or a description of the problems we have with the respective built-in predicate for this approach in Chapter 4.

2.3 Dependency Triples

The dependency triple framework was introduced in [Sch08] and [SGN09] to obtain a modular and powerful framework for termination analysis of logic programs. It is adapted from the successful dependency pair framework [AG00, GTSF03, GTS05, TGS04, HM05, GTSF06] for term rewriting (see for example [Der87, Zan95, AG00, GTSF06, EWZ06, Sch08, FGP+09] for more information and recent work on termination analysis of term rewriting).

The basic structure in the dependency triple framework is very similar to a clause in logic programming.

Definition 2.19 (Dependency Triple [Sch08, SGN09])**.** *A* dependency triple *(DT) is a clause $H \leftarrow I, B$ where H and B are atoms and I is a list of atoms. For a logic program $\mathcal{P}$, the set of its dependency triples is $DT(\mathcal{P}) = \{H \leftarrow I, B \mid H \leftarrow I, B, \cdots \in \mathcal{P}\}$.*

Example 2.20. Consider again the logic program $\mathcal{P}$ for subtraction from Example 2.9. The set of its DTs is $DT(\mathcal{P}) = \{\mathsf{minus}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{minus}(X, Y, Z)\}$.

For dependency triples, a derivation is defined in terms of a chain.

Definition 2.21 (Chain [Sch08, SGN09])**.** *Let $\mathcal{D}$ and $\mathcal{P}$ be sets of clauses and let $\mathcal{C}$ be a set of atoms. A (possibly infinite) list $(H_0 \leftarrow I_0, B_0), (H_1 \leftarrow I_1, B_1), \ldots$ of variants from $\mathcal{D}$ is a* $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-chain *iff there are substitutions θ_i, σ_i and an $A \in \mathcal{C}$ such that $\theta_0 = mgu(A, H_0)$ and for all i, we have $\sigma_i \in Answer(\mathcal{P}, I_i\theta_i)$, $\theta_{i+1} = mgu(B_i\theta_i\sigma_i, H_{i+1})$, and $B_i\theta_i\sigma_i \in \mathcal{C}$.*

Example 2.22. Consider once more the logic program $\mathcal{P}$ from Example 2.9 with the query set $\mathcal{Q} = \{\mathsf{minus}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0)), Z)\}$, the set $\mathcal{D} = DT(\mathcal{P})$ and the set $\mathcal{C} = Call(\mathcal{P}, \mathcal{Q})$ $= \{\mathsf{minus}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0)), Z), \mathsf{minus}(\mathsf{s}(0), \mathsf{s}(0), Z), \mathsf{minus}(0, 0, Z), \square\}$. Then, for the derivation

$$\mathsf{minus}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0)), Z) \vdash_{(2),[X/\mathsf{s}(0),Y/\mathsf{s}(0)]} \mathsf{minus}(\mathsf{s}(0), \mathsf{s}(0), Z) \vdash_{(2),[X/0,Y/0]}$$
$$\mathsf{minus}(0, 0, Z) \vdash_{(1),[Z/0]} \square$$

we have the $(\mathcal{D}, \mathcal{C}, \mathcal{P})$-chain

$$\mathsf{minus}(\mathsf{s}(X_1), \mathsf{s}(Y_1), Z) \leftarrow \mathsf{minus}(X_1, Y_1, Z), \mathsf{minus}(\mathsf{s}(X_2), \mathsf{s}(Y_2), Z) \leftarrow \mathsf{minus}(X_2, Y_2, Z)$$

with the substitutions $\theta_0 = [X_1/\mathsf{s}(0), Y_1/\mathsf{s}(0)], \theta_1 = [X_2/0, Y_2/0]$ and $\sigma_0 = \sigma_1 = id$.

Finally, we introduce the notion of a dependency triple problem which essentially is a triple of a logic program, a call set and another logic program corresponding to the elements needed for chains.

Definition 2.23 (DT Problem [Sch08, SGN09])**.** *A* DT problem *is a triple* $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ *where* $\mathcal{D}$ *and* $\mathcal{P}$ *are finite sets of clauses and* $\mathcal{C}$ *is a set of atoms. A DT problem* $(\mathcal{D}, \mathcal{C}, \mathcal{P})$ *is* terminating *iff there is no infinite* $(\mathcal{D}, \mathcal{C}, \mathcal{P})$*-chain.*

From [Sch08, SGN09] we obtain the following result.

Theorem 2.24 (Termination Criterion [Sch08, SGN09])**.** *A logic program* $\mathcal{P}$ *is terminating for a set of atomic queries* $\mathcal{Q}$ *iff there is no infinite* $(DT(\mathcal{P}), Call(\mathcal{P}, \mathcal{Q}), \mathcal{P})$*-chain.*

The proof of this theorem can be found in [Sch08, SGN09].

Example 2.25. As we have seen in Example 2.22, the chain for the DT problem $(DT(\mathcal{P}), Call(\mathcal{P}, \mathcal{Q}), \mathcal{P})$ with $\mathcal{P}$ and $\mathcal{Q}$ defined as in Example 2.22 is finite. In fact, this chain is the only chain for this DT problem and, hence, the logic program $\mathcal{P}$ terminates w.r.t. $\mathcal{Q}$ as we have also seen in the derivation leading to $\square$.

Thus, we can use DT problems to analyze termination of logic programs and, hence, Prolog programs as we will see in Chapter 7.

3 Cuts, Meta-Programming and Rational Terms

As the title of this chapter indicates, the first step from definite logic programs to Prolog covers the handling of cuts, meta-programming and rational terms. Since these structures and mechanisms are not part of the standard definitions used in logic programming, we have to use an appropriate description of the evaluation method used in Prolog. For this reason we rely on the notion of concrete and abstract state-derivations introduced in [Sch08] which we will extend to cover more of the real Prolog behavior (in contrast to just logic programming with cuts). While this chapter is to some extent taken from [Sch08], we introduce some new definitions for the concrete and abstract inference rules to handle meta-programming and undefined predicate calls more precisely and adapt the proofs to unification without occurs-check.

Structure of the Chapter

In the remainder of this chapter, our goal is to prove universal termination of Prolog programs without built-in predicates other than !/0 and call/1 (these are necessary for handling cuts and meta-programming) for a (typically infinite) set of queries $\mathcal{Q}$. These sets are typically given by the user by providing a predicate and by specifying which of its arguments are instantiated by ground terms.

In Section 3.1 we introduce concrete states and a set of concrete inference rules which characterize the evaluation process used in Prolog programs without built-in predicates other than !/0 and call/1.

Since we want to prove termination w.r.t. sets of queries instead of single queries, we extend the concrete states and inference rules to abstract states and inference rules in Section 3.2. These rules for abstract states are the basis for the construction of so-called termination graphs which we will discuss in Chapter 6.

Finally, in Section 3.3 we introduce additional abstract inference rules which are needed to obtain a finite analysis, i.e., a finite termination graph instead of an infinite termination tree. To this end, we refer back to already existing states as instances and generalize or split states and goals to find such instances.

We summarize the contributions of this chapter in Section 3.4.

3.1 Concrete State-Derivations

A very essential feature of Prolog which does not belong to the standard definition of definite logic programs is the cut-operator (!). While there are attempts, e.g. by [DW90], to banish this operator from Prolog it is virtually always used in real Prolog applications. Instead of using all clauses for proving a goal, the unused clauses for the current goal are cut off when a cut-operator is reached during the evaluation process of Prolog. In other words, the backtracking possibilities for the current goal are dropped, once we reach a cut. On the one hand, this operator can be used to drop double or wrong results without changing the termination behavior. On the other hand, it can also cut off backtracking possibilities which would lead to an infinite derivation and, thus, change the termination behavior from non-termination to termination. Since it is not possible to change the termination behavior from termination to non-termination by just using cuts, an easy way of handling the cut-operator is just to ignore it. This is done in most automated termination provers, since a more precise handling of the cut-operator was not available before 2008, in spite of the fact that research was done on that problem for more than twenty years. We always assume $!/0 \in \Sigma$ in this thesis.

The following example program for division with remainder demonstrates the use of cut-operators in Prolog.

Example 3.1. Consider the following Prolog program for division with remainder divremain.pl

$$\begin{aligned}
\mathsf{div}(X, \mathsf{0}, Z, R) &\leftarrow !, \mathsf{failure}(\mathsf{a}). && (3)\\
\mathsf{div}(\mathsf{0}, Y, Z, R) &\leftarrow !, \mathsf{eq}(Z, \mathsf{0}), \mathsf{eq}(R, \mathsf{0}). && (4)\\
\mathsf{div}(X, Y, \mathsf{s}(Z), R) &\leftarrow \mathsf{minus}(X, Y, U), !, \mathsf{div}(U, Y, Z, R). && (5)\\
\mathsf{div}(X, Y, \mathsf{0}, X) &\leftarrow \square. && (6)\\
\mathsf{minus}(X, \mathsf{0}, X) &\leftarrow \square. && (7)\\
\mathsf{minus}(\mathsf{s}(X), \mathsf{s}(Y), Z) &\leftarrow \mathsf{minus}(X, Y, Z). && (8)\\
\mathsf{eq}(X, X) &\leftarrow \square. && (9)\\
\mathsf{failure}(\mathsf{b}) &\leftarrow \square. && (10)
\end{aligned}$$

and the set of queries $\mathcal{Q} = \{\mathsf{div}(t_1, t_2, t_3, t_4) \mid t_1, t_2 \text{ are ground}\}$. Any termination analyzer that ignores the cut must fail on this example as $\mathsf{div}(\mathsf{0}, \mathsf{0}, Z, R)$ leads to the subtraction of $\mathsf{0}$ using the third div-rule and, thus, starts an infinite derivation.

Another additional feature of Prolog is meta-programming. While [Sch08] already handles simple cases for meta-programming, the real Prolog behavior allows for a more complex use of cuts inside of meta calls. We will, thus, change the concrete inference rules from [Sch08] to cover the full Prolog behavior for meta-programming according to the ISO

standard for Prolog [DEC96]. Also, we always assume $\mathsf{call}/1 \in \Sigma$ in this thesis. However, the more complicated behavior of meta-calls in Prolog will only occur together with built-in predicates and, thus, we will come back to this issue in Chapter 4.

The following example program for addition of natural numbers demonstrates the use of meta-programming in form of negation-as-failure [Cla78] in Prolog.

Example 3.2. Consider the following Prolog program add3.pl

$$\mathsf{add}(X, \mathsf{0}, X) \leftarrow \square. \tag{11}$$
$$\mathsf{add}(X, Y, \mathsf{s}(Z)) \leftarrow \mathsf{no}(\mathsf{isZero}(Y)), \mathsf{p}(Y, P), \mathsf{add}(X, P, Z). \tag{12}$$
$$\mathsf{p}(\mathsf{0}, \mathsf{0}) \leftarrow \square. \tag{13}$$
$$\mathsf{p}(\mathsf{s}(X), X) \leftarrow \square. \tag{14}$$
$$\mathsf{isZero}(\mathsf{0}) \leftarrow \square. \tag{15}$$
$$\mathsf{no}(X) \leftarrow \mathsf{call}(X), !, \mathsf{failure}(\mathsf{a}). \tag{16}$$
$$\mathsf{no}(X) \leftarrow \square. \tag{17}$$
$$\mathsf{failure}(\mathsf{b}) \leftarrow \square. \tag{18}$$

and the set of queries $\mathcal{Q} = \{\mathsf{add}(t_1, t_2, t_3) \mid t_2 \text{ is ground}\}$. Again, any termination analyzer which ignores the cut (or cannot handle meta-programming) must fail on this example as $\mathsf{add}(X, \mathsf{0}, Z)$ would lead to the addition of $\mathsf{0}$ using the second add-rule and, thus, starts an infinite derivation.

Next, we also want to handle unification without occurs-check. While definite logic programs make use of the occurs-check to exclude the construction of infinite terms, most real Prolog implementations omit the occurs-check for efficiency. As a side-effect, one can construct and use infinite rational terms in Prolog. [Col82] suggests some ways to use this for elegant programs while there is a considerable part of the logic programming community which tries to avoid programming with infinite rational terms. This culminates in the fact that the ISO standard for Prolog [DEC96] does not define any behavior for infinite rational terms. However, most implementations offer this behavior and, thus, we have to deal with it in real Prolog applications.

The following example program used to check if a natural number is even demonstrates the use of unification without occurs-check in Prolog.

Example 3.3. Consider the following Prolog program even.pl

$$\mathsf{even}(X) \leftarrow \mathsf{eq}(Y, \mathsf{f}(\mathsf{e}, \mathsf{f}(\mathsf{o}, Y))), \mathsf{c}(Y, X). \tag{19}$$
$$\mathsf{c}(\mathsf{f}(\mathsf{e}, X), \mathsf{0}) \leftarrow \square. \tag{20}$$
$$\mathsf{c}(\mathsf{f}(Z, X), \mathsf{s}(Y)) \leftarrow \mathsf{c}(X, Y). \tag{21}$$
$$\mathsf{eq}(X, X) \leftarrow \square. \tag{22}$$

and the set of queries $\mathcal{Q} = \{\mathsf{even}(t) \mid t \text{ is ground}\}$. This program constructs an infinite rational term alternately labeled with the symbols e and o. Then it descends this term as deep as the natural number indicates and succeeds if the current symbol is e.

We also have to deal with some less intuitive behavior of Prolog programs without built-in predicates other than ! and call.

Example 3.4. The position of a cut inside a clause can change the termination behavior. To see this, consider the following two Prolog programs:

cutpos1.pl:

$$\begin{array}{lcl} \mathsf{p} & \leftarrow & \mathsf{r}. \\ \mathsf{r} & \leftarrow & \mathsf{q}, !. \\ \mathsf{r} & \leftarrow & \mathsf{q}. \\ \mathsf{q} & \leftarrow & \Box. \\ \mathsf{q} & \leftarrow & \mathsf{r}. \end{array}$$

cutpos2.pl:

$$\begin{array}{lcl} \mathsf{p} & \leftarrow & \mathsf{r}. \\ \mathsf{r} & \leftarrow & !, \mathsf{q}. \\ \mathsf{r} & \leftarrow & \mathsf{q}. \\ \mathsf{q} & \leftarrow & \Box. \\ \mathsf{q} & \leftarrow & \mathsf{r}. \end{array}$$

While cutpos1.pl is terminating for the query p, cutpos2.pl is not terminating for the same query.

Example 3.5. Obviously, the linear query $\mathsf{p}(X, Y)$ might not terminate while the non-linear query $\mathsf{p}(X, X)$ terminates. Consider for example the Prolog program consisting of the single clause $\mathsf{p}(\mathsf{a}, \mathsf{b}) \leftarrow \mathsf{p}(\mathsf{a}, \mathsf{b})$.

For definite logic programs, the linear query always allows more derivations. For Prolog programs this need not be the case. Consider for example the following two Prolog programs:

ts05.pl:

$$\begin{array}{lcl} \mathsf{q} & \leftarrow & \mathsf{p}(X, Y). \\ \mathsf{p}(\mathsf{a}, \mathsf{b}) & \leftarrow & !. \\ \mathsf{p}(\mathsf{a}, \mathsf{a}) & \leftarrow & \mathsf{p}(\mathsf{a}, \mathsf{a}). \end{array}$$

ts06.pl:

$$\begin{array}{lcl} \mathsf{q} & \leftarrow & \mathsf{p}(X, X). \\ \mathsf{p}(\mathsf{a}, \mathsf{b}) & \leftarrow & !. \\ \mathsf{p}(\mathsf{a}, \mathsf{a}) & \leftarrow & \mathsf{p}(\mathsf{a}, \mathsf{a}). \end{array}$$

While ts05.pl is terminating for the query q, ts06.pl is not terminating for the same query.

Example 3.6. It is important to know whether the analyzed Prolog implementation makes use of the occurs-check. Consider the following Prolog program terminate.pl

$$\begin{array}{rcl} \mathsf{t} & \leftarrow & \mathsf{eq}(X, \mathsf{f}(X)), !. \\ \mathsf{t} & \leftarrow & \mathsf{t}. \\ \mathsf{eq}(X, X) & \leftarrow & \Box. \end{array}$$

and the query t. If the implementation uses unification with occurs-check, this query does not terminate as the predicate eq will fail and we reach the second clause for t. If otherwise the occurs-check is omitted, the query terminates by reaching the cut.

The correct handling of meta-programming requires a transformation of variables used as predications into applications of the built-in predicate call. Thus, we cannot have variables in predication positions anymore and we can exclude them from the definitions of goals as we have done in Chapter 2. The required transformation is done by the *Transformed* function which we can use for the inference rule handling the call predicate and for clauses and queries from the program and input respectively.

In the preceding examples we have seen that the use of the cut in Prolog programs requires a more detailed analysis of the backtracking behavior than this would be necessary in definite logic programs. Instead of representing the current state of the computation by just goals and local substitutions organized in a search-tree, we choose a more explicit representation where backtrack information is given by lists of goals which are optionally labeled by the clauses that may be applied to these goals next. This, together with explicit marks for the scope of a cut, will allow us to express the non-local effect of the cut by local rules.

The main idea for handling cuts, as described in [Sch08], is to label each cut with a fresh natural number when it is introduced by a step in the derivation. By additionally inserting such a number into the backtracking list, we can determine the scope of the correspondingly labeled cut. Moreover, we must be able to update the scope of a cut according to meta-calls which restrict the effect of a cut to the calls inside the meta-call. This together with the problem that labeled cuts must be equal w.r.t. unification or term equality used in built-in predicates, can be handled using the transformation by the function *Transformed* which only labels cuts in predication positions. These cuts cannot be an argument of another predicate performing unification, equality tests or meta-calls.

Putting everything together, our states are lists of three different types of elements:

- The list may contain an *unlabeled goal* $Q \in \mathit{Goal}(\Sigma, \mathcal{V})$ which just represents itself.
- A *labeled goal* $Q^i_m \in \mathit{Goal}(\Sigma, \mathcal{V}) \times \mathbb{N} \times \mathbb{N}$ represents that we must apply the i-th clause in the program to the goal Q. The m determines how a cut introduced by the body of the i-th clause will be labeled.
- A natural number $m \in \mathbb{N}$ in our backtracking lists marks that, when a cut labeled by m is reached, all elements preceding m are discarded. We denote m as $?_m$ in our backtracking lists.

The following example demonstrates the intended use of these states.

Example 3.7. Consider again the logic program for div from Example 3.1 and the query $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)$. This would be represented by the concrete state consisting of just the goal $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)$. As this atom unifies with the head of all four clauses for div, we obtain the identical behavior from the state $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^3_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^4_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^5_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^6_1 \mid ?_1$. This denotes that we first try to apply clause (3) and then backtrack using first clause (4), second clause (5) and finally clause (6). Now, we can evaluate the first labeled goal using (3) and obtain $!_1, \mathsf{failure}(\mathsf{a}) \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^4_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^5_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^6_1 \mid ?_1$. By applying the cut we get rid of the backtracking goals $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^4_1$, $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^5_1$ and $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^6_1$ and obtain the state $\mathsf{failure}(\mathsf{a}) \mid ?_1$, which eventually fails, since no clause is applicable. Backtracking leads us to the state $?_1$ and finally to the empty word ε where the computation stops. Note that due to the cut, we did not have to backtrack using the other div clauses.

In contrast, consider the state $\mathsf{minus}(\mathsf{s}(\mathsf{0}),Y,Z)$. Here, both clauses for minus are applicable and our state becomes $\mathsf{minus}(\mathsf{s}(\mathsf{0}),Y,Z)^7_1 \mid \mathsf{minus}(\mathsf{s}(\mathsf{0}),Y,Z)^8_1 \mid ?_1$. By using (7) we obtain $\square \mid \mathsf{minus}(\mathsf{s}(\mathsf{0}),Y,Z)^8_1 \mid ?_1$. Now, as we consider universal termination, we need to backtrack by removing the first element of our backtrack list and get $\mathsf{minus}(\mathsf{s}(\mathsf{0}),Y,Z)^8_1 \mid ?_1$ which we evaluate to $\mathsf{minus}(\mathsf{0},Y',Z) \mid ?_1$ using (8). Now, only the first clause for minus is applicable and we obtain $\mathsf{minus}(\mathsf{0},Y',Z)^7_2 \mid ?_2 \mid ?_1$ which we evaluate to $\square \mid ?_2 \mid ?_1$. Further backtracking leads first to $?_2 \mid ?_1$, then to $?_1$ and finally to the empty word ε where the computation stops.

The following definition formalizes the representation of such a concrete state.

Definition 3.8 (Concrete State)**.** *The set of* concrete states $\mathit{State}(\Sigma,\mathcal{V})$ *is the set of all finite words over* $\mathit{StateElements} = \mathit{Goal}(\Sigma,\mathcal{V}) \cup (\mathit{Goal}(\Sigma,\mathcal{V}) \times \mathbb{N} \times \mathbb{N}) \cup \mathbb{N}$.

Now, we connect goals to concrete states.

Definition 3.9 (Initial States from Goal)**.** *Given a goal* Q*, its representation as a concrete state is* $\mathit{Transformed}(Q,0)$.

The following example demonstrates the representation for our concrete states.

Example 3.10. Let $\cdot$ denote the composition of our backtracking lists. Now, consider again some of the states from Example 3.7.
A state consisting of just a goal (for instance $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)$) is represented by itself.
The state $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^3_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^4_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^5_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^6_1 \mid ?_1$, where we explicitly list all alternative clauses that might be applied to $\mathsf{div}(\mathsf{0},\mathsf{0},Z,R)$, is represented as:
$(\mathsf{div}(\mathsf{0},\mathsf{0},Z.R),3,1) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),4,1) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),5,1) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),6,1) \cdot 1$. Finally, the state $!_1, \mathsf{failure}(\mathsf{a}) \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^4_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^5_1 \mid \mathsf{div}(\mathsf{0},\mathsf{0},Z,R)^6_1 \mid ?_1$ is represented by:
$!_1, \mathsf{failure}(\mathsf{a}) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),4,1) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),5,1) \cdot (\mathsf{div}(\mathsf{0},\mathsf{0},Z,R),6,1) \cdot 1$.

For readability we will use the intuitive notation.

With the help of the representation of concrete states as defined in Definition 3.8 we can now express derivations in Prolog without built-in predicates other than ! and call by 10 simple inference rules.

Definition 3.11 (Concrete Inference Rules (cf. [Sch08]))**.**

$$\frac{\square \mid S}{S}\ (\textsc{Success}) \qquad \frac{?_m \mid S}{S}\ (\textsc{Failure}) \qquad \frac{\mathsf{call}(x), Q \mid S}{\varepsilon}\ (\textsc{VariableError})$$

$$\frac{t, Q \mid S}{\varepsilon}\ (\textsc{UndefinedError}) \quad \textit{where } \mathit{Slice}(\mathcal{P}, t) = \varnothing$$

$$\frac{!_m, Q \mid S \mid ?_m \mid S'}{Q \mid ?_m \mid S'}\ (\textsc{Cut}) \quad \begin{array}{l}\textit{where } S\\ \textit{contains}\\ \textit{no } ?_m\end{array} \qquad \frac{!_m, Q \mid S}{Q}\ (\textsc{CutAll}) \quad \begin{array}{l}\textit{where } S\\ \textit{contains}\\ \textit{no } ?_m\end{array}$$

$$\frac{t, Q \mid S}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S}\ (\textsc{Case}) \quad \begin{array}{l}\textit{where } m \in \mathbb{N} \textit{ is fresh, } i_1 < \ldots < i_k,\\ \textit{and } \mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\} \neq \varnothing\end{array}$$

$$\frac{(t, Q)^i_m \mid S}{B'_i\sigma, Q\sigma \mid S}\ (\textsc{Eval}) \quad \begin{array}{l}\textit{where } i \neq b,\ c_i = H_i \leftarrow B_i,\ \mathit{mgu}(t, H_i) = \sigma,\\ \textit{and } B'_i = \mathit{Transformed}(B_i, m)\end{array}$$

$$\frac{(t, Q)^i_m \mid S}{S}\ (\textsc{Backtrack}) \quad \textit{where } i \neq b,\ c_i = H_i \leftarrow B_i \textit{ and } t \not\sim H_i$$

$$\frac{\mathsf{call}(t'), Q \mid S}{t'', Q \mid ?_m \mid S}\ (\textsc{Call}) \quad \begin{array}{l}\textit{where } m \in \mathbb{N} \textit{ is fresh, } t' \in\\ \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V},\ t' \textit{ has only}\\ \textit{finitely many predication positions and}\\ t'' = \mathit{Transformed}(t', m)\end{array}$$

In the above rules we use the following conventions:

- The unlabeled term t must not be a variable, i.e., $t \notin \mathcal{V}$.
- The root symbol of t may not be a built-in predicate, i.e., $\mathit{root}(t) \notin \mathit{BuiltInPredicates}$.
- The list of terms Q may be $\square$ and then $t, Q = t, \square$ collapses to just t.
- S and S' denote concrete states.
- x represents a variable from $\mathcal{N}$.[4]

Note that these rules do not overlap, i.e., there is at most one rule that can be applied to any state. We call the subsequent application of these rules a *concrete state-derivation*.

[4]Since we do not have abstract variables in concrete states, we have $\mathcal{V} = \mathcal{N}$ here.

The only cases when no rule is applicable are when we call a term whose transformation according to the ISO standard would not terminate, since it has infinitely many predication positions[5] or when the state is the empty list (denoted ε).

Example 3.12. Consider the following Prolog program

$$\begin{aligned} \mathsf{p} &\leftarrow \mathsf{eq}(X, ,(\mathsf{q}, X)), \mathsf{call}(X). \\ \mathsf{eq}(X, X) &\leftarrow \square. \end{aligned}$$

and the query p. Even assuming we could already handle the built-in predicate ,/2, one might think that this query would lead to an error due to the undefined predicate q and, hence, terminate. In fact, the query does not terminate as the transformation of the infinite term does not terminate and we do not reach the call of the undefined predicate q.

Now, consider the next Prolog program

$$\begin{aligned} \mathsf{p} &\leftarrow \mathsf{eq}(X, \mathsf{call}(,(\mathsf{q}, X))), \mathsf{call}(X). \\ \mathsf{eq}(X, X) &\leftarrow \square. \end{aligned}$$

and the query p. This time, we reach the undefined predicate call and terminate with an error as the infinite term has not infinitely many predication positions and the transformation terminates.

We now describe the intuition behind the individual rules.

- Success: According to the ISO standard, a successful computation for a goal continues with backtracking. As $\square$ denotes the empty goal and, hence, a completely proved goal, this corresponds directly to a successful derivation. Thus, we have to backtrack using the Success rule. This corresponds to the analysis of universal termination. For analyzing existential termination we could modify this rule to yield the empty word, but this is not necessary as we will introduce a built-in predicate just restricting the computation to the first successful concrete state-derivation (cf. Chapter 4).

- Failure: Whenever we reach a question mark in the concrete state-derivation, there are no further backtracking possibilities for the current predicate call. Thus, we backtrack to the next predicate call before the current one or reach the empty state if no former predicate call exists.

[5] The ISO standard does not define any behavior for infinite rational terms explicitly, but if we use the transformation for finite terms given in the standard to transform terms which contain infinite paths along predication positions, this transformation would not terminate. This behavior is shown by real Prolog implementations such as SWI.

- VariableError: The call of an uninstantiated variable leads to an error in Prolog. Thus, the evaluation terminates directly and we can infer the empty state ε.

- UndefinedError: The call of an undefined predicate also leads to an error in Prolog.[6] Thus, the evaluation terminates directly and we can infer the empty state ε again.

- Case: While the execution model in the ISO standard performs the choice of suitable clauses for a predicate, the unification of the clause heads with the first current goal term, the replacement of this term with the clause body in case of a succeeding unification and the instantiation of this new goal with the mgu used for the unification simultaneously, we split these actions into separate rules. The Case rule determines the clauses in the program having the same function symbol as the first current goal term. Then it replaces this unlabeled goal with a number of labeled copies of it where each copy is labeled with the number of a corresponding clause in the order of the clauses in the program. Additionally, the new state elements are labeled with a fresh scope number to make the scope of possibly introduced cuts explicit. To mark the end of this scope, a question mark with the same scope number is inserted after the labeled copies of the replaced state element.

- Eval: The Eval rule corresponds to the second part of the execution model in case of a succeeding unification. It replaces the first current goal term with the corresponding clause body and applies the mgu of the unification with the clause head to the new goal.

- Backtrack: The Backtrack rule corresponds to failing unifications with clause heads. While the execution model would directly drop not unifying backtrack possibilities, we delay this dropping until we really reach the corresponding state element. Though this delay would not be necessary for concrete state-derivations, it is useful for handling classes of queries in the abstract state-derivations introduced in the next section.

- Cut: The Cut rule drops all state elements between the cut and the correspondingly labeled question mark. As described for the Case rule, this question mark is introduced at the end of the scope level of the corresponding cut. So all backtracking possibilities for predicates inside the scope of the cut which have not been

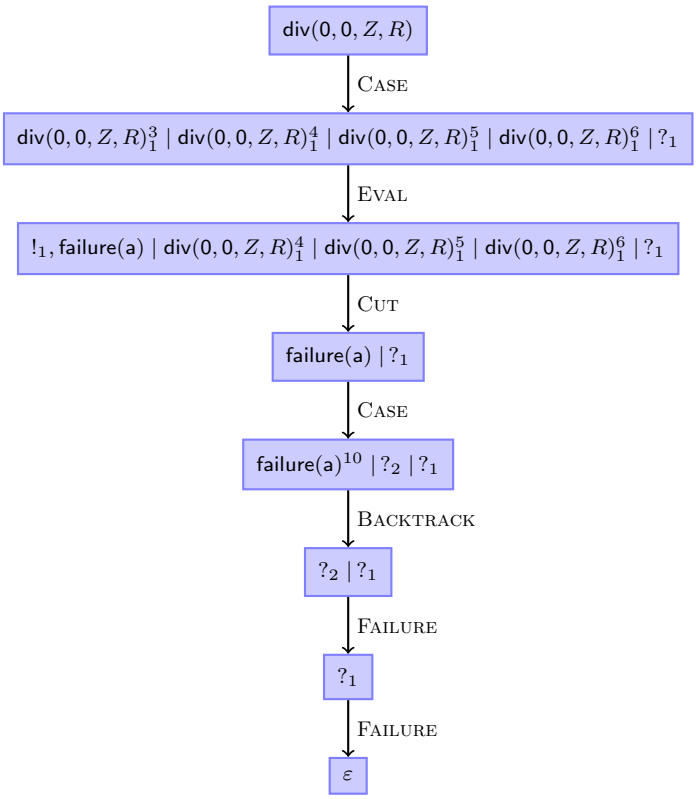
[6] This is at least true for most of the existing Prolog implementations in their default configuration. In fact, the behavior for undefined predicate calls depends on the flag unknown belonging to the environment of the Prolog processor as defined in [DEC96]. Since we do not consider such environment configurations or flags explicitly in this approach, we just handle this case according to the most common behavior observed for real Prolog implementations.

evaluated yet are discarded. Note that the question mark must not be deleted as the remaining goal can still have cuts with the same scope. Here, we profit from our more explicit representation of backtracking possibilities and can express the global side effect of a cut by a local rule.

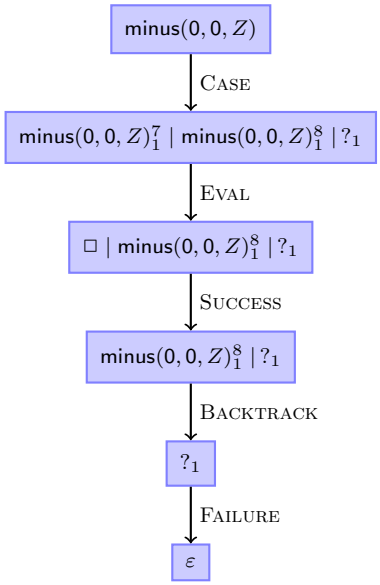
- CutAll: The CutAll rule would not be necessary for concrete state-derivations at all, but due to the splitting of states we have to perform in Section 3.3 to obtain a finite analysis, we may encounter the situation that there is no corresponding question mark for a labeled cut. Then all following backtracking possibilities must be discarded. This rule is equivalent to the Cut rule if the last state element is the corresponding question mark.

- Call: For meta-calls we use the Call rule. According to the ISO standard, meta-calls have to be transformed such that no variable is in a predication position. Additionally, cuts being part of the meta-call may not have any effect on the execution outside the meta-call. Both is handled by the *Transformed* function which replaces variables in predication positions by new meta-calls of these variables and labels cuts in predication positions with a new scope, thus restricting their effect to the meta-call. Note that we also solve the problem of unifying or comparing labeled cuts as we do not label cuts outside of predication positions.

Example 3.13. Consider again the logic program for div from Example 3.1 and the query $\mathsf{div}(0, 0, Z, R)$ from Example 3.7. Using our 10 inference rules we obtain the following tree.



Similar, for the query $\mathsf{minus}(0, 0, Z)$ from Example 3.7 we obtain the following state-derivation using the rules from Definition 3.11.



Finally, we define what it means for a state to be terminating.

Definition 3.14 (Termination of States [Sch08])**.** *We say that a given state $S \in State(\Sigma, \mathcal{V})$ is* terminating *if, and only if, there is no infinite concrete state-derivation starting from S using the inference rules from Definition 3.11.*

3.2 Abstract State-Derivations

After introducing concrete state-derivations for single queries, we continue by introducing *abstract state-derivations* for sets of queries as we want to analyze termination of a Prolog program w.r.t. a class of queries. In order to represent sets of queries, we introduce *abstract terms*, i.e., terms not only containing function symbols from Σ and usual variables as in Prolog programs, but also *abstract variables* which represent arbitrary but fixed terms. The set $\mathcal{A}$ is the set of all abstract variables, while the variables corresponding to variables in Prolog are from the set $\mathcal{N}$. The set $\mathcal{V}$ of all variables is therefore defined as $\mathcal{V} = \mathcal{N} \uplus \mathcal{A}$. Thus, $PrologTerms(\Sigma, \mathcal{V})$ is now the set of all abstract terms, while $PrologTerms(\Sigma, \mathcal{N})$ is the set of *concrete terms*, i.e., terms containing no abstract variables. Throughout the thesis, we often use the notation $\mathcal{N}(t)$ and $\mathcal{A}(t)$ to denote the set of all non-abstract and abstract variables occurring in a term t as defined in Definition 2.2, respectively. In many situations we will consider substitutions which are equal on a certain set of variables, while they do not replace any other variables. We call such substitutions *restricted* to a certain set. The restriction of σ to a set of variables $\mathcal{V}' \subseteq \mathcal{V}$ (denoted $\sigma|_{\mathcal{V}'}$) is therefore defined as $\sigma|_{\mathcal{V}'}(X) = \sigma(X)$, if $X \in \mathcal{V}'$, and $\sigma|_{\mathcal{V}'}(X) = X$, otherwise. Finally, we often need variables which do not occur anywhere else. We call such variables *fresh variables*

and denote by $\mathcal{V}_{fresh} \subseteq \mathcal{V}$ the subset of fresh variables. Analogously, we denote the subset of fresh abstract and non-abstract variables by $\mathcal{A}_{fresh}$ and $\mathcal{N}_{fresh}$, respectively.

Abstract variables represent arbitrary terms in general, but to describe classes of queries typically specified by a function symbol and argument positions which should be instantiated by ground terms, we need to constrain the terms by which the abstract variables may be instantiated. Additionally, we want to keep track of non-abstract variables which do not occur in the terms represented by abstract variables. Finally, due to failing unifications during the concrete state-derivation, we gather knowledge about non-unifiable terms. Therefore, we add a *knowledge base* representable by a triple $KB = (\mathcal{G}, \mathcal{F}, \mathcal{U})$ to a concrete state containing abstract terms where $\mathcal{G} \subseteq \mathcal{A}$, $\mathcal{F} \subseteq \mathcal{N}$, and $\mathcal{U} \subseteq \mathit{PrologTerms}(\Sigma, \mathcal{V}) \times \mathit{PrologTerms}(\Sigma, \mathcal{V})$. Here, $\mathcal{G}$ is the set of all abstract variables whose instantiations are restricted to ground terms, while $\mathcal{F}$ contains those non-abstract variables which may not occur in the terms represented by abstract variables. Moreover, $\mathcal{U}$ represents a set of pairs of terms, where a pair of terms (s, t) represents that s and t are not unifiable after instantiating the abstract variables, i.e., that we have $s\gamma \nsim t\gamma$ for a given instantiation γ of the abstract variables. We can now define an abstract state based on a concrete state with abstract variables and a knowledge base.

Definition 3.15 (Abstract State)**.** *The set of* abstract states $\mathit{AState}(\Sigma, \mathcal{N}, \mathcal{A})$ *is a set of pairs* $(s; KB)$ *of a concrete state* $s \in \mathit{State}(\Sigma, \mathcal{N} \cup \mathcal{A})$ *and a* knowledge base KB*.*

To define which concrete states are represented by an abstract state, we introduce the notion of a *concretization*. A concretization is a substitution γ replacing all and only abstract variables and which respects the knowledge base $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. Thus, we have $\gamma|_{\mathcal{A}} = \gamma$ and $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$. Also, abstract variables from $\mathcal{G}$ are only replaced by ground terms, i.e., $\mathit{Range}(\gamma|_{\mathcal{G}}) \subseteq \mathit{GroundTerms}(\Sigma)$. Likewise, γ may not introduce variables from $\mathcal{F}$. This can be specified by $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$. Finally, for all pairs $(t, t') \in \mathcal{U}$ we need to ensure that $t\gamma$ and $t'\gamma$ do not unify, i.e., that $t\gamma \nsim t'\gamma$. Taking everything into account, we obtain the following definition.

Definition 3.16 (Concretization)**.** *A substitution* γ *is a* concretization *w.r.t. a knowledge base* $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ *if, and only if,* $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$, $\mathit{Range}(\gamma|_{\mathcal{G}}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$, *and* $\bigwedge_{(t,t') \in \mathcal{U}} t\gamma \not\sim t'\gamma$.

For an abstract state $(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$*, we define the set of* concretized states $\mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ *as the set* $\{S\gamma \mid \gamma$ *is a concretization w.r.t. the knowledge base* $(\mathcal{G}, \mathcal{F}, \mathcal{U})\}$.

Example 3.17. Consider the abstract state $\mathsf{minus}(T_1, T_2, T_3); (\{T_1, T_2\}, \varnothing, \{(T_1, T_3)\})$ with $T_i \in \mathcal{A}$ for all i. This represents all concrete states $\mathsf{minus}(t_1, t_2, t_3)$ where t_1, t_2 are ground terms and t_1 and t_3 do not unify, i.e., t_3 does not match t_1. For example, the concrete state $\mathsf{minus}(\mathsf{0}, \mathsf{0}, Z)$ is not represented as $\mathsf{0}$ and Z unify. In contrast, the concrete state

$\mathsf{minus}(\mathsf{s}(0), \mathsf{s}(0), 0)$ is represented and, using clause (8), can be reduced to $\mathsf{minus}(0, 0, 0)$. But this clause cannot be applied to all concretizations. Consider e.g. the concrete state $\mathsf{minus}(0, 0, \mathsf{s}(0))$ represented by our abstract state, for which no clause is applicable.

Finally, we connect classes of queries w.r.t. a Prolog program, as typically specified by the user, to abstract states.

Definition 3.18 (Initial State for Predicate and Moding Function)**.** *Given a Prolog program $\mathcal{P}$ and a class of queries $\mathcal{Q}$ described by a symbol $p \in \Sigma$ with arity n and a moding function $m : \Sigma \times \mathbb{N} \to \{\boldsymbol{in}, \boldsymbol{out}\}$, the initial state $S(p, m)$ for $\mathcal{P}$ w.r.t. $\mathcal{Q}$ is defined as $p(T_1, \ldots, T_n); (\mathcal{G}, \varnothing, \varnothing)$ where $\mathcal{G} = \{T_i \mid m(p, i) = \boldsymbol{in}\}$.*

Example 3.19. Consider the Prolog program divremain.pl from Example 3.1 as $\mathcal{P}$ and the class of queries $\mathcal{Q}$ described by the symbol $\mathsf{div}/4$ and the moding function m with $m(\mathsf{div}, 1) = m(\mathsf{div}, 2) = \mathbf{in}$ and $m(f, i) = \mathbf{out}$ otherwise. Then the initial state $S(\mathsf{div}, m)$ is $\mathsf{div}(T_1, T_2, T_3, T_4); (\{T_1, T_2\}, \varnothing, \varnothing)$.

As we have seen in the example above, abstract states may represent several concrete states for which different concrete inference rules are applicable. Additionally, we also want to exploit the knowledge specified by the knowledge base of an abstract state.

As we are considering classes of queries and, thus, sets of concrete states represented by abstract states as defined by Definition 3.15, the general idea of the following abstract rules is that all states represented by the parent node are terminating if all the states represented by its children are terminating.

Definition 3.20 (Sound Rules [Sch08])**.** *A rule $\rho : AState(\Sigma, \mathcal{N}, \mathcal{A}) \to 2^{AState(\Sigma, \mathcal{N}, \mathcal{A})}$ is a* sound rule *if for all abstract states $(S; KB)$, all $S\gamma \in \mathcal{CON}(S; KB)$ are terminating if all states in $\{R\gamma' \mid (R; KB') \in \rho(S; KB), R\gamma' \in \mathcal{CON}(R; KB')\}$ are terminating.*

The rules for Success, Failure, VariableError, UndefinedError, Cut, CutAll, Case and Call do not mandate changes to the knowledge base and are, thus, straightforward to adapt to the abstract case.

Therefore, we define the first set of abstract rules corresponding to the first part of the original rules from Definition 3.11. We will handle the Backtrack and Eval rules in later definitions as the same abstract state may represent concrete state where Eval is applicable as well as concrete states where Backtrack is applicable. We will even introduce additional rules used to obtain a finite analysis in spite of the fact that we have no bound on the size of the terms represented by the abstract variables.

Definition 3.21 (Abstract Inference Rules – Part 1 (Success, Failure, VariableError, UndefinedError, Cut, CutAll, Case, Call) (cf. [Sch08]))**.**

$$\frac{\square \mid S; KB}{S; KB}\ (\textsc{Success}) \qquad \frac{?_m \mid S; KB}{S; KB}\ (\textsc{Failure}) \qquad \frac{\mathsf{call}(x), Q \mid S; KB}{\varepsilon; KB}\ (\textsc{VariableError})$$

$$\frac{t, Q \mid S; KB}{\varepsilon; KB}\ (\textsc{UndefinedError}) \quad \textit{where } \mathit{Slice}(\mathcal{P}, t) = \varnothing$$

$$\frac{!_m, Q \mid S \mid ?_m \mid S'; KB}{Q \mid ?_m \mid S'; KB}\ (\textsc{Cut}) \ \begin{array}{l}\textit{where } S\\ \textit{contains}\\ \textit{no } ?_m\end{array} \qquad \frac{!_m, Q \mid S; KB}{Q; KB}\ (\textsc{CutAll}) \ \begin{array}{l}\textit{where } S\\ \textit{contains}\\ \textit{no } ?_m\end{array}$$

$$\frac{t, Q \mid S; KB}{(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S; KB}\ (\textsc{Case}) \quad \begin{array}{l}\textit{where } m \in \mathbb{N} \textit{ is fresh, } i_1 < \ldots < i_k,\\ \textit{and } \mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\} \neq \varnothing\end{array}$$

$$\frac{\mathsf{call}(t'), Q \mid S; KB}{t'', Q \mid ?_m \mid S; KB}\ (\textsc{Call})$$

where $m \in \mathbb{N}$ *is fresh,* $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$*,* t' *has only finitely many predication positions,* $\forall \mathit{pos} \in \mathit{Occ}(t') : \mathit{pos}$ *is a predication position* $\implies t'|_{\mathit{pos}} \notin \mathcal{A}$ *and* $t'' = \mathit{Transformed}(t', m)$

In the above rules, in addition to the notation used in Definition 3.11, we write $S \mid S'; KB$ for an abstract state $((S \mid S'); KB)$ with knowledge base KB.

Note that we are stuck in the case of meta-calls of terms where we cannot guarantee the termination of the call itself. This is the case for terms having infinitely many predication positions as for the concrete inference rules. Also, we are stuck in the case of meta-calls where we already know that we cannot prove termination of the term to execute. This is the case for terms with an abstract variable as predication w.r.t. the term. This is due to the fact that abstract variables stand for arbitrary terms. Even for ground terms we cannot guarantee that their execution is terminating. Hence, we are not able to prove termination of the given program successfully with the approach presented here. As this approach is not termination preserving (see Chapter 7), we are not able to prove non-termination of the given program, either. So there is nothing we can do and we just give up.[7]

[7] One might make some assumptions concerning the terms represented by abstract variables which are used as predications in meta-calls to overcome such situations. But as the halting problem is not decidable, one will hardly find any suitable assumptions ensuring termination of the meta-calls, being automatically and efficiently verifiable for input arguments and still allowing for a realistic and interesting class of queries. Thus, we refrain from making such assumptions here.

Since the knowledge bases are not changed in the rules Success, Failure, Cut, CutAll and Case, the proofs for their soundness in [Sch08] remain unchanged and the rules are already shown to be sound.

Lemma 3.22 (Soundness of VariableError, UndefinedError and Call)**.** *The rules* VariableError*,* UndefinedError *and* Call *from Definition 3.21 are sound.*

Proof. For the soundness of VariableError assume there is an infinite concrete state-derivation from $\mathsf{call}(x)\gamma, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{call}(x), Q \mid S; KB)$. As γ is only defined for abstract variables, the only applicable concrete rule is VariableError, which results in ε. Since no concrete rule is applicable to the empty list, we have a contradiction to our assumption and VariableError is trivially sound.

For the soundness of UndefinedError assume there is an infinite concrete state-derivation from $t\gamma, Q\gamma \mid S\gamma \in \mathcal{CON}(t, Q \mid S; KB)$. As γ does not change the root symbol of t, we have $\mathit{Slice}(\mathcal{P}, t\gamma) = \varnothing$ and the only applicable concrete rule is UndefinedError leading to the state ε which is again a contradiction to our assumption. Thus, UndefinedError is trivially sound, too.

For the soundness of Call assume there is an infinite concrete state-derivation from $\mathsf{call}(t')\gamma, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{call}(t'), Q \mid S; KB)$ where there are only finitely many predication positions in $\mathit{Occ}(t')$ and no predication w.r.t. t' is an abstract variable. As γ is only defined for abstract variables, we obtain that $t'\gamma$ has only finitely many predication positions. Hence, the only applicable concrete inference rule is Call leading to the state $\mathit{Transformed}(t'\gamma, m), Q\gamma \mid ?_m \mid S\gamma$ for a fresh $m \in \mathbb{N}$ starting an infinite concrete state-derivation. By the fact that no predication position w.r.t. t' is an abstract variable and Definition 2.17 we obtain $\mathit{Transformed}(t'\gamma, m) = \mathit{Transformed}(t', m)\gamma = t''\gamma$ and, thus, $\mathit{Transformed}(t'\gamma, m), Q\gamma \mid ?_m \mid S\gamma = t''\gamma, Q\gamma \mid ?_m \mid S\gamma \in \mathcal{CON}(t'', Q \mid ?_m \mid S; KB)$ having an infinite concrete state-derivation. □

So far, like for the concrete rules, the applicable rule is uniquely determined by the first state element of the abstract state. Now, for the concrete Eval and Backtrack rule we determine which of these two rules is applicable by trying to unify the first term in the first state element with the head of the corresponding clause. As demonstrated by Example 3.17, we might need to apply Eval for some concrete states represented by an abstract state and Backtrack for others. Assume we are in a state $(t, Q)^i_m$ and the i-th clause is $H_i \leftarrow B_i$. Consider that the abstract variables represent arbitrary but fixed terms. Thus, if H_i and t unify by replacing abstract variables by other abstract variables or non-variable terms, this might or might not succeed.

We can detect that the concrete Backtrack rule is applicable for all concretized states if either the abstract term t does not unify with the clause head H_i or if $\mathit{mgu}(t, H_i) = \sigma$, but σ contradicts information in $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we can apply the following abstract Backtrack rule.

Unfortunately, for the definition of the abstract BACKTRACK rule in [Sch08] it is unclear how to implement the check for non-unifiability. Here we present a modified version of the BACKTRACK rule which is efficiently implementable and still sound.

The first condition $t \nsim H_i$ is already sufficient for BACKTRACK. If otherwise $mgu(t, H_i) = \sigma$, then σ contradicts information in $\mathcal{G}$ if we replace a variable from $\mathcal{G}$ by an infinite term. Furthermore, σ contradicts information in $\mathcal{U}$ if there is a pair $(s, s') \in \mathcal{U}$ such that $s\sigma$ and $s'\sigma$ unify by only replacing free variables, i.e., variables from $\mathcal{F}$.

Definition 3.23 (Abstract Inference Rules – Part 2 (BACKTRACK))**.**

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U})} \text{ (BACKTRACK)}$$

where $i \neq b$, $c_i = H_i \leftarrow B_i$ *and either* $t \nsim H_i$ *or* $\sigma = mgu(t, H_i)$ *with* $\exists a \in \mathcal{G} : a\sigma \notin FiniteTermsOf(\Sigma, \mathcal{V})$ *or* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{G}})) \subseteq \mathcal{A}$ *and* $\exists (s, s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge Dom(\sigma') \subseteq \mathcal{N} \wedge \sigma'\sigma' = \sigma'$.

Lemma 3.24 (Soundness of BACKTRACK)**.** *The rule* BACKTRACK *from Definition 3.23 is sound.*

Proof. Assume there is an infinite concrete state-derivation from $(t\gamma, Q\gamma)^i_m \mid S\gamma \in \mathcal{CON}((t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$.

Let γ be a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. If $t\gamma \nsim H_i$ then the only applicable concrete rule is BACKTRACK, which results in $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ starting an infinite concrete state-derivation.

Thus, we are left to show that there is no concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ with $t\gamma \sim H_i$.

If $t \nsim H_i$, then there is no substitution δ with $t\delta \sim H_i\delta$. By definition we have that $\mathcal{V}(t) \cap \mathcal{V}(H_i) = \varnothing$. Thus we obtain that there is no substitution δ with $t\delta \sim H_i$, either. In particular, there is no concretization γ with $t\gamma \sim H_i$.

So let $\sigma = mgu(t, H_i)$. If there is a variable $a \in \mathcal{G}$ with $a\sigma \notin FiniteTermsOf(\Sigma, \mathcal{V})$, there is no concretization γ with $t\gamma \sim H_i$. To see this, assume there is an mgu σ'''' of $t\gamma$ and $H_i = H_i\gamma$. Then there must be a substitution δ' with $\sigma\delta' = \gamma\sigma''''$. We can assume $a \in \mathcal{V}(t)$, since $\mathcal{G}(H_i) = \varnothing$ and σ is most general. As $a\sigma \notin FiniteTermsOf(\Sigma, \mathcal{V})$ we also have $a\sigma\delta' = a\gamma\sigma'''' \notin FiniteTermsOf(\Sigma, \mathcal{V})$. Since γ has to replace all abstract variables in t and $\mathcal{A}(Range(\gamma)) = \varnothing$, we obtain $a\gamma \notin FiniteTermsOf(\Sigma, \mathcal{V}) \supseteq GroundTermsOf(\Sigma)$. Contradiction.

Now let $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$ and $\exists (s, s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge Dom(\sigma') \subseteq \mathcal{N} \wedge \sigma'\sigma' = \sigma'$.

We show that $\forall \delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$ by showing that $\sigma'\sigma'\delta$ is a unifier of $s\sigma|_{\mathcal{G}}\delta$ and $s'\sigma|_{\mathcal{G}}\delta$. Let δ be a substitution with

$Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}$. We immediately have that $\delta = \delta\delta$ since $Dom(\delta) \cap \mathcal{V}(Range(\delta)) = \varnothing$. Let $x \in \mathcal{V}$. If $x \in \mathcal{N}$, we obviously have $x\delta\sigma'\sigma'\delta = x\sigma'\delta = x\sigma'\delta\sigma'$. If otherwise $x \in \mathcal{A}$, we have $x\delta\sigma'\sigma'\delta = x\delta\sigma' = x\sigma'\delta\sigma'$. Thus, we have $\delta\sigma'\sigma'\delta = \sigma'\delta\sigma'$. Now we obtain:

$$\begin{aligned} s\sigma|_{\mathcal{G}}\delta\sigma'\sigma'\delta &\overset{\delta\sigma'\sigma'\delta=\sigma'\delta\sigma'}{=} s\sigma|_{\mathcal{G}}\sigma'\delta\sigma' \\ &\overset{\sigma'=mgu(s|_{\mathcal{G}},s'|_{\mathcal{G}})}{=} s'\sigma|_{\mathcal{G}}\sigma'\delta\sigma' \\ &\overset{\delta\sigma'\sigma'\delta=\sigma'\delta\sigma'}{=} s'\sigma|_{\mathcal{G}}\delta\sigma'\sigma'\delta \end{aligned}$$

From $\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$ it follows that $\forall\delta\exists(s,s') \in \mathcal{U} : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$. By disjunctively adding $t\delta \nsim H_i$ we obtain:

$$\forall\delta\exists(s,s') \in \mathcal{U} : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta \vee t\delta \nsim H_i$$

$$\overset{Dom(\delta)\subseteq\mathcal{A}\wedge\mathcal{V}(Range(\delta))\subseteq\mathcal{N}\setminus\mathcal{F} \text{ independent from } s,s',t, \text{ and } H_i}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies (\exists(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta) \vee t\delta \nsim H_i$$

$$\overset{\text{Double negation}}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies \neg(\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta) \vee t\delta \nsim H_i$$

$$\overset{\text{Definition of implication}}{\iff}$$

$$\forall\delta : \neg((Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F})) \vee \neg(\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta) \vee t\delta \nsim H_i$$

$$\overset{\text{Factor out negation}}{\iff}$$

$$\forall\delta : \neg(Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta)) \vee t\delta \nsim H_i$$

$$\overset{\text{Definition of implication}}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta)) \implies t\delta \nsim H_i$$

We continue by showing that for all concretizations γ we have $t\gamma \nsim H_i$.
Let γ be a concretization with $\gamma = \sigma|_{\mathcal{G}}\gamma'$ where γ' is an arbitrary substitution. Then we have $\gamma\sigma|_{\mathcal{G}} = \sigma|_{\mathcal{G}}\gamma$. By definition, γ satisfies $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\gamma \nsim s'\gamma)$. Since $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$ we also have $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\gamma\sigma|_{\mathcal{G}} \nsim s'\gamma\sigma|_{\mathcal{G}})$ and hence $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\gamma \nsim s'\sigma|_{\mathcal{G}}\gamma)$ which implies $t\gamma \nsim H_i$.
Now let γ be a concretization with $\gamma \neq \sigma|_{\mathcal{G}}\gamma'$ for all substitutions γ'. Assume $t\gamma \sim H_i$.

Then we would also have $t\gamma \sim H_i\gamma$ because $\mathcal{A}(H_i) = \varnothing$. So there is a substitution σ'' with $t\gamma\sigma'' = H_i\gamma\sigma''$. Since $mgu(t, H_i) = \sigma$ and $\sigma = \sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}$ we obtain $\exists\sigma''' : \sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''' = \gamma\sigma''$. As we have $Dom(\gamma) \subseteq \mathcal{A}$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we also have that $(\sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''')|_{\mathcal{N}} = (\sigma|_{\mathcal{N}}\sigma''')|_{\mathcal{N}} = \sigma''$ and $(\sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''')|_{\mathcal{G}} = (\sigma|_{\mathcal{G}}\sigma''')|_{\mathcal{G}} = \gamma|_{\mathcal{G}}$.
Let $x \in \mathcal{V}$ be any variable. We perform a case analysis over the partition $(Range(\sigma|_{\mathcal{G}}) \uplus Dom(\sigma|_{\mathcal{G}}) \uplus (\mathcal{V} \setminus (Dom(\sigma|_{\mathcal{G}}) \cup Range(\sigma|_{\mathcal{G}}))))$:

- $x \in (\mathcal{V} \setminus (Dom(\sigma|_{\mathcal{G}}) \cup Range(\sigma|_{\mathcal{G}})))$:
  We define $x\gamma' = x\gamma$ and have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$.

- $x \in Range(\sigma|_{\mathcal{G}})$:
  Then there is a variable $a \in Dom(\sigma|_{\mathcal{G}})$ and a position π with $(a\sigma|_{\mathcal{G}})|_\pi = x$. Additionally we know that $x \in \mathcal{A}$ and $a\sigma|_{\mathcal{G}}\sigma''' = a\gamma$. Hence we have $x\sigma''' = a\gamma|_\pi$. W.l.o.g. we demand $x\gamma = x\sigma'''$ since x is fresh. So we define $x\gamma' = x\sigma'''$. Then we have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$.

- $x \in Dom(\sigma|_{\mathcal{G}})$:
  We define $x\gamma' = x$ as x will already be replaced by $\sigma|_{\mathcal{G}}$. So we still have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$ since all variables in $Range(\sigma|_{\mathcal{G}})$ are properly replaced.

So we have in all cases $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$ and, therefore, $\sigma|_{\mathcal{G}}\gamma' = \gamma$. Contradiction. Thus, we have $t\gamma \nsim H_i$ again. □

In case the abstract Backtrack rule is not applicable, it is still possible that $t\gamma$ does not unify with H_i for a concretization γ. This was already demonstrated by Example 3.17. Thus, the Eval rule for abstract states has two successor states combining both the concrete Eval and the concrete Backtrack rule.

Now, one might argue that the abstract Backtrack rule is superfluous as we can represent all concrete state-derivations using the Eval rule alone. For the correctness of our method this is, in fact, right. But as the abstract states might represent only concrete states where the concrete Backtrack rule is applicable, we would significantly lose precision by using the abstract Eval rule in such cases. The reason for this is, that we would add a state representing concrete states which would not be reachable by the concrete state-derivations from any concrete state represented by the parent state. Hence, without the abstract Backtrack rule our method is less powerful. Another problem is that the left successor is not defined at all if t and H_i do not unify since we have no mgu then.

Example 3.25. Consider the following simple Prolog program $\mathcal{P}$

$$\begin{aligned} \mathsf{p}(0) &\leftarrow !. \\ \mathsf{p}(X) &\leftarrow \mathsf{q}(X). \\ \mathsf{q}(0) &\leftarrow \mathsf{q}(0). \\ \mathsf{q}(X) &\leftarrow \square. \end{aligned}$$

and the set of queries $\mathcal{Q} = \{\mathsf{p}(t) \mid \mathsf{p}(t) \in \mathit{PrologTerms}(\Sigma, \mathcal{N})\}$. Without Backtrack, there would always be a successor node resulting from Eval with the first clause for q. Thus, although the program clearly terminates for $\mathcal{Q}$, we could not prove termination of this program since we would introduce states which are in fact not reachable for those queries. Using Backtrack, we find out that the first clause for q is never applicable for queries from $\mathcal{Q}$ and we can easily prove termination of $\mathcal{P}$ w.r.t. $\mathcal{Q}$.

Before we motivate the definition of the abstract Eval rule, we introduce a substitution used for refreshing variables from a certain set. Such substitutions will for example be used to simulate the sharing effects by replacing abstract variables which are not known to represent ground terms only.

Definition 3.26 (Replacement by Fresh Abstract Variables)**.** *We define $\alpha_{\mathcal{M}}$ for a set of variables $\mathcal{M}$ as follows:*

$$\alpha_{\mathcal{M}}(x) = \begin{cases} a & \text{if } x \in \mathcal{M} \setminus \mathcal{V}_{\mathit{fresh}} \text{ for } a \in \mathcal{A}_{\mathit{fresh}} \\ x & \text{otherwise} \end{cases}$$

Now we give some intuition for the (rather complex) definition of the abstract Eval rule.

To avoid name conflicts and to obtain an idempotent substitution we demand that the range of the most general unifier σ of t and H_i contains only fresh variables. Thus, the condition on σ is $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{V}_{\mathit{fresh}}$.

The next condition ensures that σ instantiates an abstract variable a only with terms over Σ and $\mathcal{A}$ (instead of $\mathcal{V}$). This condition is necessary to retain the generality of the abstract state. Without the condition $\mathcal{V}(\mathit{Range}(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, one might for example replace the abstract variable T_1 which represents $\mathsf{s}(0)$ by $\mathsf{s}(X)$ where $X \in \mathcal{N}$. But then we would not represent $\mathsf{s}(0)$ anymore. By replacing T_1 with $\mathsf{s}(T_2)$ instead where $T_2 \in \mathcal{A}$, the term $\mathsf{s}(0)$ is still represented.

Furthermore, we do not need to introduce new abstract variables other than those we introduce for the already existing abstract variables. The reason for this is that non-abstract variables just represent themselves and not arbitrary terms where variables inside the terms may be replaced due to sharing effects. So we constrain the range of $\sigma|_{\mathcal{N}}$ by

$\mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{A}}))$.

Note that all these restrictions are without loss of generality as they can be obtained by a simple variable renaming from any most general unifier of t and H_i.

Once we have an mgu satisfying the above conditions, we must update the knowledge base accordingly. If we replace abstract variables from $\mathcal{G}$, all variables in the replaced term must be abstract and represent ground terms. Otherwise we would suddenly represent non-ground terms at the position of a ground term. Thus, the new set $\mathcal{G}'$ of abstract variables representing ground terms is $\mathcal{G} \cup \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$. As all variables in $H_i \leftarrow B_i$ are freshly renamed, the ones not occurring in H_i must be free and cannot occur in the terms represented by abstract variables. Hence, we add $\mathcal{N}(B_i) \setminus \mathcal{N}(H_i)$ to $\mathcal{F}$. Moreover, variables in the range of $\sigma|_{\mathcal{F}}$ not being in the range of any variable from $\mathcal{V} \setminus \mathcal{F}$ or the head H_i cannot occur in the represented terms of abstract variables, either. Therefore, we also add $\mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(\mathit{Range}(\sigma|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}_{H_i})}))$ to $\mathcal{F}$. Together, this yields the new set of free variables $\mathcal{F}'$. For the successor state corresponding to the application of the concrete Backtrack rule, we update $\mathcal{U}$ by adding the non-unifying pair (t, H_i) as the concrete Backtrack rule is only applicable if these terms do not unify. For the successor state corresponding to the application of the concrete Eval rule, we apply $\sigma|_{\mathcal{G}}$ to the terms in $\mathcal{U}$ and in all state elements following the first one. We can do this as the abstract variables in $\mathcal{G}$ represent terms without any variables. Thus, the replacement of abstract variables from $\mathcal{G}$ corresponds to a kind of shape analysis instead of instantiating variables from $\mathcal{N}$ inside the represented terms. The latter would not be correct for backtracking goals as the instantiations are canceled for backtracking. Therefore, it would not be correct for the terms in $\mathcal{U}$ either, since we keep the same knowledge base for backtracked state elements. For other abstract variables than the ones from $\mathcal{G}$, we cannot distinguish between the replacements due to shape analysis and instantiation of non-abstract variables that easily, since the abstract variables may also represent non-abstract variables.[8]

Concerning sharing effects, we still have a problem if we replace variables not being from $\mathcal{G} \cup \mathcal{F}$. Such variables may occur in the terms represented by abstract variables. Thus, by replacing them, we would change the represented terms. Additionally, by replacing abstract variables not from $\mathcal{G}$, we might instantiate non-abstract variables in the represented terms which occur elsewhere in the state. To overcome this problem, we approximate such instantiations due to variable sharing by replacing variables whose represented terms might have changed or which might be instantiated due to sharing effects by fresh abstract variables. If σ replaces non-abstract variables not being from $\mathcal{F}$, we

[8] An idea to increase precision concerning shape analysis would be to first perform a case analysis for the possibly represented variables and extending the knowledge base to also have a set of abstract variables not representing non-abstract variables (but still terms containing such variables). Thus, we could infer the root symbol of the terms represented by such abstract variables. We could especially profit from this idea in combination with the built-in predicates for type testing (cf. Chapter 4). For now, we leave a more sophisticated shape analysis to future work.

have to replace all abstract variables not being from $\mathcal{G}$ as their represented terms might have changed. If we replace abstract variables not from $\mathcal{G}$ we must additionally replace all non-abstract variables not being from $\mathcal{F}$ as they might be instantiated due to sharing effects. The respective approximation is performed by the *Approx* function.

As for the concrete rules, the successors are updated by replacing the first term in the first state element by the transformed and instantiated clause body or by dropping the first state element respectively.

In case the unification of the abstract term t with the head H_i of the clause will definitely succeed, i.e., the mgu σ is a bijection on abstract variables (and hence just a variable renaming on abstract variables), we do not have to backtrack, of course. Thus, in such cases we can spare the successor corresponding to the application of the concrete Backtrack rule. We introduce the OnlyEval rule for such cases, having the condition that $\sigma|_{\mathcal{A}}$ is a variable renaming on $\mathcal{A}$. To obtain non-overlapping rules, the conditions of the Backtrack and OnlyEval rules are negated for the Eval rule. This amounts to $Range(\sigma|_{\mathcal{G}}) \subseteq FinitePrologTerms(\Sigma, \mathcal{A})$, $\sigma|_{\mathcal{A}}$ is not a variable renaming on $\mathcal{A}$ and $\forall (s, s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_{\mathcal{G}}\sigma'' = s'\sigma|_{\mathcal{G}}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{N})$.

Definition 3.27 (Abstract Inference Rules – Part 3 (Eval, OnlyEval))**.**

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{B'_i\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}}) \qquad S; (\mathcal{G}, \mathcal{F} \cup \mathcal{N}(H_i), \mathcal{U} \cup \{(t, H_i)\})} \text{ (Eval)}$$

where $i \neq b$, $c_i = H_i \leftarrow B_i$, $mgu(t, H_i) = \sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, $Range(\sigma|_{\mathcal{G}}) \subseteq FinitePrologTerms(\Sigma, \mathcal{A})$, $\sigma|_{\mathcal{A}}$ *is not a variable renaming on* $\mathcal{A}$, $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\forall (s, s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_{\mathcal{G}}\sigma'' = s'\sigma|_{\mathcal{G}}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{N})$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}}))\backslash\mathcal{N}(Range(\sigma|_{\mathcal{N}\backslash(\mathcal{F}\cup\mathcal{N}(H_i))})))\cup(\mathcal{N}(B_i)\backslash\mathcal{N}(H_i))$, $\sigma' = Approx(\sigma, t, c_i, \mathcal{G}, \mathcal{F})$, *and* $B'_i = Transformed(B_i, m)$.

$$\frac{(t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{B'_i\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})} \text{ (OnlyEval)}$$

where $i \neq b$, $c_i = H_i \leftarrow B_i$, $mgu(t, H_i) = \sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\sigma|_{\mathcal{A}}$ *is a variable renaming on* $\mathcal{A}$, $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}}))\backslash\mathcal{N}(Range(\sigma|_{\mathcal{N}\backslash(\mathcal{F}\cup\mathcal{N}(H_i))})))\cup(\mathcal{N}(B_i)\backslash\mathcal{N}(H_i))$, $\sigma' = Approx(\sigma, t, c_i, \mathcal{G}, \mathcal{F})$, *and* $B'_i = Transformed(B_i, m)$.

Approx replaces some variables by fresh abstract variables:

$$Approx(\sigma, t, H_i \leftarrow B_i, \mathcal{G}, \mathcal{F}) = \begin{cases} \sigma & \text{if } \mathcal{A}(t) \subseteq \mathcal{G} \text{ and } \mathcal{N}(t) \subseteq \mathcal{F} \\ \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'} & \text{if } \mathcal{A}(t) \subseteq \mathcal{G} \text{ and } \mathcal{N}(t) \not\subseteq \mathcal{F} \\ \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} & \text{if } \mathcal{A}(t) \not\subseteq \mathcal{G} \end{cases}$$

Lemma 3.28 (Soundness of EVAL)**.** *The rule* EVAL *from Definition 3.27 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ with $mgu(t\gamma, H_i) = \sigma''$ there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $\gamma\sigma'' = \sigma'\gamma'$, $\gamma = \sigma|_{\mathcal{G}}\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.*

Proof. Assume $(t\gamma, Q\gamma)_m^i \mid S\gamma \in \mathcal{CON}(t, Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U}))$ has an infinite concrete state-derivation. There are two cases depending on whether $t\gamma$ and H_i unify.

First, if $t\gamma$ does not unify with H_i, then the only applicable concrete rule is BACKTRACK, which results in $S\gamma$ starting an infinite concrete state-derivation. From $t\gamma \not\sim H_i$ and $\mathcal{A}(H_i) = \varnothing$ we know $H_i\gamma = H_i$ and, therefore, $t\gamma \not\sim H_i\gamma$. As all variables in H_i are fresh, we also have $\mathcal{N}(Range(\gamma)) \cap \mathcal{N}(H_i) = \varnothing$. Thus, γ is also a concretization w.r.t. $(\mathcal{G},\mathcal{F}\cup\mathcal{N}(H_i),\mathcal{U}\cup\{(t;H_i)\})$ and $S\gamma \in \mathcal{CON}(S;(\mathcal{G},\mathcal{F}\cup\mathcal{N}(H_i),\mathcal{U}\cup\{(t;H_i)\}))$.

Second, if $t\gamma \sim H_i$, we follow the proof from [Sch08] and find that the unique applicable concrete rule is EVAL. From $H_i\gamma = H_i$ we know that $t\gamma \sim H_i\gamma$ and thus t also unifies with H_i. Let $mgu(t\gamma, H_i) = \sigma''$. Then due to $H_i\gamma = H_i$ and $mgu(t, H_i) = \sigma$ there must be a substitution σ''' such that $\gamma\sigma'' = \sigma\sigma'''$. W.l.o.g., we demand that $\mathcal{V}(Range(\sigma'')) \subseteq \mathcal{N}_{fresh}$.

By application of the concrete EVAL rule we obtain $B_i'\sigma'', Q\gamma\sigma'' \mid S\gamma$ where $B_i' = Transformed(B_i, m)$. We are, thus, left to show that $B_i'\sigma'', Q\gamma\sigma'' \mid S\gamma \in \mathcal{CON}(B_i'\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}))$, i.e., that there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $B_i'\sigma'' = B_i'\sigma'\gamma'$, $Q\gamma\sigma'' = Q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

We perform a case analysis over $\sigma' \in \{\sigma, \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}, \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\}$.

Case 1: $\sigma' = \sigma$, i.e., $\mathcal{A}(t) \subseteq \mathcal{G}$ and $\mathcal{N}(t) \subseteq \mathcal{F}$:
Here, we can assume $Dom(\sigma) = \mathcal{G}(t) \cup \mathcal{F}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$ and $\gamma'(a) = \gamma(a)$ otherwise. As $Range(\sigma)$ contains only fresh variables, we clearly have that $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

W.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma'''' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$. This is shown in [Sch08].

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in [Sch08].

To show that $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, we make a case analysis over $a \in \mathcal{G}' = \mathcal{G} \uplus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. For $a \in \mathcal{G}$ we know that $a\gamma \in GroundTerms(\Sigma)$ and by $\gamma'|_{\mathcal{G}}$

$\overset{\mathcal{A}(Range(\sigma))\subseteq\mathcal{V}_{fresh}\wedge Def.\gamma'}{=} \gamma|_{\mathcal{G}}$ we obtain $a\gamma' \in GroundTerms(\Sigma)$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $a\sigma|_{\mathcal{G}} \in FinitePrologTerms(\Sigma, \mathcal{A})$ and $a\gamma' = a\sigma'''$. For all $a' \in Dom(\sigma|_{\mathcal{G}})$, $a'\sigma\sigma''' \overset{Def.\sigma'''}{=} a'\gamma\sigma'' \overset{a'\in\mathcal{G}}{\in} GroundTerms(\Sigma)$. Thus, $a\gamma' \in GroundTerms(\Sigma)$.

Now, we are left to show that $B'_i\sigma'' = B'_i\sigma'\gamma'$, $Q\gamma\sigma'' = Q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$, which is also shown in [Sch08]. Thus, $\gamma\sigma'' = \sigma'\gamma'$ and $\gamma = \sigma|_{\mathcal{G}}\gamma'$ follows from the fact that we only defined γ' differently from γ for fresh variables.

This concludes the case of $\sigma' = \sigma$.

Case 2: $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$, i.e., $\mathcal{A}(t) \subseteq \mathcal{G}$ and $\mathcal{N}(t) \not\subseteq \mathcal{F}$:

Here, we can assume $Dom(\sigma) = \mathcal{G}(t) \cup \mathcal{N}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in \mathcal{A} \setminus \mathcal{G}'$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{\mathcal{A}\setminus\mathcal{G}'}$ are fresh. As $Range(\sigma) \cup Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})$ contains only fresh variables and $Dom(\alpha_{\mathcal{A}\setminus\mathcal{G}'}) = \mathcal{A} \setminus \mathcal{G}'$, we also have that $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

First, w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ as shown in [Sch08].

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\approx t'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in [Sch08].

$Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ follows from the identical argument as in the case $\sigma' = \sigma$.

Now, we are left to show that $B'_i\sigma'' = B'_i\sigma'\gamma'$, $Q\gamma\sigma'' = Q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$, which is also shown in [Sch08]. Thus, $\gamma\sigma'' = \sigma'\gamma'$ and $\gamma = \sigma|_{\mathcal{G}}\gamma'$ follows from the fact that we only defined γ' differently from γ for fresh variables.

This concludes the case of $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$.

Case 3: $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$, i.e., $\mathcal{A}(t) \not\subseteq \mathcal{G}$:

Here, we can assume $Dom(\sigma) = \mathcal{A}(t) \cup \mathcal{N}(t) \cup \mathcal{N}(H_i)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in (\mathcal{A} \setminus \mathcal{G}') \cup (\mathcal{N} \setminus \mathcal{F}')$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ are fresh. As $Range(\sigma) \cup Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})$ contains only fresh variables and $Dom(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}) = (\mathcal{A} \setminus \mathcal{G}') \cup (\mathcal{N} \setminus \mathcal{F}')$, we also have that $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.

First, w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma''''|_{\mathcal{F}\cup\mathcal{N}(H_i)} = \sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}\cup\mathcal{N}(H_i)}))}$ and $\sigma''''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))} = \sigma''|_{\mathcal{N}\setminus(\mathcal{F}\cup\mathcal{N}(H_i))}$. This is shown in [Sch08], too.

We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\approx t'\gamma'$.

Again, all these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in [Sch08].

By the identical argument as for the case of $\sigma' = \sigma$, we obtain $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$.

Now, we are left to show that $B_i'\sigma'' = B_i'\sigma'\gamma'$, $Q\gamma\sigma'' = Q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$, which is finally shown in [Sch08] as well. Thus, $\gamma\sigma'' = \sigma'\gamma'$ and $\gamma = \sigma|_{\mathcal{G}}\gamma'$ follows from the fact that we only defined γ' differently from γ for fresh variables.

This concludes the case of $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ and, consequently, our proof for the soundness of the Eval rule. □

Lemma 3.29 (Soundness of OnlyEval)**.** *The rule* OnlyEval *from Definition 3.27 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ with $mgu(t\gamma, H_i) = \sigma''$ there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $\gamma\sigma'' = \sigma'\gamma'$, $\gamma = \sigma|_{\mathcal{G}}\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.*

Proof. We have to show that if there is an infinite concrete state-derivation starting in $(t\gamma, Q\gamma)^i_m \mid S\gamma \in \mathcal{CON}((t,Q)^i_m \mid S;(\mathcal{G},\mathcal{F},\mathcal{U}))$ then there is an infinite concrete state-derivation starting in $B_i'\sigma'\gamma', Q\sigma'\gamma' \mid S\sigma|_{\mathcal{G}}\gamma' \in \mathcal{CON}(B_i'\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}))$ and the additional conditions for the substitutions used in this rule. Since the OnlyEval rule is identical to the Eval rule if we drop the right successor state of the Eval rule and all conditions of the Eval rule are implied by the conditions of the OnlyEval rule, we are left to show that there is no concretization γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ for which we have $t\gamma \nsim H_i$. Then the soundness of the OnlyEval rule and all additional conditions for the substitutions used in this rule are implied by the soundness and the identical conditions of the Eval rule. We show the equivalent condition: $\forall\gamma : (\gamma \text{ is a concretization w.r.t. } (\mathcal{G},\mathcal{F},\mathcal{U}) \implies t\gamma \sim H_i)$.
So let γ be a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$. We show $t\gamma \sim H_i$ by defining a unifier σ'' of $t\gamma$ and H_i. Since $\sigma|_{\mathcal{A}} : Dom(\sigma|_{\mathcal{A}}) \to Range(\sigma|_{\mathcal{A}})$ is bijective and $Range(\sigma|_{\mathcal{A}}) \subseteq \mathcal{A}$, there is a substitution σ^{-1} with $\sigma^{-1}(\sigma(a)) = a$ for all $a \in Dom(\sigma|_{\mathcal{A}})$ and $\sigma^{-1}(x) = x$ otherwise. So we have $\sigma\sigma^{-1} = \sigma|_{\mathcal{N}}$. As we have $t\sigma = H_i\sigma$, we obtain $t\sigma\sigma^{-1} = H_i\sigma\sigma^{-1} \iff t\sigma|_{\mathcal{N}} = H_i\sigma|_{\mathcal{N}}$. We define $\sigma'' = \sigma|_{\mathcal{N}}\gamma$. Then we have:

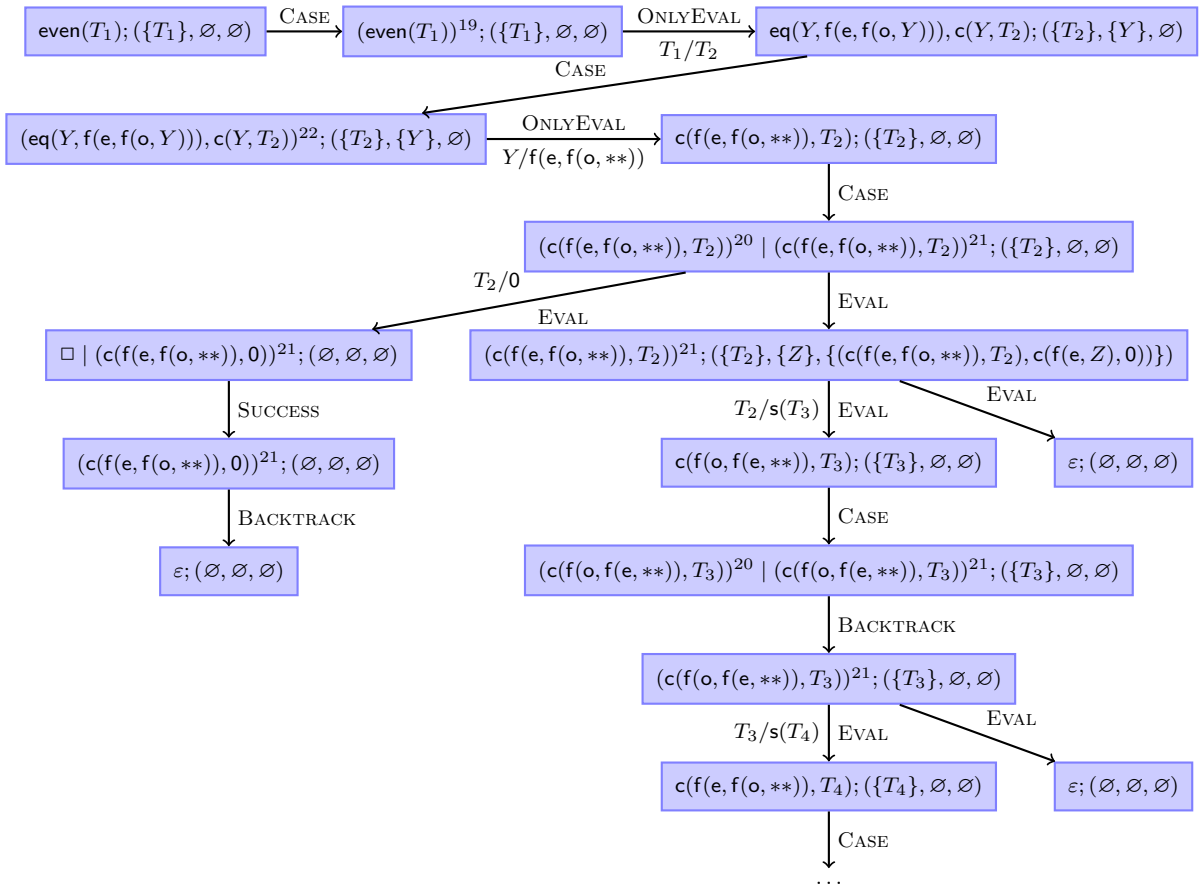
$$\begin{aligned}
t\gamma\sigma'' &\overset{Def.\sigma''}{=} t\gamma\sigma|_{\mathcal{N}}\gamma \\
&\overset{Dom(\gamma)\subseteq\mathcal{A}\wedge\gamma\gamma=\gamma}{=} t\sigma|_{\mathcal{N}}\gamma\gamma \\
&\overset{t\sigma|_{\mathcal{N}}=H_i\sigma|_{\mathcal{N}}}{=} H_i\sigma|_{\mathcal{N}}\gamma\gamma \\
&\overset{Dom(\gamma)\subseteq\mathcal{A}\wedge\gamma\gamma=\gamma}{=} H_i\gamma\sigma|_{\mathcal{N}}\gamma \\
&\overset{Def.\sigma''}{=} H_i\gamma\sigma'' \\
&\overset{Dom(\gamma)\subseteq\mathcal{A}}{=} H_i\sigma''
\end{aligned}$$

□

With these abstract rules any concrete state-derivations for sets of concrete states using the rules from Definition 3.11 can be simulated. If we obtain a finite abstract state-derivation with the abstract inference rules introduced so far, we can in fact infer termination of $\mathcal{P}$ w.r.t. the class of queries represented by the root node of the state-derivation tree. Unfortunately, in general we will obtain infinite state-derivation trees. This is even true for terminating goals, as the number of times an abstract evaluation may succeed is not limited. This is due to the absence of a bound on the size of the terms represented by abstract variables. The following example illustrates this problem.

Example 3.30. Consider again the Prolog program even.pl from Example 3.3 and the query set $\mathcal{Q} = \{\mathsf{even}(t) \mid t \text{ is ground}\}$. Clearly, the program terminates w.r.t. $\mathcal{Q}$.

Now, consider the tree built using the rules from Parts 1 – 3. For simplicity and as there are no cuts in this example, we omit the question marks and scopes. Also, we drop information from the knowledge base which is not relevant for the current state anymore. We will show in the next section that this is still correct.



Obviously, we obtain an infinite tree by continuing this process.

Therefore, to obtain finite graphs instead of infinite trees, we need a possibility to close the graph by referring back to already existing states. We introduce additional abstract inference rules for this purpose in the following section.

3.3 Finite Analysis

In order to avoid infinite graphs in spite of the fact that we do not have a bound on the size of the terms occurring in our states, we will define an INSTANCE rule, which can be used to refer back to an already existing state. Thus, we construct cyclic graphs instead of infinite trees. This INSTANCE rule has to ensure that the termination of the current state (called instance child) is implied by the termination of the state which we instantiate (called instance father). The instantiation is done by a matching substitution respecting the knowledge bases in the considered states and a renaming of the scopes occurring in the states in question. Then we can conclude termination of those states if the cyclic evaluation represented in the termination graph is well founded. To this end, we show in Chapter 7 how to extract dependency triple problems from termination graphs where termination of the dependency triple problems implies termination of the root states in the termination graphs.

Before we start, we define the notion of a *scope variant* and show that different scope variants allow for exactly the same concrete state-derivations. This property is used for the soundness proof of INSTANCE, later.

Definition 3.31 (Scope Variant)**.** *Given a concrete (abstract) state S, we call a concrete (abstract) state S' a* scope variant *of S, iff there is a bijection $f : \mathbb{N} \to \mathbb{N}$, both states have the same length and the following conditions are satisfied for all $i \in \{1, \ldots, length(S)\}$ and elements e_i of S at position i and e'_i of S' at position i:*

- *If e_i is an unlabeled list of terms t, then e'_i is an unlabeled list of terms t' with $t' = t[!_j/!_{f(j)} \forall j \in \mathbb{N}]$.*
- *If e_i is a labeled list of terms t^r_s, then e'_i is a labeled list of terms $t'^r_{f(s)}$ with $t' = t[!_j/!_{f(j)} \forall j \in \mathbb{N}]$.*
- *If $e_i = ?_s$, then $e'_i = ?_{f(s)}$.*

Example 3.32. Consider the state $\mathsf{p}, !_1 \mid (\mathsf{p})^2_1 \mid ?_1$. A scope variant of this state is for example $\mathsf{p}, !_2 \mid (\mathsf{p})^2_2 \mid ?_2$.

Note that if S' is a scope variant of S then S is also a scope variant of S' as the transformation between them is bijective.

Intuitively, a scope variant of some state is just a renaming of the scopes used in it and represents exactly the same concrete state-derivations. This is shown by the following two lemmata.

Lemma 3.33 (Equivalent Concrete State-Derivations for Concrete Scope Variants)**.** *Given a concrete state S and a scope variant S' of S, all concrete state-derivations possible for S are also possible for S'.*

Proof. To show Lemma 3.33 it is sufficient to show that for all concrete rules the applicability of a rule for S implies the applicability for S' and after application of the rule the resulting states are still scope variants of each other. We perform a case analysis over the applicability of the concrete inference rules for S.

- Success is applicable:
  Then we have $S = \square \mid S''$. Since S' is a scope variant of S, we also have $S' = \square \mid S'''$ and Success is applicable for S', too. After application of Success we obtain the states S'' and S''', which are scope variants of each other as $\square \mid S''$ and $\square \mid S'''$ are scope variants.

- Failure is applicable:
  Then we have $S = ?_s \mid S''$ and as S' is a scope variant of S, we also have $S' = ?_{f(s)} \mid S'''$. Thus, Failure is applicable for S', too. After application of Failure we obtain the states S'' and S''', which are scope variants of each other as $?_s \mid S''$ and $?_{f(s)} \mid S'''$ are scope variants.

- VariableError is applicable:
  Then we have $S = \mathsf{call}(x), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{call}(x), Q' \mid S'''$. Thus, VariableError is applicable for S', too. After application of VariableError we obtain the states ε and ε, which clearly are scope variants of each other.

- UndefinedError is applicable:
  Then we have $S = t, Q \mid S''$ where $\mathit{Slice}(\mathcal{P}, t) = \varnothing$ and as S' is a scope variant of S, we also have $S' = t', Q' \mid S'''$ with $\mathit{Slice}(\mathcal{P}, t') = \varnothing$. Thus, UndefinedError is applicable for S', too. After application of VariableError we obtain the states ε and ε, which clearly are scope variants of each other.

- Cut is applicable:
  Then we have $S = !_s, Q \mid S'' \mid ?_s \mid S'''$ with S'' contains no $?_s$ and as S' is a scope variant of S, we also have $S' = !_{f(s)}, Q' \mid S'''' \mid ?_{f(s)} \mid S''''$ with S'''' contains no $?_{f(s)}$. Thus, Cut is applicable for S', too. After application of Cut we obtain the states $Q \mid ?_s \mid S'''$ and $Q' \mid ?_{f(s)} \mid S''''$, which are scope variants of each other as $!_s, Q \mid S'' \mid ?_s \mid S'''$ and $!_{f(s)}, Q' \mid S'''' \mid ?_{f(s)} \mid S''''$ are scope variants.

- CutAll is applicable:
  Then we have $S = !_s, Q \mid S''$ with S'' contains no $?_s$ and as S' is a scope variant of S, we also have $S' = !_{f(s)}, Q' \mid S'''$ with S''' contains no $?_{f(s)}$. Thus, CutAll is applicable for S', too. After application of CutAll we obtain the states Q and Q', which are scope variants of each other as $!_s, Q \mid S''$ and $!_{f(s)}, Q' \mid S'''$ are scope variants.

- Case is applicable:
  Then we have $S = t, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = t', Q' \mid S'''$. Thus, Case is applicable for S', too. After application of Case we obtain the states $(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S''$ and $(t', Q')^{i'_1}_n \mid \ldots \mid (t', Q')^{i'_{k'}}_n \mid ?_n \mid S'''$, where m and n are fresh, $i_1 < \ldots < i_k$, $i'_1 < \ldots < i'_{k'}$, $\mathit{Slice}(\mathcal{P}, t) = \{c_{i_1}, \ldots, c_{i_k}\}$ and $\mathit{Slice}(\mathcal{P}, t') = \{c_{i'_1}, \ldots, c_{i'_{k'}}\}$. As S and S' are scope variants, t and t' have the same root symbol and, thus, we have $\mathit{Slice}(\mathcal{P}, t) = \mathit{Slice}(\mathcal{P}, t')$. W.l.o.g. we can also demand $f(m) = n$ as both m and n are fresh. Hence, the second state is $(t', Q')^{i_1}_{f(m)} \mid \ldots \mid (t', Q')^{i_k}_{f(m)} \mid ?_{f(m)} \mid S'''$, which is a scope variant of $(t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_k}_m \mid ?_m \mid S''$ as $t, Q \mid S''$ and $t', Q' \mid S'''$ are scope variants.

- Eval is applicable:
  Then we have $S = (t, Q)^i_m \mid S''$ with $c_i = H_i \leftarrow B_i$ and $\mathit{mgu}(t, H_i) = \sigma$ and as S' is a scope variant of S, we also have $S' = (t', Q')^i_{f(m)} \mid S'''$ with $\mathit{mgu}(t', H_i) = \sigma'$. Thus, Eval is applicable for S', too. After application of Eval we obtain the states $B'_i\sigma, Q\sigma \mid S''$ and $B''_i\sigma', Q'\sigma' \mid S'''$, where $B'_i = \mathit{Transformed}(B_i, m)$ and $B''_i = \mathit{Transformed}(B_i, f(m))$. As S and S' are scope variants, we have for all terms $r \in \mathit{Range}(\sigma)$ and $r' \in \mathit{Range}(\sigma')$ that $r' = r[!_j/!_{f(j)} \forall j \in \mathbb{N}]$ and $\mathit{Dom}(\sigma) = \mathit{Dom}(\sigma')$. Hence, $B'_i\sigma, Q\sigma \mid S''$ and $B''_i\sigma', Q'\sigma' \mid S'''$ are scope variants of each other as $(t, Q)^i_m \mid S''$ and $(t', Q')^i_{f(m)} \mid S'''$ are scope variants.

- Backtrack is applicable:
  Then we have $S = (t, Q)^i_m \mid S''$ where $c_i = H_i \leftarrow B_i$ and $t \nsim H_i$. As S' is a scope variant of S, we also have $S' = (t', Q')^i_{f(m)} \mid S'''$ where $t' \nsim H_i$. Thus, Backtrack is applicable for S', too. After application of Backtrack we obtain the states S'' and S''', which are scope variants of each other as $(t, Q)^i_m \mid S''$ and $(t', Q')^i_{f(m)} \mid S'''$ are scope variants.

- Call is applicable:
  Then we have $S = \mathsf{call}(t'), Q \mid S''$ where $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and t' has only finitely many predication positions. As S' is a scope variant of S, we also have $S' = \mathsf{call}(t''), Q' \mid S'''$ where $t'' \in \mathit{PrologTerms}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ and t'' has only finitely many predication positions. Thus, Call is applicable for S', too. After application of Call we obtain the states $t''', Q \mid ?_m \mid S''$ and $t'''', Q' \mid ?_{m'} \mid S'''$ where $t''' = \mathit{Transformed}(t', m)$ and $t'''' = \mathit{Transformed}(t'', m')$. As m and m' are fresh, we can demand $m' = f(m)$. Since the transformation by the function *Transformed* uses the same scope for all cuts in predication positions, the reached states are scope variants of each other as $\mathsf{call}(t'), Q \mid S''$ and $\mathsf{call}(t''), Q' \mid S'''$ are scope variants.

□

Lemma 3.34 (Equivalent Concrete State-Derivations for Abstract Scope Variants)**.** *Given an abstract state S and a scope variant S' of S, for every concrete state S_c represented by S there exists a concrete state S'_c represented by S' such that all concrete state-derivations possible for S_c are also possible for S'_c.*

Proof. As concretizations only replace abstract variables, we have for every concretization γ that $S'\gamma$ is a scope variant of $S\gamma$. By Lemma 3.33 we obtain that all concrete state-derivations possible for $S\gamma$ are also possible for $S'\gamma$. □

Example 3.35. Consider again the graph from Example 3.30. The abstract state $\mathsf{c}(\mathsf{f}(\mathsf{e},\mathsf{f}(\mathsf{o},**)),T_2);(\{T_2\},\varnothing,\varnothing)$ of the fourth node and the abstract state $\mathsf{c}(\mathsf{f}(\mathsf{e},\mathsf{f}(\mathsf{o},**)),T_4);(\{T_4\},\varnothing,\varnothing)$ of the last node are very similar. Indeed, if one uses a substitution $\mu = \{T_2/T_4\}$ we see that the last state is an instance of the fourth state.

The basic idea of the following Instance rule is that we can show that some abstract state is an instance of another abstract state, instead of showing that the abstract state is terminating by applying the abstract inference rules from Parts 1 – 3. Let $S;(\mathcal{G},\mathcal{F},\mathcal{U})$ be the instance child and $S';(\mathcal{G}',\mathcal{F}',\mathcal{U}')$ the instance father, i.e., there is a substitution μ such that $S = S'\mu$.

To define a sound Instance rule, we must make sure that for every concrete state-derivation for some concrete state represented by the instance child, we have the same concrete state-derivation for some concrete state represented by the instance father. In particular, the matching substitution μ must respect the knowledge in both states. Thus, we have to ensure that all abstract variables from $\mathcal{G}'$ are instantiated by μ to a term for which all variables are from $\mathcal{G}$. As we consider unification without occurs-check, we have to ensure that ground terms are replaced by finite terms, too. So we demand for all $a \in \mathcal{G}'$, that $a\mu \in \mathit{FinitePrologTerms}(\Sigma,\mathcal{G})$. While we may not instantiate non-abstract variables in general as such variables do not represent any other term than themselves, we allow μ to be a variable renaming on $\mathcal{N}$. This is correct as every concrete state-derivation for some goal works identically for a goal where all variables are renamed. However, this renaming must respect the knowledge about free variables. Hence, we demand that $\mathcal{F}'\mu \subseteq \mathcal{F}$. Furthermore, μ must not instantiate abstract variables with terms containing free variables. To avoid this, we demand $\mathcal{F}'\mu(\mathit{Range}(\mu|_{\mathcal{A}})) = \varnothing$. Finally, we have to show that the non-unification information in $\mathcal{U}'$ is implied by the one in $\mathcal{U}$. This can be ensured by having $\mathcal{U}'\mu$ being a subset of $\mathcal{U}$.

While the Instance rule from [Sch08] is also used for generalization, we distinguish between the application of the Instance rule to already existing states in our graph and the application of the Generalization rule to new states to obtain a more intuitive description for the heuristic in Chapter 6. Apart from this, the two rules are identical.

Definition 3.36 (Abstract Inference Rules – Part 4 (Instance, Generalization))**.**

$$\frac{S;(\mathcal{G},\mathcal{F},\mathcal{U})}{S';(\mathcal{G}',\mathcal{F}',\mathcal{U}')}\ (\textsc{Instance})$$

if there is a μ *such that* $S = S''\mu$ *for a scope variant* S'' *of* S'*, for all* $a \in \mathcal{G}'$*,* $a\mu \in FinitePrologTerms(\Sigma,\mathcal{G})$*,* $\mu|_{\mathcal{N}}$ *is a variable renaming,* $\mathcal{F}'\mu \subseteq \mathcal{F}$*,* $\mathcal{F}'\mu(Range(\mu|_{\mathcal{A}})) = \varnothing$*,* $\mathcal{U}'\mu \subseteq \mathcal{U}$ *and the state* $(S';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ *is already existing in our graph.*

$$\frac{S;(\mathcal{G},\mathcal{F},\mathcal{U})}{S';(\mathcal{G}',\mathcal{F}',\mathcal{U}')}\ (\textsc{Generalization})$$

if there is a μ *such that* $S = S''\mu$ *for a scope variant* S'' *of* S'*, for all* $a \in \mathcal{G}'$*,* $a\mu \in FinitePrologTerms(\Sigma,\mathcal{G})$*,* $\mu|_{\mathcal{N}}$ *is a variable renaming,* $\mathcal{F}'\mu \subseteq \mathcal{F}$*,* $\mathcal{F}'\mu(Range(\mu|_{\mathcal{A}})) = \varnothing$*,* $\mathcal{U}'\mu \subseteq \mathcal{U}$ *and the state* $(S';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ *is not already existing in our graph.*

Lemma 3.37 (Soundness of Instance)**.** *The rule* Instance *from Definition 3.36 is sound. Additionally, for every concretization* γ *w.r.t.* $(\mathcal{G},\mathcal{F},\mathcal{U})$ *there is a concretization* γ' *w.r.t.* $(\mathcal{G}',\mathcal{F}',\mathcal{U}')$ *such that* $S\gamma = S''\gamma'\mu|_{\mathcal{N}}$*.*

Proof. Assume we have an infinite concrete state-derivation starting from $S\gamma \in \mathcal{CON}(S;(\mathcal{G},\mathcal{F},\mathcal{U}))$. We show that there is a substitution γ' such that $S'\gamma' \in \mathcal{CON}(S';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ and $S'\gamma'$ has an infinite concrete state-derivation.

For this purpose, we first show that $S''\gamma' \in \mathcal{CON}(S'';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ has an infinite concrete state-derivation.

Following the proof in [Sch08], there must be a μ^{-1} such that $\mu|_{\mathcal{N}}\mu^{-1} = \mu^{-1}\mu|_{\mathcal{N}} = id$ as $\mu|_{\mathcal{N}}$ is a variable renaming. Let $\gamma' = \mu\gamma\mu^{-1}$. Clearly, as $S''\mu = S$ and μ^{-1} is a variable renaming, $S''\gamma' = S\gamma\mu^{-1}$ has an infinite concrete state-derivation. Additionally, we have that $S''\gamma'\mu|_{\mathcal{N}} = S\gamma\mu^{-1}\mu|_{\mathcal{N}} = S\gamma$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}')$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}} t\gamma' \not\approx t'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in [Sch08].

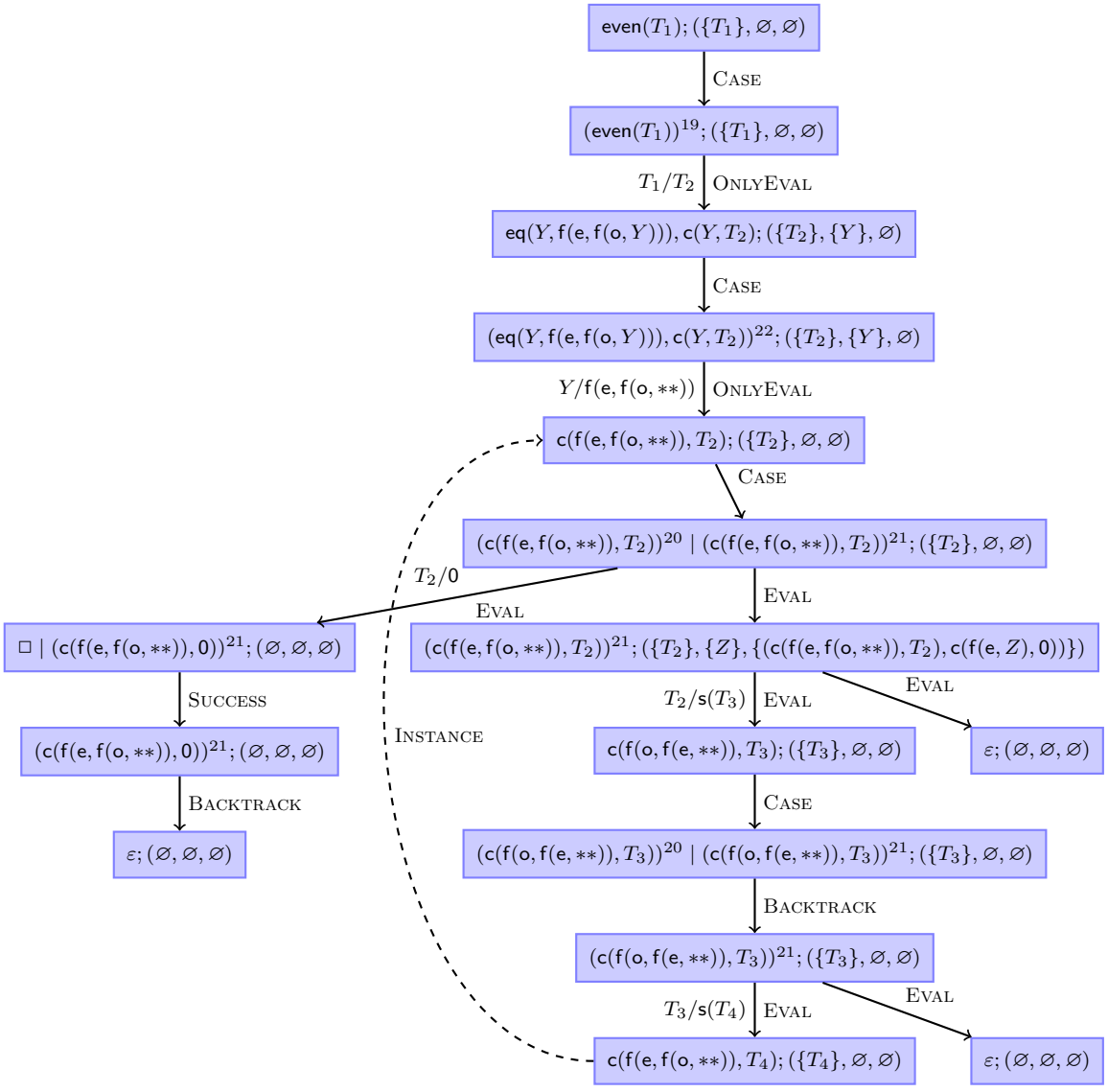
We know that for all $a \in \mathcal{G}'$, $a\mu \in FinitePrologTerms(\Sigma,\mathcal{G})$. Further, as γ is a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ we know that for all $a \in \mathcal{G}$, $a\gamma \in GroundTerms(\Sigma)$. Thus, for all $a \in \mathcal{G}'$, we have $a\gamma' \overset{Def.\gamma'}{=} a\mu\gamma\mu^{-1} = a\mu\gamma \in GroundTerms(\Sigma)$ and, therefore, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$.

Since S'' is a scope variant of S' and γ' replaces only abstract variables, $S''\gamma'$ is also a scope variant of $S'\gamma'$. As $S''\gamma' \in \mathcal{CON}(S'';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ has an infinite concrete state-derivation, we obtain by Lemma 3.34 that $S'\gamma' \in \mathcal{CON}(S';(\mathcal{G}',\mathcal{F}',\mathcal{U}'))$ has an infinite concrete state-derivation, too. □

Lemma 3.38 (Soundness of Generalization)**.** *The rule* Generalization *from Definition 3.36 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}')$ such that $S\gamma = S''\gamma'\mu|_{\mathcal{N}}$.*

Proof. Since the Generalization rule is identical to the Instance rule except for the existence condition of a state in the graph and this condition does not affect the concrete states represented by an abstract state, the soundness of the Generalization rule and the additional conditions for the substitutions used in this rule are implied by the soundness and the identical conditions of the Instance rule. □

Example 3.39. Now, we can close the graph from Example 3.30 using Instance and obtain the following finite graph.



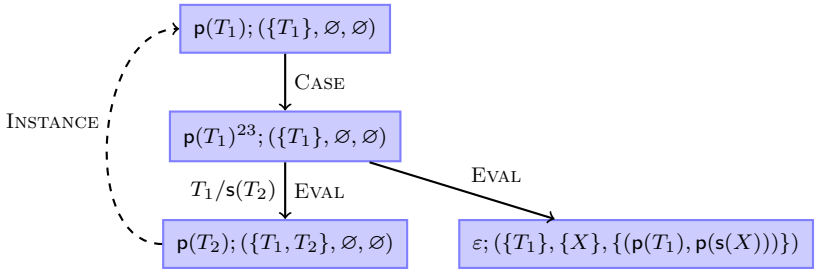
Note that we used some simplifications in the example above. First, we dropped information from the knowledge bases. This is correct due to the Generalization rule.

The Generalization rule can be used for three purposes. The first possibility is to generalize a knowledge base. This allows for example to get rid of superfluous information.

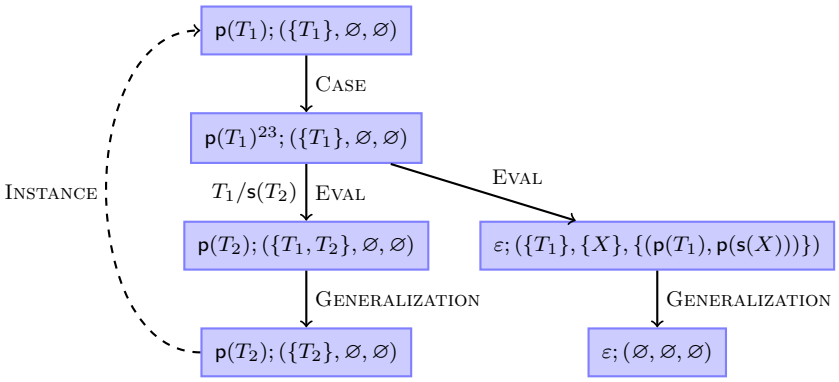
Example 3.40. We already used Generalization implicitly in Example 3.30 and Example 3.39 to drop information from the knowledge bases which is not relevant for the respective states anymore. To show the difference explicitly, consider the very simple Prolog program $\mathcal{P}$ consisting of only one clause

$$\mathsf{p}(\mathsf{s}(X)) \quad \leftarrow \quad \mathsf{p}(X). \tag{23}$$

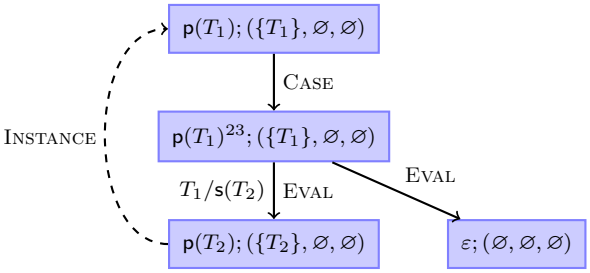
and the set of queries $\mathcal{Q} = \{\mathsf{p}(t) \mid t \text{ is ground}\}$ which is already used as an example in [Sch08]. For the moment, we ignore the question marks and scopes again. Without Generalization, we obtain the following graph with superfluous information in the knowledge bases.



Now, using Generalization we can drop the superfluous information.



Using Generalization implicitly, we obtain the following more readable graph.



From here on we will always use implicit generalizations in our examples to keep the knowledge base relevant to the current state. In particular, information about variables no longer used in the state can safely be disregarded. This specific way of using the Generalization rule does not lose precision.

Generalization may also be used to obtain a smaller analysis in cases, where the lost precision still allows to prove termination and the extracted DT problem is not (much) harder than the one we obtain without the generalization. Thus, we may improve the speed of our analysis in such cases.

Example 3.41. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{q} \leftarrow \mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))))))))). \tag{24}$$

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{25}$$

$$\mathsf{p}(\mathsf{0}) \leftarrow \square. \tag{26}$$

and query set $\mathcal{Q}$ consisting of the single query q. Instead of evaluating the concrete goal $\mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))))))))$ ten times, we can generalize it to $\mathsf{p}(T_1)$ where we know that T_1 is ground. This goal must only be evaluated once to find an instance and we still obtain a terminating DT problem for the constructed graph by the transformation from Chapter 7. While our approach is, of course, not necessary for termination analysis of such simple programs, one can use similar parts in a more complex program such that the use of Generalization may offer a significant speed gain.

Third, the use of the Generalization rule is for example needed for programs where terms are growing during the evaluation and those terms do not contain variables. While this way of using the Generalization rule may significantly lose precision, it is necessary to obtain a finite analysis in such cases.

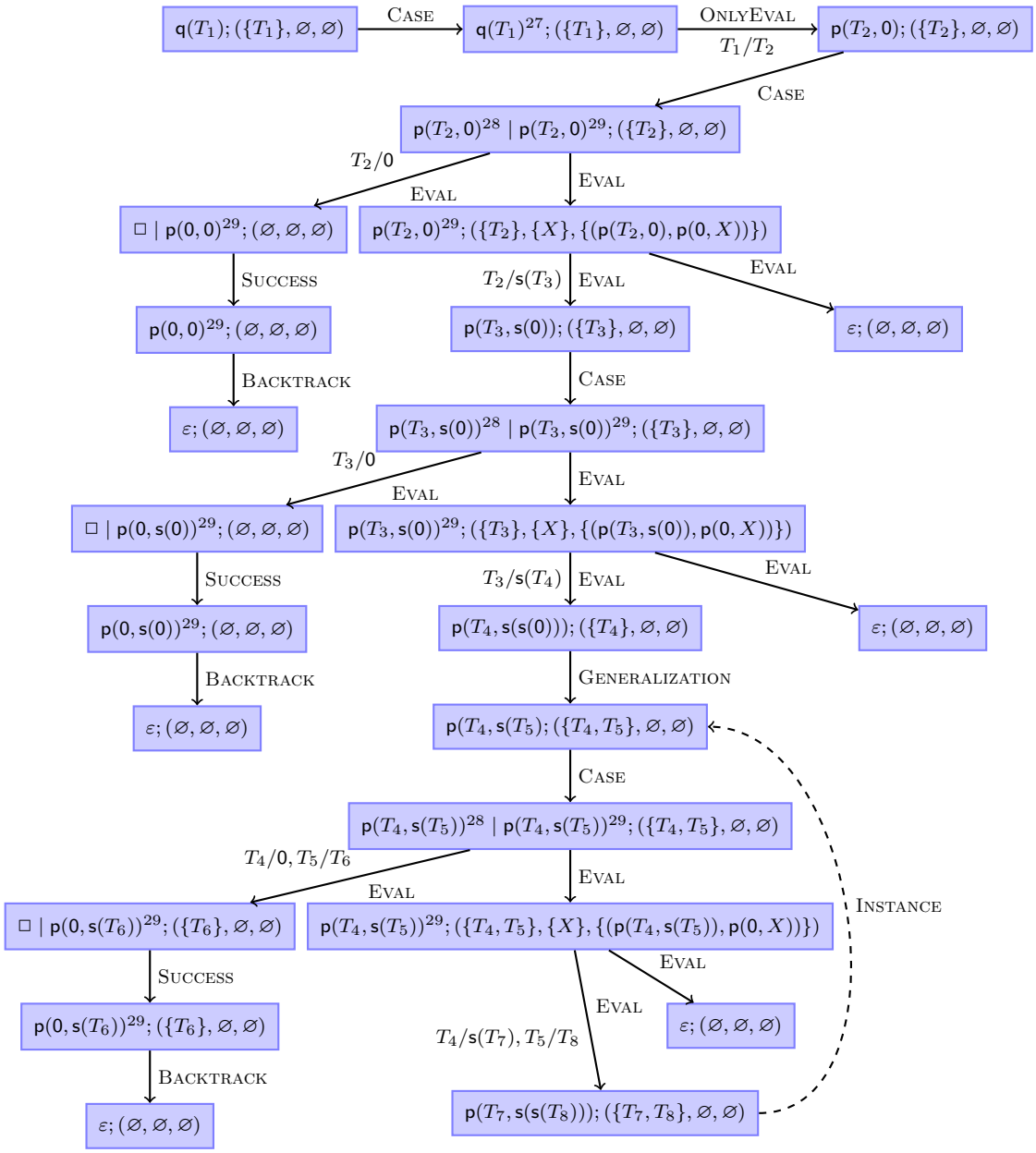
Example 3.42. Consider the following Prolog program ts08.pl:

$$\mathsf{q}(X) \leftarrow \mathsf{p}(X, \mathsf{0}). \tag{27}$$

$$\mathsf{p}(\mathsf{0}, X) \leftarrow \square. \tag{28}$$

$$\mathsf{p}(\mathsf{s}(X), Y) \leftarrow \mathsf{p}(X, \mathsf{s}(Y)). \tag{29}$$

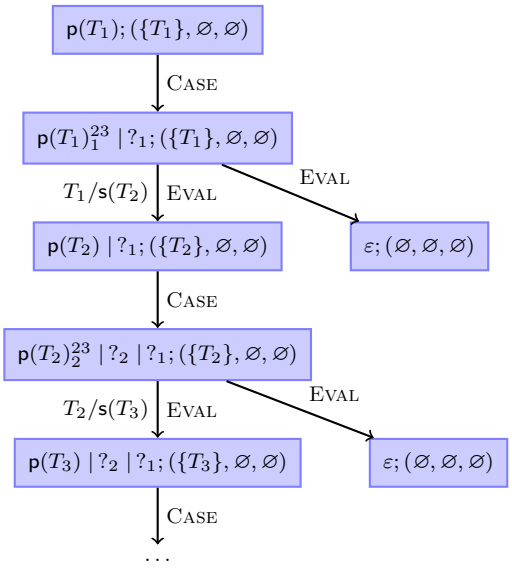
For queries of the form $\mathsf{q}(t)$ where t is ground, the program terminates. But as the second argument of p grows in each evaluation step by one s symbol and it does not contain any variables, there will never be an already existing state where we can apply the Instance rule to. However, by generalizing the term $\mathsf{s}(\mathsf{0})$ to a fresh ground variable, we can use the Generalization rule to a new state which can then be used for instantiation after further evaluation. Again, we ignore the question marks and scopes for the moment.



Where and when to use Generalization for an advantageous analysis is not easy to decide. We will present a heuristic in Chapter 6 which has shown to be successful in practice.

Still, being able to refer back to already existing states and to generalize states is not enough to obtain an expedient finite analysis as we do not want to construct cycles only consisting of Instance and Generalization edges. The following example demonstrates, that we still cannot close the graph even for simple programs.

Example 3.43. In the last examples we omitted the question marks and scopes to simplify the graphs. This is not correct in general since cuts may then drop too many state elements. Still, we need to get rid of superfluous backtracking goals, even if they only consist of the question marks in order to find instances. To see this, note that we would in fact still obtain an infinite tree even for the simple Prolog program $\mathcal{P}$ from Example 3.40.

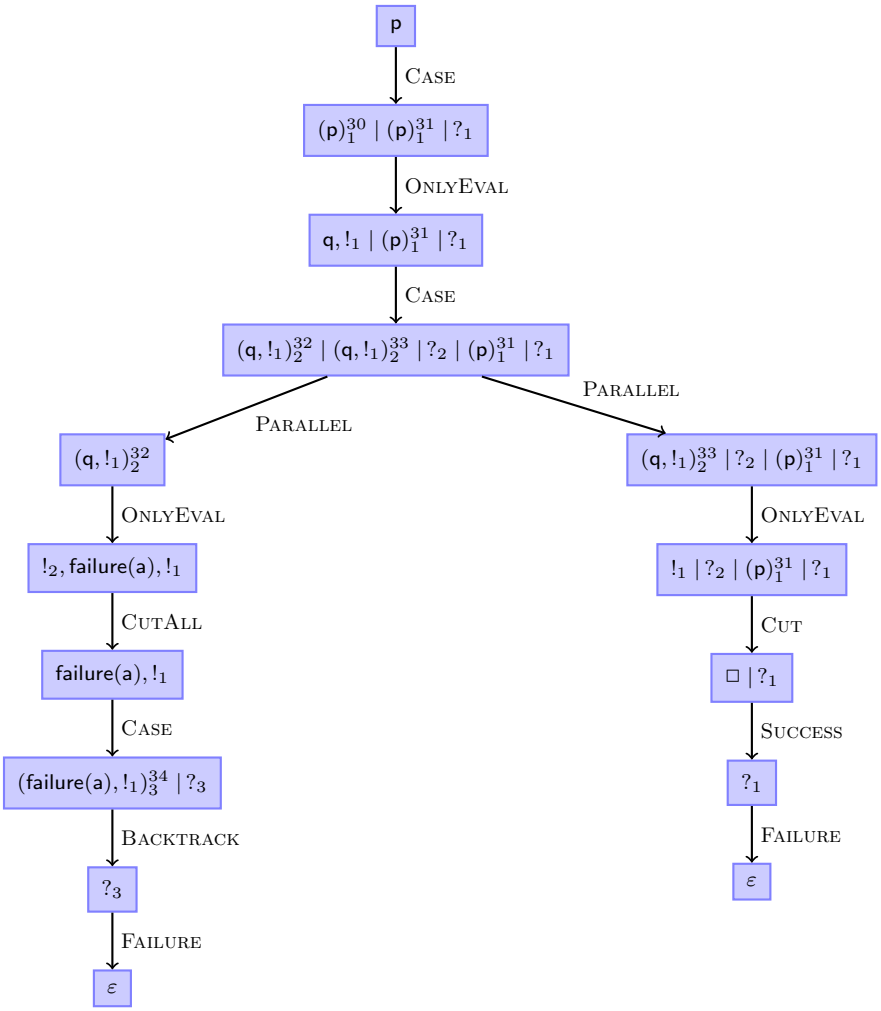


The problem is due to the possibility of introducing new state elements in every evaluation. Thus, the states grow bigger and bigger without reaching a state where we can find an instance father for. To solve this problem, we introduce the Parallel rule which is capable of splitting a backtracking list of a state into two separate backtracking lists for its successor states. Although this rule may lose precision, we virtually always need it to find instances. But even worse, the splitting of backtracking lists can also be incorrect due to cuts. The reason for this is that we can split the backtracking list at a point where a cut removes backtracking goals. Thus, we preserve some backtracking goals from being cut. These can in turn reach a cut for a lower scope level and, hence, cut off more backtracking goals than it would have been possible without the splitting of the backtracking list. The additionally cut off goals may then have an infinite concrete state-derivation and cause the incorrectness.

Example 3.44. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{p} \leftarrow \mathsf{q}, !. \tag{30}$$
$$\mathsf{p} \leftarrow \mathsf{p}. \tag{31}$$
$$\mathsf{q} \leftarrow !, \mathsf{failure}(\mathsf{a}). \tag{32}$$
$$\mathsf{q} \leftarrow \square. \tag{33}$$
$$\mathsf{failure}(\mathsf{b}) \leftarrow \square. \tag{34}$$

and the set of queries $\mathcal{Q} = \{\mathsf{p}\}$. This program is not terminating w.r.t. $\mathcal{Q}$ as the predicate q is always failing and, hence, we cannot reach the cut in clause 30. Using Parallel without any conditions on the backtracking lists, we can construct the following finite acyclic graph and falsely prove termination. As we do not have any variables in this example, we omit the knowledge bases for simplicity as they would only consist of empty sets.



To overcome this problem, we must consider the effects of cuts in the backtracking list. However, not every cut occurring in the backtracking list causes a problem for the Parallel rule. We introduce the concept of active cuts and active marks to characterize positions where the splitting of the backtracking list is unproblematic. Both active cuts and active marks are represented by a subset of $\mathbb{N}$. Active cuts contain all those m for which $!_m$ occurs in S or $(t, Q)_m^i$ occurs in S and B_i contains a cut where $H_i \leftarrow B_i$ is the i-th clause which is supposed to be applied to $(t, Q)_m^i$. The latter is due to the fact that the Eval rule might introduce the cut with the scope m once we reach the corresponding state element. Likewise, the active marks contain those m for which $?_m$ occurs in S at a position different from the first or last position. The former is due to the fact that we will discard question marks at the first position with the Failure rule such that they do not have any effect for cuts. The latter is possible, as applying the CutAll rule is equivalent to applying the Cut rule to a state where the corresponding question mark for the cut in question is at the last position of the state.

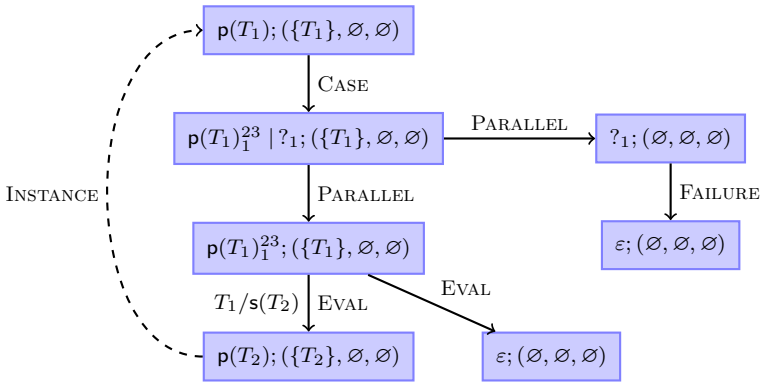
Definition 3.45 (Abstract Inference Rules – Part 5 (Parallel) [Sch08])**.**

$$\frac{S \mid S'; KB}{S; KB \qquad S'; KB}\ (\textsc{Parallel}) \quad \textit{if } AC(S) \cap AM(S') = \varnothing$$

Here, the active cuts $AC(S)$ *of a state* S *are defined as the set of all* m *such that* $S = S' \mid Q, !_m, Q' \mid S''$ *or* $S = S' \mid (t, Q)^j_m \mid S''$ *and* $c_j = H_j \leftarrow B_j, !, B'_j$*, while the* active marks $AM(S)$ *of a state* S *are defined as all* m *such that* $S = S' \mid ?_m \mid S''$ *and* $S' \neq \varepsilon \neq S''$*.*

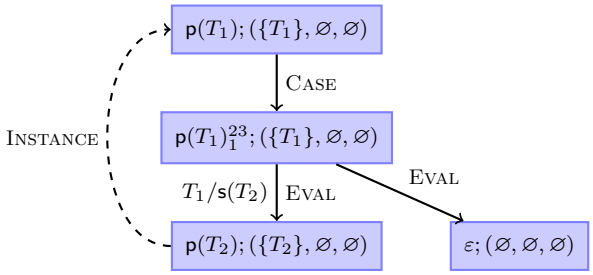
In [Sch08] the Parallel rule is shown to be sound.

Example 3.46. Consider once more the one-rule logic program from Example 3.40. Now, using the rules from Parts 1 – 5 we can obtain the following finite tree.



By using Parallel and Failure we can always remove question marks at the last position of a state. In the remainder of the thesis we will always implicitly use these two rules in such cases.

Example 3.47. Consider for the last time the one-rule logic program from Example 3.40. Using implicit removal of trailing question marks, we really obtain the last finite graph from Example 3.40.



So far, we have demonstrated how to get rid of trailing question marks using the Parallel rule. But this is of course not the only purpose for introducing it. In general, any kind of state elements must be split from the remaining backtracking list to find instances (and, thus, obtain a finite analysis).

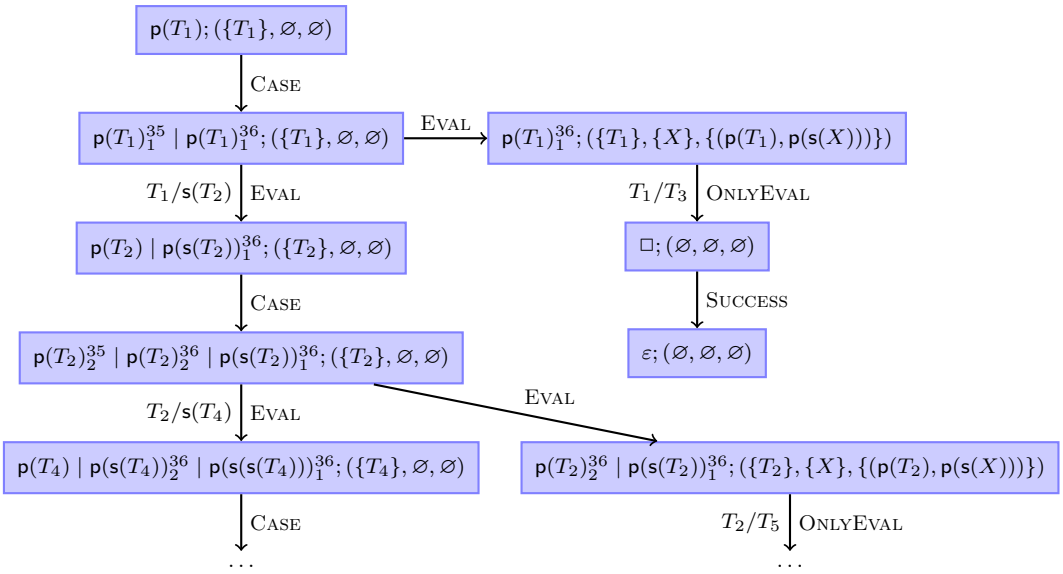
Example 3.48. We extend the simple Prolog program from Example 3.40 by a fact and obtain the Prolog program

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{35}$$

$$\mathsf{p}(X) \leftarrow \square. \tag{36}$$

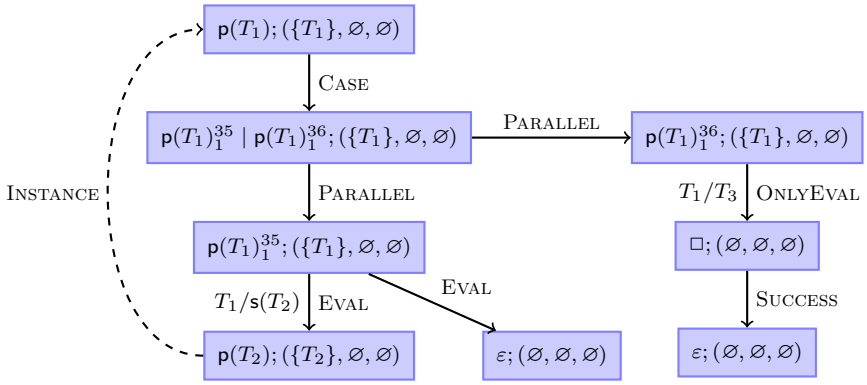
and the query set $\mathcal{Q} = \{\mathsf{p}(t) \mid t \text{ is ground}\}$. This program also clearly terminates w.r.t. $\mathcal{Q}$.

Now, consider the graph built using the rules from Parts 1 – 5 where Parallel is only used implicitly to remove trailing question marks:



This process can obviously be continued infinitely often without encountering an instance of a previous state along the leftmost path. The reason is that each application of the Case rule produces an additional backtracking goal.

Now, by using Parallel more liberally, we can obtain the following alternative graph:



Thus, we have to use the Parallel rule in these non-trivial cases to close the graph.

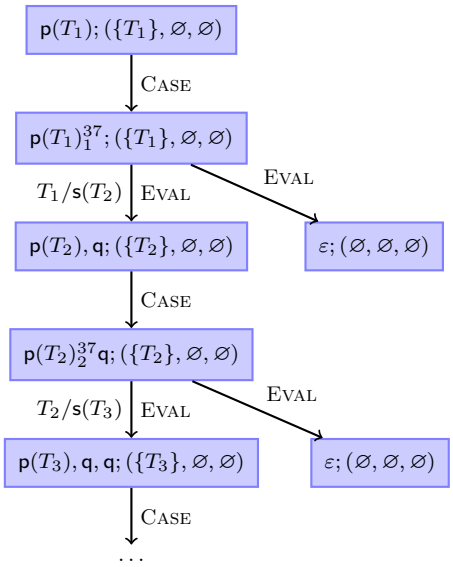
While we are able to close the graph for growing terms (using Generalization) and growing backtracking lists (using Parallel), we still have a problem due to growing goals, i.e., growing term lists inside of one state element.

Example 3.49. We again modify the Prolog program from Example 3.40 by appending an additional goal to the body of the clause and adding a corresponding fact. Thus, consider the resulting simple Prolog program $\mathcal{P}$

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X), \mathsf{q}. \tag{37}$$

$$\mathsf{q} \leftarrow \square. \tag{38}$$

and the query set $\mathcal{Q} = \{\mathsf{p}(t) \mid t \text{ is ground}\}$. This program again clearly terminates w.r.t. $\mathcal{Q}$. Now, consider the graph built using the rules from Parts 1 – 5:



This process can obviously be continued infinitely often without encountering an instance of a previous state along the leftmost path. The reason is that the goal grows by one q for each application of clause 37.

For splitting goals containing more than one term, we therefore introduce the abstract Split rule. The basic idea for a state $t', Q \mid S; KB$[9] is to consider the first term t' in the goal alone and approximate its answer substitution for the remaining goal Q. Again, we have the same problem as for Parallel. The active cuts in t', Q must not have a corresponding active mark in S. In addition to that we must take into account that if the concrete state-derivation for some concrete state represented by $t'; KB$ fails, we would have to backtrack to $S; KB$ instead of $Q \mid S; KB$. To avoid these problems we restrict the Split rule to states having only one state element. Using Parallel repeatedly, we

[9] We use t' instead of t here as t' may also have a built-in predicate as its root symbol.

can always split backtracking lists into single state elements as we will see in the heuristic presented in Chapter 6.

Thus, we take a state $t', Q; (\mathcal{G}, \mathcal{F}, \mathcal{U})$ and split it into two successors $t'; (\mathcal{G}, \mathcal{F}, \mathcal{U})$ and $Q\mu; (\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$. Here, μ is an overapproximation of the answer substitutions for t'. All variables in t' not being from $\mathcal{G}$ are potentially instantiated and, thus, we replace them by fresh abstract variables. As for the EVAL rule, we also have to consider sharing effects. If t' contains non-abstract variables not in $\mathcal{F}$, we have to replace all abstract variables not being from $\mathcal{G}$ in Q. If t' also contains abstract variables not in $\mathcal{G}$, we additionally have to replace all non-abstract variables not being from $\mathcal{F}$ in Q. After instantiating free variables by μ', the resulting terms do not necessarily contain free variables anymore. Thus, we have to drop the free variables in t' from $\mathcal{F}'$.

To obtain a more precise analysis, we make use of a groundness analysis for the answer substitutions μ'. The groundness analysis has to decide whether an argument position of a function symbol has to be instantiated by a ground term for all answer substitutions. The information a groundness analysis may use is the program, the function symbol and a set of argument positions for the function symbol known to be ground. Thus, we can update the set $\mathcal{G}'$ by adding those variables introduced by μ which are at a position for which the groundness analysis knows that only ground terms can be instantiated there by the answer substitutions.

Finally, we update $\mathcal{U}$ by applying the overapproximation μ.

Definition 3.50 (Abstract Inference Rules – Part 6 (SPLIT))**.**

$$\frac{t', Q; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{t'; (\mathcal{G}, \mathcal{F}, \mathcal{U}) \qquad Q\mu; (\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)} \text{ (SPLIT)}$$

where $t' \neq !_m$ *for some* $m \in \mathbb{N}$, $t' \neq \mathsf{call}(\mathsf{x})$ *for some* $x \in \mathcal{V}$, $root(t') \in BuiltInPredicates \vee Slice(\mathcal{P}, t') \neq \varnothing$, $\mu = ApproxSub(t', \mathcal{G}, \mathcal{F})$, $\mathcal{G}' = \mathcal{G} \cup ApproxGnd(t', \mu)$, *and* $\mathcal{F}' = \mathcal{F} \setminus \mathcal{F}(t')$.

Here, ApproxSub approximates the substitutions of the answer sets of all concretizations w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ *of* t'*:*

$$ApproxSub(t', \mathcal{G}, \mathcal{F}) = \begin{cases} \alpha_{\mathcal{F}(t')} & \text{if } \mathcal{V}(t') \subseteq \mathcal{G} \cup \mathcal{F} \\ \alpha_{\mathcal{N}(t')}\alpha_{\mathcal{A}\setminus\mathcal{G}} & \text{if } \mathcal{A}(t') \subseteq \mathcal{G} \wedge \mathcal{N}(t') \not\subseteq \mathcal{F} \\ \alpha_{\mathcal{F}(t')}\alpha_{\mathcal{A}\setminus\mathcal{G}}\alpha_{\mathcal{N}\setminus\mathcal{F}} & \text{otherwise} \end{cases}$$

Finally, ApproxGnd approximates the abstract variables that have to be instantiated by ground terms using a given groundness analysis $Ground_{\mathcal{P}} : \Sigma \times 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ *which given a predicate* p *and a set of ground argument positions computes the set of ground arguments positions after a successful computation using the clauses from* $\mathcal{P}$*:*

$$ApproxGnd(t', \mu) = \{\mathcal{A}(t_i\mu) \mid t' = p(t_1, \ldots, t_n), i \in Ground_{Slice(\mathcal{P}, t')}(p, \{i \mid \mathcal{V}(t_i) \subseteq \mathcal{G}\})\}$$

In addition to the conventions we already introduced for Definition 3.11 and Definition 3.21, we consider t' to be an arbitrary term from $\mathit{PrologTerms}(\Sigma, \mathcal{V})$.

Lemma 3.51 (Soundness of SPLIT)**.** *The rule* SPLIT *from Definition 3.50 is sound. Additionally, for every concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ and for every answer substitution μ' of a successful concrete state-derivation for $t\gamma$, there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$ such that $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{G}\cup\mathcal{A}(\mathcal{U})}$.*

Proof. Assume that $t'\gamma, Q\gamma \in \mathcal{CON}(t', Q; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite concrete state-derivation. Then, following the proof in [Sch08], there are two cases. If $t'\gamma$ has an infinite concrete state-derivation, we immediately have that $t'\gamma \in \mathcal{CON}(t'; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ has an infinite concrete state-derivation. If $t'\gamma$ does not have an infinite concrete state-derivation and we did not reach a state of the form $Q\gamma\mu' \mid S'\gamma$ for some answer substitution μ' and state S', we would reach the state ε, which contradicts our assumption that $t'\gamma, Q\gamma$ has an infinite concrete state-derivation. Therefore, if $t'\gamma$ does not have an infinite concrete state-derivation, we reach states of the form $Q\gamma\mu' \mid S'\gamma$ for answer substitutions μ' and states S'. If all $Q\gamma\mu'$ did not have an infinite concrete state-derivation, this would contradict our assumption that $t'\gamma, Q\gamma$ has an infinite concrete state-derivation. Thus, there must be a state $Q\gamma\mu'$ that has an infinite concrete state-derivation. We now show that there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$ such that $\gamma\mu' = \mu\gamma'$ for all answer substitutions μ' corresponding to a successful concrete state-derivation of $t\gamma$. Then, in particular, we have an infinite concrete state-derivation from $Q\mu\gamma' \in \mathcal{CON}(Q\mu; (\mathcal{G}', \mathcal{F}', \mathcal{U}\mu))$. There are three subcases.

First, if $\mathcal{V}(t') \subseteq \mathcal{G} \cup \mathcal{F}$ we have $t'\gamma \in \mathit{PrologTerms}(\Sigma, \mathcal{F})$ as γ is a concretization and, therefore, for all $a \in \mathcal{G}(t')$, $a\gamma \in \mathit{GroundTerms}(\Sigma)$. Thus, we have $\mathit{Dom}(\mu') \subseteq \mathcal{F}(t'\gamma)$. From $\mu = \alpha_{\mathcal{F}(t')}$ we know that for all $x \in \mathcal{F}(t'\gamma) = \mathcal{F}(t')$, $x\mu \in \mathcal{A}$ is a fresh variable. We define $\gamma'(x\mu) = x\mu'$ for $x \in \mathcal{F}(t')$ and $\gamma'(x) = \gamma(x)$ otherwise. Then, obviously, $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $\mathit{Range}(\gamma'|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}'(\mathit{Range}(\gamma')) = \varnothing$, and $\bigwedge_{(s,s')\in\mathcal{U}\mu} s\gamma' \not\approx s'\gamma'$.

All these properties except for $\mathit{Range}(\gamma'|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$ are shown in [Sch08].

We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (\mathit{ApproxGnd}(t', \mu) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have defined $a\gamma' = a\gamma$ and thus $a\gamma' \in \mathit{GroundTerms}(\Sigma)$. For $a \in \mathit{ApproxGnd}(t', \mu) \setminus \mathcal{G}$ by definition of $\mathit{ApproxGnd}$ and equality of $\gamma\mu'$ and $\mu\gamma'$ we know that $a\gamma' \in \mathit{GroundTerms}(\Sigma)$.

Second, if $\mathcal{A}(t') \subseteq \mathcal{G}$, but $\mathcal{N}(t') \not\subseteq \mathcal{F}$, the answer substitution μ' can instantiate non-abstract variables in t' which might occur in the terms represented by the abstract variables in Q. However, μ' cannot instantiate non-abstract variables not occurring in t'. We define γ' in such a way that $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. This is always possible because $\mathit{Dom}(\mu') \cap (\mathcal{N} \setminus \mathcal{N}(t)) = \varnothing$ and all variables in $\mathit{Range}(\mu)$ are fresh. Then,

clearly, $Q\gamma\mu' = Q\mu\gamma$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$. As we only need to define γ' for abstract variables, clearly $\gamma'|_{\mathcal{A}} = \gamma'$. From $\mathcal{A}(Range(\mu')) = \varnothing$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we know that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$. We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t', \mu) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have effectively defined $a\gamma' = a\gamma$ and thus $a\gamma' \in GroundTerms(\Sigma)$. For $a \in ApproxGnd(t', \mu) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\mu'$ and $\mu\gamma'$ we again know that $a\gamma' \in GroundTerms(\Sigma)$. Furthermore, note that w.l.o.g. $\mathcal{F}(Range(\mu')) \subseteq \mathcal{F}(t')$ and $\mathcal{F}(Range(\gamma)) = \varnothing$. Thus, $\mathcal{F}(Range(\gamma')) \subseteq \mathcal{F}(t')$ and, consequently, $\mathcal{F}'(Range(\gamma')) = \varnothing$. For all $(s, s') \in \mathcal{U}$ we have $s\gamma \not\approx s'\gamma$ and, consequently $s\gamma\mu' \not\approx s'\gamma\mu'$. But from $s\gamma\mu' = s\mu\gamma'$ and $s'\gamma\mu' = s'\mu\gamma'$ we get $s\mu\gamma' \not\approx s'\mu\gamma'$. Thus, for all $(s'', s''') \in \mathcal{U}\mu$, we have $s\gamma' \not\approx s'\gamma'$.

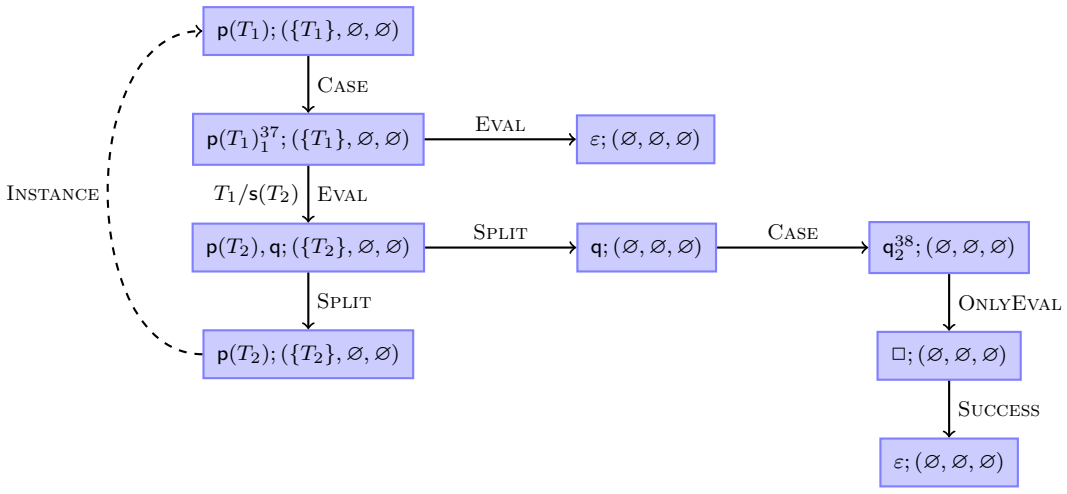
Third, if $\mathcal{V}(t') \not\subseteq \mathcal{G} \cup \mathcal{F}$, the answer substitution μ' can potentially instantiate any non-ground term in $Q\gamma$ except for variables from $\mathcal{F}(Q) \setminus \mathcal{F}(t')$. We define γ' in such a way that $\gamma\mu' = \mu\gamma'$ and $\gamma|_{\mathcal{A}(t)\cup\mathcal{A}(Q)} = \gamma'|_{\mathcal{A}(t)\cup\mathcal{A}(Q)}$. This is always possible because $Dom(\mu') \cap (\mathcal{F} \setminus \mathcal{F}(t')) = \varnothing$ and all variables in $Range(\mu)$ are fresh. Then, clearly, $Q\gamma\mu' = Q\mu\gamma'$. We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\mu)$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(s,s')\in\mathcal{U}\mu} s\gamma' \not\approx s'\gamma'$.

All these properties except for $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ are shown in [Sch08].

We perform a case analysis based on the partition $\mathcal{G}' = \mathcal{G} \uplus (ApproxGnd(t', \mu) \setminus \mathcal{G})$. For $a \in \mathcal{G}$ we have effectively defined $a\gamma' = a\gamma$ and thus $a\gamma' \in GroundTerms(\Sigma)$. For $a \in ApproxGnd(t', \mu) \setminus \mathcal{G}$ by definition of $ApproxGnd$ and equality of $\gamma\mu'$ and $\mu\gamma'$ we again know that $a\gamma' \in GroundTerms(\Sigma)$. □

Now we can close any termination graph by appropriately applying the rules Generalization, Instance, Parallel and Split.

Example 3.52. Consider again the Prolog program from Example 3.49. Now, consider the graph built using the rules from Parts 1 – 6:



Thus, with the help of the Split rule, we can easily close the graph.

We will show more example graphs in Chapter 6 and Chapter 7.

3.4 Summary

We have introduced the structures and operations for the construction of termination graphs used to handle cuts in logic programming as given in [Sch08]. We extended this method to fully handle meta-programming as defined in the ISO standard for Prolog and unification without occurs-check as used in most Prolog implementations. Furthermore, we introduced additional operations to handle instantiation errors and errors due to undefined predicate calls in Prolog. Additionally, we improved the automation and precision of the operations used for this approach. Finally, we illustrated the capabilities of this approach with a number of examples.

4 Built-in Predicates

Up to now, we can handle the two built-in predicates call/1 and !/0. However, real Prolog applications make use of many more built-in predicates as offered by the ISO standard. While some of these predicates can be expressed by clauses and, thus, could be handled by adding default definitions for them to a program, most of them cause side effects not directly expressible by clauses. Thus, we refrain from adding default clauses to programs and introduce additional inference rules for built-in predicates instead. There are 26 built-in predicates we can handle directly in our approach as it is. For the remaining built-in predicates we will show problems occurring with this approach and possibilities how this framework can be extended to solve these problems. As some results of the preceding chapter depend on the set of inference rules used in this approach, we also adapt the corresponding proofs to the extended rule set.

Structure of the Chapter

We group the built-in predicates according to their thematic classification in the ISO standard. We start with the built-in predicates for logic and control in Section 4.1. Except for catch/3 we can handle all built-in predicates belonging to this classification.

We continue in Section 4.2 by introducing additional inference rules for two built-in predicates for term comparison. In contrast to unification, these predicates test for term equality and inequality respectively.

In Section 4.3 we add inference rules for the very commonly used built-in predicates for term unification. While these predicates have a similar effect to the Eval rule from Chapter 3, they can be used for any pair of terms and not only for unification with clause heads. However, they do not introduce new terms to the current goal.

While Prolog is an untyped language, the ISO standard offers some built-in predicates for type testing. These predicates can for example be used to check inputs given by the user. Most of them are handled by the inference rules in Section 4.4.

Finally, we will handle some special cases of built-in predicates used for term or character output in Section 4.5. We exploit the default configuration for the input and output environment as defined in the ISO standard to exclude error cases for these predicates.

After introducing new inference rules to handle some built-in predicates, we show where there are problems with the remaining built-in predicates in Section 4.6. We also give

some ideas to overcome these problems in extensions of the framework presented here.

Section 4.7 covers the adaptions necessary for the proofs from Chapter 3 to be still valid for the extended rule set.

The contributions of this chapter are summarized in Section 4.8.

4.1 Logic and Control

The built-in predicates for logic and control influence the control flow of Prolog programs. In fact, the two built-in predicates we have already considered in Chapter 3 belong to this classification, too. The cut operator removes some backtracking possibilities while the call/1 predicate allows for meta-programming and restricts the effect of cuts inside such a meta-call. The remaining built-in predicates for logic and control cause additional side effects to influence the control flow.

The predicate ,/2 is used to evaluate a conjunction of goals. Implicitly, we have already used conjunctions by defining clause bodies and goals as lists of terms. While it is always possible to flatten nested conjunctions to a list, we still need this predicate for conjunctions in meta-calls as the call/1 predicate has only one argument and we want to be able to use meta-calls for more than one term at once. The operation performed by this built-in predicate is simply to evaluate both of its arguments in sequence. Thus, the goal $,(t_1, t_2), Q$ is evaluated to t_1, t_2, Q. If we could omit the transformation of variables in predication positions to applications of the call/1 predicate to such variables, we could express the ,/2 predicate by the meta-clause $,(G1, G2) \leftarrow G1, G2$. But as the transformation restricts the scope of cuts for such meta-clauses, this is in fact not equivalent.

Example 4.1. The goal $,(,(\mathsf{p}, \mathsf{q}), \mathsf{r})$ is first evaluated to the goal $,(\mathsf{p}, \mathsf{q}), \mathsf{r}$ and then to the goal $\mathsf{p}, \mathsf{q}, \mathsf{r}$. The concrete inference rule Conjunction will just perform this evaluation for ,/2.

For disjunctions, we have the predicate ;/2. However, this predicate is ambiguous, since it is also used in combination with ->/2 to express a conditional execution corresponding to if-then-else in imperative programming languages. For normal disjunctions, the first argument of ;/2 must not have ->/2 as its root symbol. Then this predicate is evaluated as follows. For a goal $;(t_1, t_2), Q$ we first try to evaluate t_1. Each time this succeeds with answer substitution σ, we continue to evaluate $Q\sigma$. When the evaluation of t_1 fails, we continue with evaluating the goal t_2, Q. Thus, we evaluate the complete goal to $t_1, Q \mid t_2, Q$. Again, without the transformation we could express the effect of this predicate by the two meta-clauses $;(G1, G2) \leftarrow G1$ and $;(G1, G2) \leftarrow G2$. However, due to the transformation for variables in predication positions this is not equivalent.

Example 4.2. Consider the Prolog program consisting of three facts for p, q and r. The goal ;(p, ;(q, r)) is evaluated to the list of goals p | ;(q, r) in one step. After evaluation of p, the goal ;(q, r) is further evaluated to q | r. The concrete inference rule DISJUNCTION will just perform this evaluation for ;/2.

The built-in predicate fail/0 is evaluated by failing directly. It is for example used to express negation-as-failure. This time, we can really express the effect of fail/0 by the two clauses fail ← failing(a) and failing(b) ← □ where failing/1 is a fresh function symbol. Thus, its effect in our setting is just to drop the current state element and backtrack to the next one.

Example 4.3. Consider again the Prolog program consisting of three facts for p, q and r. The list of goals p, fail, q | r is then first evaluated to fail, q | r and next to r which is finally evaluated to □. The concrete inference rule FAIL will just perform this evaluation for fail/0.

The two predicates halt/0 and halt/1 are used to force termination of the execution directly. While the former predicate just stops the execution, the latter can give some information to the Prolog processor (cf. [DEC96]) executing the Prolog program. Additionally, the latter might throw an error if its argument is not properly instantiated. As we do not consider the implementation dependent behavior after executing a Prolog goal in this thesis and do not cover error handling other than terminating directly, we can just assume that the computation stops in both cases without further operations. Hence we reach the empty state by evaluating one of these predicates. The same is true for the throw/1 predicate. It is used to generate an error which will lead to direct termination in our setting, too.[10]

Example 4.4. The list of goals halt, p, q | r evaluates directly to the empty goal list ε. The same is true in our setting for the evaluation of halt/1 and throw/1. The concrete inference rules HALT, HALT1 and THROW will just perform this evaluation for halt/0, halt/1 and throw respectively.

For conditional executions corresponding to if-then or if-then-else constructs in imperative programming languages, we have the ->/2 predicate. If it is the root symbol of the current goal, it corresponds to an if-then construct. Its first argument is evaluated and if this succeeds, its second argument is evaluated afterwards. Otherwise it fails. Cuts reached in the evaluation of the first argument have no effect for the evaluation of the second argument (or following evaluations). Also, the first argument is only evaluated once. Further solutions are not considered. Thus, the goal $->(t_1, t_2), Q$ is evaluated to $\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m$. Note that this predicate is not equivalent to an implication, because

[10] See [DEC96] and Section 4.6 for more information about the catch/3 predicate which would be capable of a more complex handling of errors.

it fails when its first argument cannot be shown. If the ->/2 predicate is used in the context of the ;/2 predicate as its first argument, i.e., we have the goal $;(->(t_1, t_2), t_3), Q$, this corresponds to an if-then-else construct for imperative programming languages. We first evaluate t_1. If this succeeds with answer substitution σ, we discard the evaluation of t_3 and continue with the evaluation of $t_2\sigma, Q\sigma$. If the evaluation of t_1 fails, we continue with t_3, Q instead. Again, t_1 is only evaluated once and cuts reached during its evaluation do not have any effect for the evaluation of $t_2\sigma, Q\sigma$ or t_3, Q respectively. Thus, we evaluate the complete goal to $\mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m$.

Example 4.5. The goal ->(p, q) is evaluated to $\mathsf{call}(\mathsf{p}), !_1, \mathsf{q} \mid ?_1$ while the goal ;(->(q, q), r) is evaluated to $\mathsf{call}(\mathsf{p}), !_1, \mathsf{q} \mid \mathsf{r} \mid ?_1$. The concrete inference rules IfThen and IfThenElse will just perform these evaluations for ->/2 and the combination of ;/2 and ->/2 respectively.

Negation-as-failure can be used by the built-in predicate \+/1. It tries to evaluate its argument and if this fails, the predicate succeeds. Otherwise it fails. Its effect can in fact be expressed by the two clauses $\backslash+(X) \leftarrow \mathsf{call}(X), !, \mathsf{fail}$ and $\backslash+(X) \leftarrow \square$, since the evaluation of its argument is not transparent for cuts. Thus, in our setting we evaluate $\backslash+(t), Q$ to $\mathsf{call}(t), !_m, \mathsf{fail} \mid Q \mid ?_m$. Note that this is not equivalent to logical negation as failing to prove a goal does not necessarily imply that the goal is wrong.[11]

Example 4.6. Consider the Prolog program only consisting of two facts $\mathsf{p} \leftarrow \square$ and $\mathsf{failure}(\mathsf{b}) \leftarrow \square$. Then, the goal $\backslash+(\mathsf{failure}(\mathsf{a}))$ is first evaluated to $\mathsf{call}(\mathsf{failure}(\mathsf{a})), !_1, \mathsf{fail} \mid \square \mid ?_1$, second to $\mathsf{failure}(\mathsf{a}), !_1, \mathsf{fail} \mid ?_2 \mid \square \mid ?_1$, over the third and fourth step it is backtracked to $?_3 \mid ?_2 \mid \square \mid ?_1$ where it reaches $\square \mid ?_1$ in two steps by the Failure rule. Hence, the original goal has a successful concrete state-derivation. Now consider the goal $\backslash+(\mathsf{p})$. Here, the evaluation is as follows. First, we reach $\mathsf{call}(\mathsf{p}), !_1, \mathsf{fail} \mid \square \mid ?_1$, second, $\mathsf{p}, !_1, \mathsf{fail} \mid ?_2 \mid \square \mid ?_1$. Then we succeed in proving p over the third and fourth step to $!_1, \mathsf{fail} \mid ?_3 \mid ?_2 \mid \square \mid ?_1$. Now we reach $\mathsf{fail} \mid ?_1$ by the Cut rule which evaluates to the empty state. Therefore, the concrete state-derivation for the original goal is not successful. The concrete inference rule Not will just perform this evaluation for \+/1.

The built-in predicate once/1 performs a special meta-call where only the first solution for its argument is considered. In other words we evaluate the goal $\mathsf{once}(t), Q$ to $\mathsf{call}(,(t, !)), Q$. This predicate can especially be used to analyze existential termination by transforming the original goal into an application of once/1 to this goal. Of course, this predicate can be expressed by the clause $\mathsf{once}(X) \leftarrow \mathsf{call}(X, !)$.

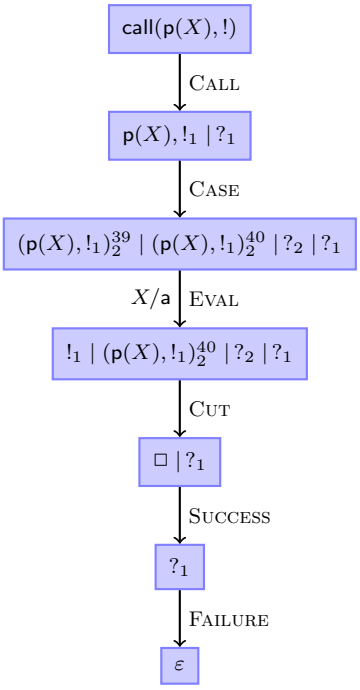
[11] See for example [Cla78] for more information about the mathematical properties of negation-as-failure.

Example 4.7. Consider the following Prolog program only consisting of two facts

$$\mathsf{p}(\mathsf{a}) \leftarrow \square. \tag{39}$$

$$\mathsf{p}(\mathsf{b}) \leftarrow \square. \tag{40}$$

and the query $\mathsf{once}(\mathsf{p}(X))$. This goal first evaluates to $\mathsf{call}(\mathsf{p}(X), !)$. From here, we obtain the following concrete state-derivation.



Thus, we only have one answer substitution $[X/\mathsf{a}]$ instead of two for the query $\mathsf{p}(X)$. The concrete inference rule Once will just perform this evaluation for $\mathsf{once}/1$.

The last two predicates can again be expressed by normal Prolog clauses. $\mathsf{repeat}/0$ is a predicate which just succeeds infinitely often. It can be expressed with the two clauses $\mathsf{repeat} \leftarrow \square$ and $\mathsf{repeat} \leftarrow \mathsf{repeat}$. Thus, we evaluate the goal repeat, Q to $Q \mid \mathsf{repeat}, Q$. The $\mathsf{true}/0$ predicate just succeeds once. It can be expressed by the clause $\mathsf{true} \leftarrow \square$. Thus, we can just drop it from the list of terms in the current goal.

Example 4.8. Consider the Prolog program consisting of only one fact $\mathsf{p} \leftarrow \square$. The goal $\mathsf{true}, \mathsf{repeat}, \mathsf{p}$ first evaluates to $\mathsf{repeat}, \mathsf{p}$ which is further evaluated to $\mathsf{p} \mid \mathsf{repeat}, \mathsf{p}$. After successful evaluation of p in three steps, we reach the goal $\mathsf{repeat}, \mathsf{p}$ again. Thus, instead of having one successful concrete state-derivation for p, we now have infinitely many. The concrete inference rules True and Repeat will just perform this evaluation for $\mathsf{true}/0$ and $\mathsf{repeat}/0$ respectively.

Altogether, we add the following concrete inference rules to handle the built-in predicates for logic and control.

Definition 4.9 (Concrete Inference Rules for Logic and Control)**.**

$$\frac{,(t_1, t_2), Q \mid S}{t_1, t_2, Q \mid S}\ (\textsc{Conjunction})$$

$$\frac{;(t_1, t_2), Q \mid S}{t_1, Q \mid t_2, Q \mid S}\ (\textsc{Disjunction}) \quad \textit{where } root(t_1) \neq \text{->}/2$$

$$\frac{\mathsf{fail}, Q \mid S}{S}\ (\textsc{Fail}) \qquad \frac{\mathsf{halt}, Q \mid S}{\varepsilon}\ (\textsc{Halt}) \qquad \frac{\mathsf{halt}(t'), Q \mid S}{\varepsilon}\ (\textsc{Halt1})$$

$$\frac{\text{->}(t_1, t_2), Q \mid S}{\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S}\ (\textsc{IfThen}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{;(\text{->}(t_1, t_2), t_3), Q \mid S}{\mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m \mid S}\ (\textsc{IfThenElse}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{\backslash+(t'), Q \mid S}{\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S}\ (\textsc{Not}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{\mathsf{once}(t'), Q \mid S}{\mathsf{call}(,(t', !)), Q \mid S}\ (\textsc{Once}) \qquad \frac{\mathsf{repeat}, Q \mid S}{Q \mid \mathsf{repeat}, Q \mid S}\ (\textsc{Repeat}) \qquad \frac{\mathsf{throw}(t'), Q \mid S}{\varepsilon}\ (\textsc{Throw})$$

$$\frac{\mathsf{true}, Q \mid S}{Q \mid S}\ (\textsc{True})$$

For the above rules we use the additional conventions that t_1, t_2 and t_3 are arbitrary terms from $\mathit{PrologTerms}(\Sigma, \mathcal{V})$.

Example 4.10. Using for example the built-in predicate ,/2, we are now able to analyze Prolog programs where the transformation of variables to applications of call/1 to the same variables really makes a difference. Consider the following Prolog program $\mathcal{P}$

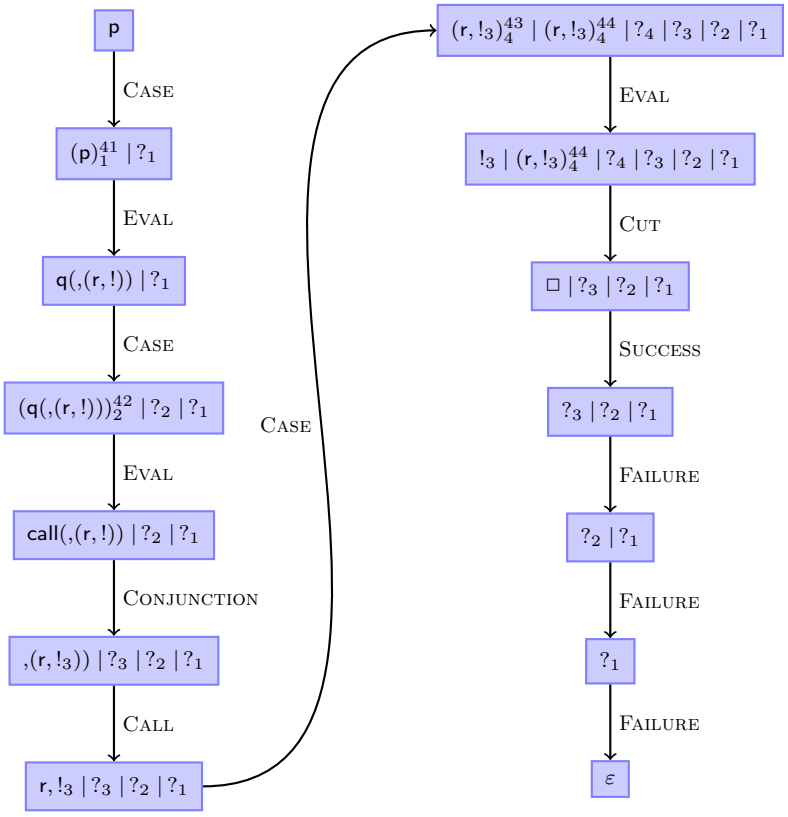
$$\mathsf{p} \leftarrow \mathsf{q}(,(\mathsf{r}, !)). \tag{41}$$

$$\mathsf{q}(X) \leftarrow \mathsf{call}(X). \tag{42}$$

$$\mathsf{r} \leftarrow \square. \tag{43}$$

$$\mathsf{r} \leftarrow \mathsf{r}. \tag{44}$$

and the query set $\mathcal{Q}$ consisting of the single query p. This program is terminating w.r.t. $\mathcal{Q}$. This can be seen by the following concrete state-derivation.



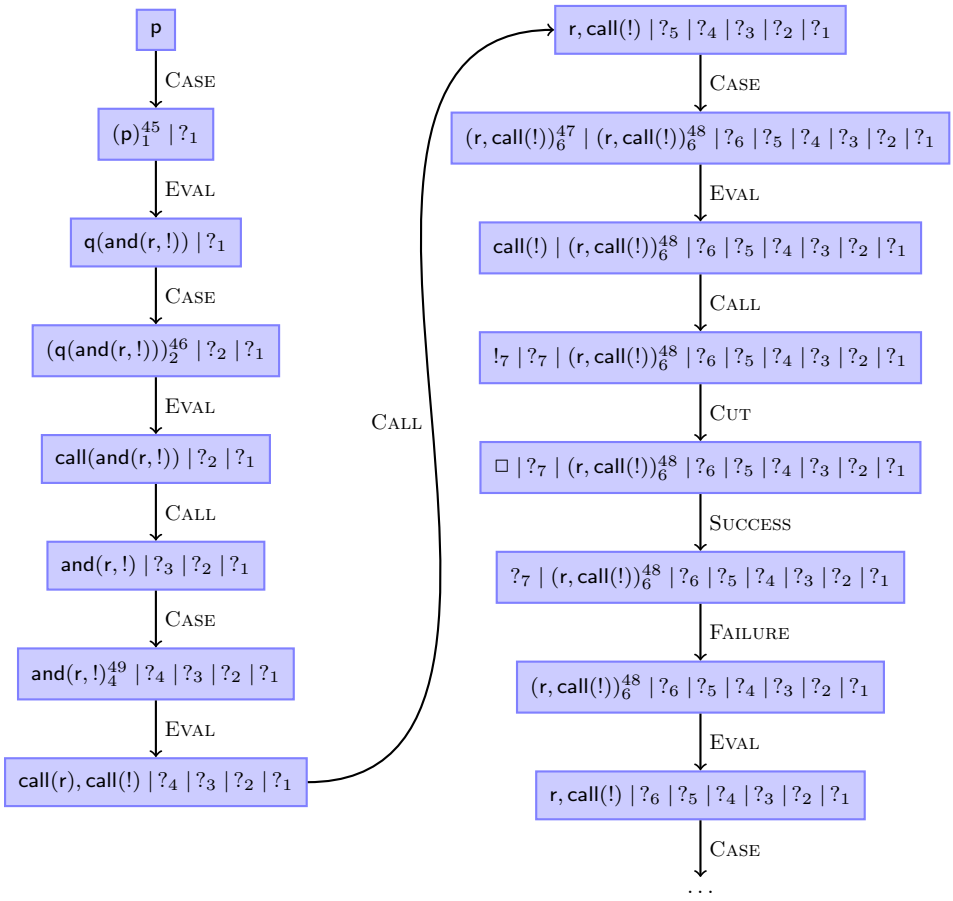
Now, consider an alternative Prolog program $\mathcal{P}'$ where we define our own predicate and/2 for conjunctions.

$$\mathsf{p} \leftarrow \mathsf{q}(\mathsf{and}(\mathsf{r},!)). \tag{45}$$
$$\mathsf{q}(X) \leftarrow \mathsf{call}(X). \tag{46}$$
$$\mathsf{r} \leftarrow \square. \tag{47}$$
$$\mathsf{r} \leftarrow \mathsf{r}. \tag{48}$$
$$\mathsf{and}(X,Y) \leftarrow \mathsf{call}(X), \mathsf{call}(Y). \tag{49}$$

We still regard the query p.

Note that even if we syntactically define and/2 by the clause $\mathsf{and}(X,Y) \leftarrow X,Y$, this definition will be transformed into the one used in $\mathcal{P}'$ by an ISO standard conforming Prolog processor (cf. [DEC96]). Hence, we also see that we are not able to express the real behavior of the built-in predicate ,/2 by clauses.

We now obtain a non-terminating concrete state-derivation for $\mathcal{Q}$.



Example 4.11. Consider again the Prolog program add3.pl from Example 3.2. Using built-in predicates, we obtain the following equivalent, but shorter program $\mathcal{P}$

$$\mathsf{add}(X, 0, X) \leftarrow \square. \tag{50}$$
$$\mathsf{add}(X, Y, \mathsf{s}(Z)) \leftarrow \backslash+(\mathsf{isZero}(Y)), \mathsf{p}(Y, P), \mathsf{add}(X, P, Z). \tag{51}$$
$$\mathsf{p}(0, 0) \leftarrow \square. \tag{52}$$
$$\mathsf{p}(\mathsf{s}(X), X) \leftarrow \square. \tag{53}$$
$$\mathsf{isZero}(0) \leftarrow \square. \tag{54}$$

with the same query set $\mathcal{Q}$ as for Example 3.2.

Since all built-in predicates for logic and control have only empty answer substitutions, we do not need to update the knowledge base when applying an abstract inference rule for one of these predicates. Therefore, the abstract inference rules for built-in predicates for logic and control are straightforward to define.

Definition 4.12 (Abstract Inference Rules for Logic and Control)**.**

$$\frac{,(t_1, t_2), Q \mid S; KB}{t_1, t_2, Q \mid S; KB}\ (\textsc{Conjunction})$$

$$\frac{;(t_1, t_2), Q \mid S; KB}{t_1, Q \mid t_2, Q \mid S; KB}\ (\textsc{Disjunction}) \quad \textit{where } root(t_1) \neq \text{->}/2 \textit{ and } t_1 \notin \mathcal{A}$$

$$\frac{\mathsf{fail}, Q \mid S; KB}{S; KB}\ (\textsc{Fail}) \qquad \frac{\mathsf{halt}, Q \mid S; KB}{\varepsilon; KB}\ (\textsc{Halt}) \qquad \frac{\mathsf{halt}(t'), Q \mid S; KB}{\varepsilon; KB}\ (\textsc{Halt1})$$

$$\frac{\text{->}(t_1, t_2), Q \mid S; KB}{\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S; KB}\ (\textsc{IfThen}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{;(\text{->}(t_1, t_2), t_3), Q \mid S; KB}{\mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m \mid S; KB}\ (\textsc{IfThenElse}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{\backslash+(t'), Q \mid S; KB}{\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S; KB}\ (\textsc{Not}) \quad \textit{for a fresh } m \in \mathbb{N}$$

$$\frac{\mathsf{once}(t'), Q \mid S; KB}{\mathsf{call}(,(t', !)), Q \mid S; KB}\ (\textsc{Once}) \qquad \frac{\mathsf{repeat}, Q \mid S; KB}{Q \mid \mathsf{repeat}, Q \mid S; KB}\ (\textsc{Repeat})$$

$$\frac{\mathsf{throw}(t'), Q \mid S; KB}{\varepsilon; KB}\ (\textsc{Throw}) \qquad \frac{\mathsf{true}, Q \mid S; KB}{Q \mid S; KB}\ (\textsc{True})$$

As we do not mandate changes to the knowledge base, these rules can easily be proved to be sound.

Lemma 4.13 (Soundness of Abstract Inference Rules for Logic and Control)**.** *The rules* Conjunction*,* Disjunction*,* Fail*,* Halt*,* Halt1*,* IfThen*,* IfThenElse*,* Not*,* Once*,* Repeat*,* Throw *and* True *from Definition 4.12 are sound.*

Proof. For Conjunction assume there is an infinite concrete state-derivation from $,(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(,(t_1, t_2), Q \mid S; KB)$. As the only applicable concrete inference rule is Conjunction, we reach the state $t_1\gamma, t_2\gamma, Q\gamma \mid S\gamma \in \mathcal{CON}(t_1, t_2, Q \mid S; KB)$ having an infinite concrete state-derivation.

For Disjunction assume there is an infinite concrete state-derivation from $;(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(;(t_1, t_2), Q \mid S; KB)$, where $root(t_1) \neq \text{->}/2$ and $t_1 \notin \mathcal{A}$. As γ can only replace abstract variables, we have $root(t_1\gamma) \neq \text{->}/2$ and the only applicable concrete inference rule is Disjunction leading to the state $t_1\gamma, Q\gamma \mid t_2\gamma, Q\gamma \mid S\gamma \in$

$\mathcal{CON}(t_1, Q \mid t_2, Q \mid S; KB)$ having an infinite concrete state-derivation.

For Fail assume there is an infinite concrete state-derivation from $\mathsf{fail}, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{fail}, Q \mid S; KB)$. As the only applicable concrete inference rule is Fail, we reach the state $S\gamma \in \mathcal{CON}(S; KB)$ starting an infinite concrete state-derivation.

For Halt assume there is an infinite concrete state-derivation from $\mathsf{halt}, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{halt}, Q \mid S; KB)$. As the only applicable concrete inference rule is Halt, we reach the state ε in contradiction to the assumption that we have an infinite concrete state-derivation. Thus, Halt is trivially sound.

For Halt1 assume there is an infinite concrete state-derivation from $\mathsf{halt}(t')\gamma, Q\gamma \mid S\gamma = \mathsf{halt}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{halt}(t'), Q \mid S; KB)$. As the only applicable concrete inference rule is Halt1, we reach the state ε in contradiction to the assumption that we have an infinite concrete state-derivation. Thus, Halt1 is trivially sound.

For IfThen assume there is an infinite concrete state-derivation from $\text{->}(t_1, t_2)\gamma, Q\gamma \mid S\gamma = \text{->}(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\text{->}(t_1, t_2), Q \mid S; KB)$. As the only applicable concrete inference rule is IfThen, we reach the state $\mathsf{call}(t_1\gamma), !_m, t_2\gamma, Q\gamma \mid ?_m \mid S\gamma = \mathsf{call}(t_1)\gamma, !_m, t_2\gamma, Q\gamma \mid ?_m \mid S\gamma \in \mathcal{CON}(\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S; KB)$ which starts an infinite concrete state-derivation.

For IfThenElse assume there is an infinite concrete state-derivation from $;(\text{->}(t_1, t_2), t_3)\gamma, Q\gamma \mid S\gamma = ;(\text{->}(t_1\gamma, t_2\gamma), t_3\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(;(\text{->}(t_1, t_2), t_3), Q \mid S; KB)$. Since the only applicable concrete inference rule is IfThenElse, we reach the state $\mathsf{call}(t_1\gamma), !_m, t_2\gamma, Q\gamma \mid t_3\gamma, Q\gamma \mid ?_m \mid S\gamma = \mathsf{call}(t_1)\gamma, !_m, t_2\gamma, Q\gamma \mid t_3\gamma, Q\gamma \mid ?_m \mid S\gamma \in \mathcal{CON}(\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S; KB)$ which starts an infinite concrete state-derivation.

For Not assume there is an infinite concrete state-derivation from $\backslash\text{+}(t')\gamma, Q\gamma \mid S\gamma = \backslash\text{+}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash\text{+}(t'), Q \mid S; KB)$. As the only applicable concrete inference rule is Not, we reach the state $\mathsf{call}(t'\gamma), !_m, \mathsf{fail} \mid Q\gamma \mid ?_m \mid S\gamma = \mathsf{call}(t')\gamma, !_m, \mathsf{fail} \mid Q\gamma \mid ?_m \mid S\gamma \in \mathcal{CON}(\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S; KB)$ having an infinite concrete state-derivation.

For Once assume there is an infinite concrete state-derivation from $\mathsf{once}(t')\gamma, Q\gamma \mid S\gamma = \mathsf{once}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{once}(t'), Q \mid S; KB)$. As the only applicable concrete inference rule is Once, we reach the state $\mathsf{call}(,(t'\gamma, !)), Q\gamma \mid S\gamma = \mathsf{call}(,(t', !))\gamma, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{call}(,(t', !)), Q \mid S; KB)$ starting an infinite concrete state-derivation.

For Repeat assume there is an infinite concrete state-derivation from $\mathsf{repeat}, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{repeat}, Q \mid S; KB)$. As the only applicable concrete inference rule is Repeat, we reach the state $Q\gamma \mid \mathsf{repeat}, Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid \mathsf{repeat}, Q \mid S; KB)$ starting an infinite concrete state-derivation.

For Throw assume there is an infinite concrete state-derivation from $\mathsf{throw}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{throw}(t'), Q \mid S; KB)$. As the only applicable concrete inference rule is Throw, we reach the state ε in contradiction to the assumption that we have an infinite concrete state-derivation. Thus, Throw is trivially sound.

For True assume there is an infinite concrete state-derivation from $\mathsf{true}, Q\gamma \mid S\gamma \in$

$\mathcal{CON}(\mathsf{true}, Q \mid S; KB)$. As the only applicable concrete inference rule is TRUE, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ starting an infinite concrete state-derivation. □

4.2 Term Comparison

The next two built-in predicates are used for term comparison. In contrast to unification, they test for equality and inequality of terms respectively. So even two variables with different names are not equal.

Example 4.14. The goal $==(\mathsf{p}(X), \mathsf{p}(X))$ succeeds while the goal $==(\mathsf{p}(X), \mathsf{p}(Y))$ fails. Conversely, the goal $\backslash==(\mathsf{p}(X), \mathsf{p}(X))$ fails while the goal $\backslash==(\mathsf{p}(X), \mathsf{p}(Y))$ succeeds.

As these predicates have only empty answer substitutions, their rules are easily defined by dropping the first term of the current goal in case of a successful test and backtracking to the next state element otherwise.

Definition 4.15 (Concrete Inference Rules for Term Comparison)**.**

$$\frac{==(t_1, t_1), Q \mid S}{Q \mid S}\ (\textsc{EqualsSuccess})$$

$$\frac{==(t_1, t_2), Q \mid S}{S}\ (\textsc{EqualsFail}) \quad \textit{where } t_1 \neq t_2$$

$$\frac{\backslash==(t_1, t_2), Q \mid S}{Q \mid S}\ (\textsc{UnequalsSuccess}) \quad \textit{where } t_1 \neq t_2$$

$$\frac{\backslash==(t_1, t_1), Q \mid S}{S}\ (\textsc{UnequalsFail})$$

Now, for the abstract case, we have to consider that different abstract variables may still represent equal terms. If we compare two equal terms, the abstract inference rules are straightforward. This is the case for EQUALSSUCCESS and UNEQUALSFAIL. For the remaining cases we try to unify the terms. If they are not even unifiable, they cannot represent equal terms and, hence we can apply the abstract EQUALSFAIL or UNEQUALSSUCCESS rule. This is also true if every unifier of the two terms must instantiate non-abstract variables as these variables just represent themselves and no other terms. If the two terms are unifiable by only instantiating abstract variables, they may or may not represent equal terms. Thus, we consider two successor states in such cases. If the represented terms are equal we can replace them with their equal instances by their mgu σ where we demand that σ has only fresh abstract variables in its range to keep the generality of the abstract

state. If we replace variables from $\mathcal{G}$ by σ, we update the knowledge base by adding the abstract variables in the range of $\sigma|_{\mathcal{G}}$ to $\mathcal{G}$. Since we know that the represented terms are equal, σ corresponds to a shape analysis and not to an instantiation of non-abstract variables. Thus, we can instantiate the abstract variables in $\mathcal{U}$ and the backtracking list by σ, too. If the represented terms are not equal, we simply succeed or fail respectively without applying any substitutions. This amounts to the following abstract inference rules for term comparison.

Definition 4.16 (Abstract Inference Rules for Term Comparison)**.**

$$\frac{==(t_1, t_1), Q \mid S; KB}{Q \mid S; KB} \text{ (EqualsSuccess)}$$

$$\frac{==(t_1, t_2), Q \mid S; KB}{S; KB} \text{ (EqualsFail)} \quad \begin{array}{l} \textit{where } t_1 \nsim t_2 \textit{ or } \forall\sigma \textit{ with } t_1\sigma = \\ t_2\sigma \textit{ we have } Dom(\sigma) \cap \mathcal{N} \neq \varnothing \end{array}$$

$$\frac{==(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q\sigma, \mid S\sigma; (\mathcal{G}', \mathcal{F}, \mathcal{U}') \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})} \text{ (EqualsCase)}$$

where $t_1 \neq t_2$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{U}' = \mathcal{U}\sigma$ *and* $mgu(t_1, t_2) = \sigma$ *with* $Dom(\sigma) \subseteq \mathcal{A}$ *and* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$

$$\frac{\backslash==(t_1, t_2), Q \mid S; KB}{Q \mid S; KB} \text{ (UnequalsSuccess)} \quad \begin{array}{l} \textit{where } t_1 \nsim t_2 \textit{ or } \forall\sigma \\ \textit{with } t_1\sigma = t_2\sigma \textit{ we} \\ \textit{have } Dom(\sigma) \cap \mathcal{N} \neq \varnothing \end{array}$$

$$\frac{\backslash==(t_1, t_1), Q \mid S; KB}{S; KB} \text{ (UnequalsFail)}$$

$$\frac{\backslash==(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q, \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}) \qquad S\sigma; (\mathcal{G}', \mathcal{F}, \mathcal{U}')} \text{ (UnequalsCase)}$$

where $t_1 \neq t_2$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{U}' = \mathcal{U}\sigma$ *and* $mgu(t_1, t_2) = \sigma$ *with* $Dom(\sigma) \subseteq \mathcal{A}$ *and* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$

The soundness of the rules where we can clearly determine whether the represented terms are equal or not is comparatively easy to prove.

Lemma 4.17 (Soundness of EqualsSuccess, EqualsFail, UnequalsSuccess and UnequalsFail)**.** *The rules* EqualsSuccess*,* EqualsFail*,* UnequalsSuccess *and* UnequalsFail *from Definition 4.16 are sound.*

Proof. For EqualsSuccess assume there is an infinite concrete state-derivation from $==(t_1\gamma, t_1\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(==(t_1, t_1), Q \mid S; KB)$. Since $t_1\gamma = t_1\gamma$ we know that the only applicable concrete inference rule is EqualsSuccess and we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation.

For EqualsFail assume there is an infinite concrete state-derivation from $==(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(==(t_1, t_2), Q \mid S; KB)$ where $t_1 \nsim t_2$ or $\forall\sigma$ with $t_1\sigma = t_2\sigma$ we have $Dom(\sigma) \cap \mathcal{N} \neq \varnothing$. If $t_1 \nsim t_2$, we obtain $t_1\gamma \neq t_2\gamma$ and the only applicable concrete inference rule is EqualsFail leading to the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation. So let $t_1 \sim t_2$ with $\forall\sigma$ with $t_1\sigma = t_2\sigma$ we have $Dom(\sigma) \cap \mathcal{N} \neq \varnothing$. As $Dom(\gamma) \subseteq \mathcal{A}$, we obtain $t_1\gamma \neq t_2\gamma$ and we reach the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation again.

For UnequalsSuccess assume there is an infinite concrete state-derivation from $\backslash==(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash==(t_1, t_2), Q \mid S; KB)$ where $t_1 \nsim t_2$ or $\forall\sigma$ with $t_1\sigma = t_2\sigma$ we have $Dom(\sigma) \cap \mathcal{N} \neq \varnothing$. If $t_1 \nsim t_2$, we obtain $t_1\gamma \neq t_2\gamma$ and the only applicable concrete inference rule is UnequalsSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation. So let $t_1 \sim t_2$ with $\forall\sigma$ with $t_1\sigma = t_2\sigma$ we have $Dom(\sigma) \cap \mathcal{N} \neq \varnothing$. As $Dom(\gamma) \subseteq \mathcal{A}$, we obtain $t_1\gamma \neq t_2\gamma$ and we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation again.

For UnequalsFail assume there is an infinite concrete state-derivation from $\backslash==(t_1\gamma, t_1\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash==(t_1, t_1), Q \mid S; KB)$. Since $t_1\gamma = t_1\gamma$ we know that the only applicable concrete inference rule is UnequalsFail and we reach the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation. □

Now we prove the soundness of the EqualsCase rule, where we must consider the two cases depending on whether the represented terms are equal and possible replacements by the mgu σ.

Lemma 4.18 (Soundness of EqualsCase)**.** *The rule* EqualsCase *from Definition 4.16 is sound.*

Proof. Assume there is an infinite concrete state-derivation from $==(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(==(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $t_1 \neq t_2$ and $mgu(t_1, t_2) = \sigma$ with $Dom(\sigma) \subseteq \mathcal{A}$ and $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$.

If $t_1\gamma \neq t_2\gamma$, the only applicable concrete inference rule is EqualsFail leading to the state $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ which starts an infinite concrete state-derivation.

So let $t_1\gamma = t_2\gamma$. Then the only applicable concrete inference rule is EqualsSuccess and we reach the state $Q\gamma \mid S\gamma$ having an infinite concrete state-derivation. As $mgu(t_1, t_2) = \sigma$ and γ is a unifier of t_1 and t_2, we know that there is a δ with $\gamma = \sigma\delta$. We define γ' by $x\gamma' = x\sigma\delta$ for $x \in Dom(\sigma)$ and $x\gamma' = x\delta$ otherwise. As $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{A}_{fresh}$ we obtain $\gamma = \sigma\delta = \sigma\gamma' = \gamma'$. Hence, we are left to show that γ is a concretiza-

tion w.r.t. $(\mathcal{G}', \mathcal{F}, \mathcal{U}')$, i.e., $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$, $\mathit{Range}(\gamma|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$, and $\bigwedge_{(t,t') \in \mathcal{U}'} t\gamma \not\approx t'\gamma$.

We obtain $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ and $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$ directly by the fact that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$.

For $\mathit{Range}(\gamma|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$ we perform a case analysis over $a \in \mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}})) \uplus \mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$. If $a \in \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$ there is a variable $a' \in \mathcal{G}$ with $a \in \mathcal{A}(a'\sigma)$. As $a'\gamma \in \mathit{GroundTerms}(\Sigma)$ and $\sigma\gamma = \gamma$ we obtain $a\gamma \in \mathit{GroundTerms}(\Sigma)$. So let $a \in \mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$. Then we know $a\gamma \in \mathit{GroundTerms}(\Sigma)$ by $\mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}})) \subseteq \mathcal{G}$.

As we also know $\bigwedge_{(t,t') \in \mathcal{U}} t\gamma \not\approx t'\gamma$, we obtain $\bigwedge_{(t,t') \in \mathcal{U}'} t\gamma \not\approx t'\gamma$ by the fact that $\sigma\gamma = \gamma$ and $\mathcal{U}' = \mathcal{U}\sigma$.

Thus, we have $Q\gamma \mid S\gamma \in \mathcal{CON}(Q\sigma, \mid S\sigma; (\mathcal{G}', \mathcal{F}, \mathcal{U}'))$. □

Likewise, we prove the soundness of UnequalsCase.

Lemma 4.19 (Soundness of UnequalsCase)**.** *The rule* UnequalsCase *from Definition 4.16 is sound.*

Proof. Assume there is an infinite concrete state-derivation from $\backslash\!\!=\!\!=(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash\!\!=\!\!=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $t_1 \neq t_2$ and $\mathit{mgu}(t_1, t_2) = \sigma$ with $\mathit{Dom}(\sigma) \subseteq \mathcal{A}$ and $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{A}_{\mathit{fresh}}$.

If $t_1\gamma \neq t_2\gamma$, the only applicable concrete inference rule is UnequalsSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ which starts an infinite concrete state-derivation.

So let $t_1\gamma = t_2\gamma$. Then the only applicable concrete inference rule is UnequalsFail and we reach the state $S\gamma$ having an infinite concrete state-derivation. As $\mathit{mgu}(t_1, t_2) = \sigma$ and γ is a unifier of t_1 and t_2, we know that there is a δ with $\gamma = \sigma\delta$. We define γ' by $x\gamma' = x\sigma\delta$ for $x \in \mathit{Dom}(\sigma)$ and $x\gamma' = x\delta$ otherwise. As $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{A}_{\mathit{fresh}}$ we obtain $\gamma = \sigma\delta = \sigma\gamma' = \gamma'$. Hence, we are left to show that γ is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}, \mathcal{U}')$, i.e., $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$, $\mathit{Range}(\gamma|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$, $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$, and $\bigwedge_{(t,t') \in \mathcal{U}'} t\gamma \not\approx t'\gamma$.

We obtain $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ and $\mathcal{F}(\mathit{Range}(\gamma)) = \varnothing$ directly by the fact that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$.

For $\mathit{Range}(\gamma|_{\mathcal{G}'}) \subseteq \mathit{GroundTerms}(\Sigma)$ we perform a case analysis over $a \in \mathcal{G}' = \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}})) \uplus \mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$. If $a \in \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$ there is a variable $a' \in \mathcal{G}$ with $a \in \mathcal{A}(a'\sigma)$. As $a'\gamma \in \mathit{GroundTerms}(\Sigma)$ and $\sigma\gamma = \gamma$ we obtain $a\gamma \in \mathit{GroundTerms}(\Sigma)$. So let $a \in \mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}}))$. Then we know $a\gamma \in \mathit{GroundTerms}(\Sigma)$ by $\mathcal{G}' \setminus \mathcal{A}(\mathit{Range}(\sigma|_{\mathcal{G}})) \subseteq \mathcal{G}$.

As we also know $\bigwedge_{(t,t') \in \mathcal{U}} t\gamma \not\approx t'\gamma$, we obtain $\bigwedge_{(t,t') \in \mathcal{U}'} t\gamma \not\approx t'\gamma$ by the fact that $\sigma\gamma = \gamma$ and $\mathcal{U}' = \mathcal{U}\sigma$.

Thus, we have $S\gamma \in \mathcal{CON}(S\sigma; (\mathcal{G}', \mathcal{F}, \mathcal{U}'))$. □

4.3 Term Unification

The built-in predicates for term unification belong to the most commonly used built-in predicates in Prolog. They try to unify their arguments and succeed or fail according to the unification. For =/2, the mgu is also the answer substitution in case the unification succeeds. Both built-in predicates for term unification can be expressed by the clauses $=(X, X) \leftarrow \square$ and $\backslash=(X, Y) \leftarrow \backslash+(=(X, Y))$.

Example 4.20. The goal $=(\mathsf{f}(X, \mathsf{b}), \mathsf{f}(\mathsf{a}, Y))$ succeeds with the answer substitution $[X/\mathsf{a}, Y\mathsf{b}]$ while the goal $=(\mathsf{f}(X), \mathsf{g}(Y))$ fails. Conversely, the goal $\backslash=(\mathsf{f}(X, \mathsf{b}), \mathsf{f}(\mathsf{a}, Y))$ fails and the goal $\backslash=(\mathsf{f}(X), \mathsf{g}(Y))$ succeeds with the empty answer substitution.

The concrete inference rules for these predicates are, therefore, given as follows.

Definition 4.21 (Concrete Inference Rules for Term Unification)**.**

$$\frac{=(t_1, t_2), Q \mid S}{Q\sigma \mid S}\ (\textsc{UnifySuccess}) \quad \textit{where } mgu(t_1, t_2) = \sigma$$

$$\frac{=(t_1, t_2), Q \mid S}{S}\ (\textsc{UnifyFail}) \quad \textit{where } t_1 \nsim t_2$$

$$\frac{\backslash=(t_1, t_2), Q \mid S}{Q \mid S}\ (\textsc{NoUnifySuccess}) \quad \textit{where } t_1 \nsim t_2$$

$$\frac{\backslash=(t_1, t_2), Q \mid S}{S}\ (\textsc{NoUnifyFail}) \quad \textit{where } t_1 \sim t_2$$

Example 4.22. Consider again the Prolog program divremain.pl from Example 3.1. Using built-in predicates, we obtain the following equivalent, but shorter program $\mathcal{P}$

$$\mathsf{div}(X, \mathsf{0}, Z, R) \leftarrow \mathsf{!}, \mathsf{fail}. \tag{55}$$
$$\mathsf{div}(\mathsf{0}, Y, Z, R) \leftarrow \mathsf{!}, =(Z, \mathsf{0}), =(R, \mathsf{0}). \tag{56}$$
$$\mathsf{div}(X, Y, \mathsf{s}(Z), R) \leftarrow \mathsf{minus}(X, Y, U), \mathsf{!}, \mathsf{div}(U, Y, Z, R). \tag{57}$$
$$\mathsf{div}(X, Y, \mathsf{0}, X) \leftarrow \square. \tag{58}$$
$$\mathsf{minus}(X, \mathsf{0}, X) \leftarrow \square. \tag{59}$$
$$\mathsf{minus}(\mathsf{s}(X), \mathsf{s}(Y), Z) \leftarrow \mathsf{minus}(X, Y, Z). \tag{60}$$

with the same query set $\mathcal{Q}$ as for Example 3.1.

Example 4.23. Consider again the Prolog program even.pl from Example 3.3. Using built-in predicates, we obtain the following equivalent, but shorter program $\mathcal{P}$

$$\mathsf{even}(X) \leftarrow =(Y, \mathsf{f}(\mathsf{e}, \mathsf{f}(\mathsf{o}, Y))), \mathsf{c}(Y, X). \tag{61}$$

$$\mathsf{c}(\mathsf{f}(\mathsf{e}, X), \mathsf{0}) \leftarrow \square. \tag{62}$$

$$\mathsf{c}(\mathsf{f}(Z, X), \mathsf{s}(Y)) \leftarrow \mathsf{c}(X, Y). \tag{63}$$

with the same query set $\mathcal{Q}$ as for Example 3.3.

Since the Backtrack, Eval and OnlyEval rules from Chapter 3 also try to perform a unification, the abstract inference rules for term unification are quite similar to these rules and have almost the same conditions. As for Backtrack, we know that we can apply the UnifyFail rule if the two terms do not unify or if their mgu contradicts information from the knowledge base. Then we just add the pair containing the two argument terms to $\mathcal{U}$ and backtrack to the next state element. Also, if the unification definitely succeeds as for OnlyEval, we can apply the UnifySuccess rule and just drop the first term in the current goal while applying the mgu σ (and possible variable refreshments due to sharing effects) to the remaining terms in the current goal and $\sigma|_{\mathcal{G}}$ to the following state elements and to $\mathcal{U}$. The set $\mathcal{G}$ is updated as for the OnlyEval rule, but for the set $\mathcal{F}$ there is a slight difference as we do not introduce free variables from a clause. Apart from this, the update for $\mathcal{F}$ still remains the same. The approximation *ApproxUnify* corresponds directly to *Approx* from Definition 3.27 except that the variables from both terms must be considered for the choice of the suitable substitution. UnifyCase is a combination of UnifySuccess and UnifyFail just like Eval is a combination of OnlyEval and Backtrack. It also has the negated conditions for the other two rules to make them non-overlapping.

Now, for the predicate \= used to test for non-unifiability, there is still a connection to the rules Backtrack, Eval and OnlyEval, but the connections are inverted and we do not apply the mgu in case of successful unification. Thus, the NoUnifySuccess rule corresponds to Backtrack and NoUnifyFail corresponds to OnlyEval. The difference is, that we just drop the first term in the current goal instead of the first state element for NoUnifySuccess and vice versa for NoUnifyFail. Likewise, the two successor states of NoUnifyCase correspond to the application of NoUnifySuccess and NoUnifyFail respectively. Again, their conditions are negated to obtain non-overlapping rules.

Taking everything into account, we define the abstract inference rules for term unification as follows.

Definition 4.24 (Abstract Inference Rules for Term Unification)**.**

$$\frac{=(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{Q\sigma' \mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})}\ (\textsc{UnifySuccess})$$

where $mgu(t_1,t_2)=\sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $Range(\sigma|_{\mathcal{A}}) \subseteq \mathcal{A}$, $\sigma|_{\mathcal{A}} : Dom(\sigma|_{\mathcal{A}}) \rightarrow Range(\sigma|_{\mathcal{A}})$ *is bijective,* $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$ *and* $\sigma' = ApproxUnify(\sigma,t_1,t_2,\mathcal{G},\mathcal{F})$

$$\frac{=(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{S;(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\})}\ (\textsc{UnifyFail})$$

where $t_1 \nsim t_2$ *or* $\sigma = mgu(t_1,t_2)$ *with* $\exists a \in \mathcal{G} : a\sigma \notin FinitePrologTerms(\Sigma,\mathcal{V})$ *or* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{G}})) \subseteq \mathcal{A}$ *and* $\exists (s,s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge Dom(\sigma') \subseteq \mathcal{F}$

$$\frac{=(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{Q\sigma' \mid S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}) \qquad S;(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\})}\ (\textsc{UnifyCase})$$

where $mgu(t_1,t_2)=\sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, $Range(\sigma|_{\mathcal{G}}) \subseteq FinitePrologTerms(\Sigma,\mathcal{A})$, $(Range(\sigma|_{\mathcal{A}}) \not\subseteq \mathcal{A} \vee \sigma|_{\mathcal{A}} : Dom(\sigma|_{\mathcal{A}}) \rightarrow Range(\sigma|_{\mathcal{A}})$ *is not bijective),* $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\forall (s,s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_{\mathcal{G}}\sigma'' = s'\sigma|_{\mathcal{G}}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{F})$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$, $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$ *and* $\sigma' = ApproxUnify(\sigma,t_1,t_2,\mathcal{G},\mathcal{F})$

$$\frac{\setminus=(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\})}\ (\textsc{NoUnifySuccess})$$

where $t_1 \nsim t_2$ *or* $\sigma = mgu(t_1,t_2)$ *with* $\exists a \in \mathcal{G} : a\sigma \notin FinitePrologTerms(\Sigma,\mathcal{V})$ *or* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{G}})) \subseteq \mathcal{A}$ *and* $\exists (s,s') \in \mathcal{U} : \sigma' = mgu(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge Dom(\sigma') \subseteq \mathcal{F}$

$$\frac{\setminus=(t_1,t_2),Q \mid S;(\mathcal{G},\mathcal{F},\mathcal{U})}{S\sigma|_{\mathcal{G}};(\mathcal{G}',\mathcal{F},\mathcal{U}\sigma|_{\mathcal{G}})}\ (\textsc{NoUnifyFail})$$

where $mgu(t_1,t_2)=\sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $Range(\sigma|_{\mathcal{A}}) \subseteq \mathcal{A}$, $\sigma|_{\mathcal{A}} : Dom(\sigma|_{\mathcal{A}}) \rightarrow Range(\sigma|_{\mathcal{A}})$ *is bijective,* $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ *and* $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$

$$\frac{\backslash\!=\!(t_1,t_2), Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U})}{Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\}) \qquad S\sigma|_{\mathcal{G}}; (\mathcal{G}',\mathcal{F},\mathcal{U}\sigma|_{\mathcal{G}})} \text{ (NoUnifyCase)}$$

where $mgu(t_1,t_2) = \sigma$ *with* $\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}$, $\mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}$, $Range(\sigma|_{\mathcal{G}}) \subseteq FinitePrologTerms(\Sigma,\mathcal{A})$, $(Range(\sigma|_{\mathcal{A}}) \not\subseteq \mathcal{A} \vee \sigma|_{\mathcal{A}} : Dom(\sigma|_{\mathcal{A}}) \to Range(\sigma|_{\mathcal{A}})$ *is not bijective),* $\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$, $\mathcal{G}' = \mathcal{G} \cup \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ *and* $\forall (s,s') \in \mathcal{U} : \forall \sigma'' : (s\sigma|_{\mathcal{G}}\sigma'' = s'\sigma|_{\mathcal{G}}\sigma'' \implies Dom(\sigma'') \not\subseteq \mathcal{F})$

ApproxUnify replaces some variables by fresh abstract variables:

$$ApproxUnify(\sigma,t_1,t_2,\mathcal{G},\mathcal{F}) = \begin{cases} \sigma & \text{if } \mathcal{A}(t_1)\cup\mathcal{A}(t_2) \subseteq \mathcal{G} \\ & \text{and } \mathcal{N}(t_1)\cup\mathcal{N}(t_2) \subseteq \mathcal{F} \\ \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'} & \text{if } \mathcal{A}(t_1)\cup\mathcal{A}(t_2) \subseteq \mathcal{G} \\ & \text{and } \mathcal{N}(t_1)\cup\mathcal{N}(t_2) \not\subseteq \mathcal{F} \\ \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} & \text{if } \mathcal{A}(t_1)\cup\mathcal{A}(t_2) \not\subseteq \mathcal{G} \end{cases}$$

We start proving the soundness of these abstract inference rules with the most complicated proof as we can refer to it for the remaining proofs. Note that this proof is quite similar to the soundness proof for Eval as given in [Sch08], while we have to deal with rational terms and two abstract terms instead of one in this proof, of course.

Lemma 4.25 (Soundness of UnifyCase)**.** *The rule* UnifyCase *from Definition 4.24 is sound.*

Proof. Assume $=\!(t_1\gamma,t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(=\!(t_1,t_2), Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U}))$ has an infinite concrete state-derivation. There are two cases depending on whether $t_1\gamma$ and $t_2\gamma$ unify.

First, if $t_1\gamma$ does not unify with $t_2\gamma$, the unique applicable concrete rule is UnifyFail and we obtain $S\gamma$ which has to start an infinite concrete state-derivation. From $t_1\gamma \not\sim t_2\gamma$ and γ being a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$, we obtain that γ is a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\})$, too. Thus, $S\gamma \in \mathcal{CON}(S; (\mathcal{G},\mathcal{F},\mathcal{U}\cup\{(t_1,t_2)\}))$.

Second, if $t_1\gamma \sim t_2\gamma$, the unique applicable concrete rule is UnifySuccess. From $t_1\gamma \sim t_2\gamma$ we directly know that t_1 also unifies with t_2. Let $mgu(t_1\gamma,t_2\gamma) = \sigma''$. Then due to $mgu(t_1,t_2) = \sigma$ there must be a substitution σ''' such that $\gamma\sigma'' = \sigma\sigma'''$. W.l.o.g., we demand that $\mathcal{V}(Range(\sigma'')) \subseteq \mathcal{N}_{fresh}$.

By application of the concrete UnifySuccess rule we obtain $Q\gamma\sigma'' \mid S\gamma$. We are, thus, left to show that $Q\gamma\sigma'' \mid S\gamma \in \mathcal{CON}(Q\sigma' \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}}))$, i.e., that there is a concretization γ' w.r.t. $(\mathcal{G}',\mathcal{F}',\mathcal{U}\sigma|_{\mathcal{G}})$ such that $Q\gamma\sigma'' = Q\sigma'\gamma'$ and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

We perform a case analysis over $\sigma' \in \{\sigma, \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}, \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\}$.

Case 1: $\sigma' = \sigma$, i.e., $\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G}$ and $\mathcal{N}(t_1) \cup \mathcal{N}(t_2) \subseteq \mathcal{F}$:
Here, we can assume $Dom(\sigma) = \mathcal{G}(t_1) \cup \mathcal{G}(t_2) \cup \mathcal{F}(t_1) \cup \mathcal{F}(t_2)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$ and $\gamma'(a) = \gamma(a)$ otherwise.

We first show that w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma'''' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ by showing that σ'''' is a most general unifier of $t_1\gamma$ and $t_2\gamma$. That σ'''' is most general follows from σ and σ'' being most general unifiers of t_1 and t_2 resp. $t_1\gamma$ and $t_2\gamma$. To see this consider that by the definition of σ''' as $\gamma\sigma'' = \sigma\sigma'''$ we have $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{N}}))}$. Clearly, $\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ is more general than $\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{N}}))}$ and, consequently, σ'''' is more general than σ'' which is a most general unifier of $t_1\gamma$ and $t_2\gamma$. We now show that σ'''' is still a unifier of $t_1\gamma$ and $t_2\gamma$:

$$
\begin{aligned}
& t_1\gamma\sigma'''' \\
\overset{Def.\sigma''''\wedge\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} \quad & t_1\gamma\sigma|_{\mathcal{N}(t_1\gamma)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_1\gamma)}))} \\
\overset{\mathcal{N}(Range(\gamma|_{\mathcal{V}(t_1)}))=\varnothing}{=} \quad & t_1\gamma\sigma|_{\mathcal{N}(t_1)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_1)}))} \\
\overset{\gamma|_{\mathcal{A}(t_1)}=(\gamma\sigma'')|_{\mathcal{A}(t_1)}=\sigma|_{\mathcal{A}(t_1)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_1)}))}}{=} \quad & t_1\sigma|_{\mathcal{A}(t_1)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_1)}))}\sigma|_{\mathcal{N}(t_1)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_1)}))} \\
\overset{\mathcal{N}(Range(\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_1)}))}))=\varnothing}{=} \quad & t_1\sigma|_{\mathcal{A}(t_1)}\sigma|_{\mathcal{N}(t_1)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{V}(t_1)}))} \\
\overset{\mathcal{V}=\mathcal{A}\uplus\mathcal{N}\wedge\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} \quad & t_1\sigma\sigma'''|_{\mathcal{A}(Range(\sigma))} \\
\overset{\sigma=mgu(t_1,t_2)}{=} \quad & t_2\sigma\sigma'''|_{\mathcal{A}(Range(\sigma))} \\
\overset{\mathcal{V}=\mathcal{A}\uplus\mathcal{N}\wedge\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} \quad & t_2\sigma|_{\mathcal{A}(t_2)}\sigma|_{\mathcal{N}(t_2)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{V}(t_2)}))} \\
\overset{\mathcal{N}(Range(\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_2)}))}))=\varnothing}{=} \quad & t_2\sigma|_{\mathcal{A}(t_2)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_2)}))}\sigma|_{\mathcal{N}(t_2)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_2)}))} \\
\overset{\gamma|_{\mathcal{A}(t_2)}=(\gamma\sigma'')|_{\mathcal{A}(t_2)}=\sigma|_{\mathcal{A}(t_2)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{A}(t_2)}))}}{=} \quad & t_2\gamma\sigma|_{\mathcal{N}(t_2)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_2)}))} \\
\overset{\mathcal{N}(Range(\gamma|_{\mathcal{V}(t_2)}))=\varnothing}{=} \quad & t_2\gamma\sigma|_{\mathcal{N}(t_2\gamma)}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}(t_2\gamma)}))} \\
\overset{Def.\sigma''''\wedge\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} \quad & t_2\gamma\sigma''''
\end{aligned}
$$

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus (\mathcal{A}\setminus\mathcal{A}(Range(\sigma)))$. For $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A}\setminus\mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

To show that $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, we make a case analysis over $a \in \mathcal{G}' = \mathcal{G} \uplus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. For $a \in \mathcal{G}$ we know that $a\gamma \in GroundTerms(\Sigma)$ and by $\gamma'|_{\mathcal{G}} \overset{\mathcal{A}(Range(\sigma)) \subseteq \mathcal{V}_{fresh} \wedge Def.\gamma'}{=} \gamma|_{\mathcal{G}}$ we obtain $a\gamma' \in GroundTerms(\Sigma)$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $a\sigma|_{\mathcal{G}} \in FinitePrologTerms(\Sigma, \mathcal{A})$ and $a\gamma' = a\sigma'''$. For all $a' \in Dom(\sigma|_{\mathcal{G}})$, $a'\sigma\sigma''' \overset{Def.\sigma'''}{=} a'\gamma\sigma'' \overset{a' \in \mathcal{G}}{\in} GroundTerms(\Sigma)$. Thus, $a\gamma' \in GroundTerms(\Sigma)$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus \mathcal{A}(Range(\sigma))) \uplus \mathcal{A}(Range(\sigma))$. For $a \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}' \setminus \mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{A}})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'') \overset{a' \in G}{=} \varnothing$.

Finally, we have $\bigwedge_{(t,t') \in \mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s') \in \mathcal{U}} s\sigma|_{\mathcal{G}}\gamma' \not\sim s'\sigma|_{\mathcal{G}}\gamma' = \bigwedge_{(s,s') \in \mathcal{U}} s\gamma \not\sim s'\gamma$ as $a\sigma|_{\mathcal{G}}\gamma' = a\gamma$ for all abstract variables in $a \in \mathcal{A}(\mathcal{U})$ by definition of γ'. To see this, consider the partition $\mathcal{A}(\mathcal{U}) = (\mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}}))$. If $a \in \mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})$ we have $a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a \notin Dom(\sigma|_{\mathcal{G}})}{=} a\sigma|_{\mathcal{G}}\gamma'$. If $a \in \mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}})$ we have $a\gamma \overset{a \in \mathcal{G}}{=} a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{a \in \mathcal{G}}{=} a\sigma|_{\mathcal{G}}\sigma''' \overset{Def.\gamma' \wedge \mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}}{=} a\sigma|_{\mathcal{G}}\gamma'$.

Now, we are left to show that $Q\gamma\sigma'' = Q\sigma'\gamma'$, and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For S there are two cases according to the partition $\mathcal{A}(S) = (\mathcal{A}(S) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(S) \cap Dom(\sigma|_{\mathcal{G}}))$. Analogous to the analysis for $\mathcal{A}(\mathcal{U})$ above, we have $a\gamma = a\sigma|_{\mathcal{G}}\gamma'$ for both cases. With $\gamma|_{\mathcal{A}} = \gamma$ and $\gamma'|_{\mathcal{A}} = \gamma'$ we obtain $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

Now, for Q consider the partition $\mathcal{V}(Q) = (\mathcal{A}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{A}(Q) \cap Dom(\sigma')) \uplus (\mathcal{N}(Q) \setminus \mathcal{F}) \uplus (\mathcal{F}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{F}(Q) \cap Dom(\sigma'))$ which leads to the following sub cases:

- $a \in \mathcal{A}(Q) \setminus Dom(\sigma')$:
  From $\mathcal{V}(t_1) = \mathcal{G}(t_1) \uplus \mathcal{F}(t_1)$ we get $\mathcal{V}(t_1\gamma) = \mathcal{N}(\mathcal{F}(t_1)\gamma) \cup \mathcal{N}(\mathcal{G}(t_1)\gamma) = \mathcal{F}(t_1)$. Together with $\mathcal{F}(a\gamma) = \varnothing$ we obtain $\mathcal{V}(a\gamma) \cap \mathcal{N}(t_1\gamma) = \mathcal{N}(a\gamma) \cap \mathcal{F}(t_1) \overset{\mathcal{F}(a\gamma) = \varnothing}{=} \varnothing$. Analogously, we obtain $\mathcal{V}(a\gamma) \cap \mathcal{N}(t_2) = \varnothing$ and, thus, $\mathcal{V}(a\gamma) \cap Dom(\sigma'') = \varnothing$. Thus, we have $a\gamma\sigma'' \overset{\mathcal{N}(a\gamma) \cap Dom(\sigma'') = \varnothing}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a \notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $a \in \mathcal{A}(Q) \cap Dom(\sigma')$:
  Note that $a \in Dom(\sigma')$ implies $a \in Dom(\sigma)$. We immediately have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma' \wedge \mathcal{V}(Range(\sigma|_{\mathcal{A}})) \subseteq \mathcal{A}}{=} a\sigma\gamma' \overset{\sigma = \sigma'}{=} a\sigma'\gamma'$.

- $x \in \mathcal{N}(Q) \setminus \mathcal{F}$:
  From $x \notin \mathcal{F}(t_1) \cup \mathcal{F}(t_2) = \mathcal{N}(t_1\gamma) \cup \mathcal{N}(t_2\gamma) = \mathcal{N}(t_1) \cup \mathcal{N}(t_2)$ we know that $x \notin Dom(\sigma'')$ and $x \notin Dom(\sigma')$. From $\gamma|_{\mathcal{A}} = \gamma$ and $\gamma'|_{\mathcal{A}} = \gamma'$ and $x \notin \mathcal{A}$ we get $x \notin Dom(\gamma)$ and $x \notin Dom(\gamma')$. Thus, we have $x\gamma\sigma'' \overset{x \notin Dom(\gamma)}{=} x\sigma'' \overset{x \notin Dom(\sigma'')}{=} x \overset{x \notin Dom(\gamma')}{=} x\gamma' \overset{x \notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(Q) \setminus Dom(\sigma')$:
  From $x \notin Dom(\sigma')$ we know $x \notin \mathcal{F}(t_1) \cup \mathcal{F}(t_2) = \mathcal{N}(t_1\gamma) \cup \mathcal{N}(t_2)$ and, consequently, $x \notin Dom(\sigma'')$. With $x \notin \mathcal{A}$ we have $x\gamma\sigma'' \overset{x \notin Dom(\gamma)}{=} x\sigma'' \overset{x \notin Dom(\sigma'')}{=} x \overset{x \notin Dom(\gamma')}{=} x\gamma' \overset{x \notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(Q) \cap Dom(\sigma')$:
  Note that $x \in Dom(\sigma')$ and $\sigma' = \sigma$ imply $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{x \notin \mathcal{A}}{=} x\sigma'' \overset{\sigma''=\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}}{=} x\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))} \overset{Def.\gamma'}{=} x\sigma|_{\mathcal{N}}\gamma' \overset{x \in \mathcal{N} \wedge \sigma = \sigma'}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(Q)$ and, consequently, $Q\gamma\sigma'' = Q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma$.

Case 2: $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$, i.e., $\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G}$ and $\mathcal{N}(t_1) \cup \mathcal{N}(t_2) \not\subseteq \mathcal{F}$:
Here, we can assume $Dom(\sigma) = \mathcal{G}(t_1) \cup \mathcal{G}(t_2) \cup \mathcal{N}(t_1) \cup \mathcal{N}(t_2)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in \mathcal{A} \setminus \mathcal{G}'$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{\mathcal{A}\setminus\mathcal{G}'}$ are fresh.

First, w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ by the identical argument as for the case of $\sigma' = \sigma$ where we made use of $\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G}$, which still holds for this case.

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t') \in \mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\approx t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})) \uplus (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))))$. For $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a \notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma'')) \subseteq \mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ there is an $a' \notin \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ such that $a'\alpha_{\mathcal{A}\setminus\mathcal{G}'} = a$ and $\mathcal{A}(a\gamma') \overset{a'\alpha_{\mathcal{A}\setminus\mathcal{G}'} = a}{=} \mathcal{A}(a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a'\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma'')) \subseteq \mathcal{N}}{=} \mathcal{A}(a'\gamma) \overset{\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing}{=} \varnothing$.

$Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$ follows from the identical argument as in the case $\sigma' = \sigma$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})))) \uplus \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'})))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}' \setminus \mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{A}})$ such that $a \in$

$\mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'') \overset{a'\in G}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{\mathcal{A}\setminus\mathcal{G}'}))$ there must be an $a' \in \mathcal{A} \setminus \mathcal{G}'$ such that $a = a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}$. Now assume $x \in F'(a\gamma')$. Then $x \in \mathcal{F}'(a'\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. Now, for $x \in \mathcal{F}'(a'\gamma\sigma'')$ there would have to be a $z \in \mathcal{N}(a'\gamma)$ such that $x \in \mathcal{N}(z\sigma'')$. As all variables in the range of σ'' are fresh, $x \notin \mathcal{F}$. From $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ we get $z \in Dom(\sigma)$. As $z \in \mathcal{N}(a'\gamma)$ and $\mathcal{F}(a'\gamma) = \varnothing$, we know $z \notin \mathcal{F}$. Thus, $z \in \mathcal{N} \setminus \mathcal{F}$ and, consequently, $x \in \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}}))$. But as $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$, this contradicts our assumption that $x \in \mathcal{F}'(a'\gamma\sigma'') = \mathcal{F}'(a\gamma')$.

Finally, we have $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ by the identical argument as the one used in the case of $\sigma' = \sigma$.

Now, we are left to show that $Q\gamma\sigma'' = Q\sigma'\gamma'$ and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For S, we can use the identical argument as for the case of $\sigma' = \sigma$.

Now, for Q consider the partition $\mathcal{V}(Q) = (\mathcal{A}(Q) \setminus \mathcal{G}) \uplus (\mathcal{G}(Q) \cap Dom(\sigma')) \uplus (\mathcal{G}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{N}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{N}(Q) \cap Dom(\sigma'))$ which leads to the following subcases:

- $a \in \mathcal{A}(Q) \setminus \mathcal{G}$:
  We immediately have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'}{=} a\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{a\notin Dom(\sigma)}{=} a\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(Q) \cap Dom(\sigma')$:
  Note that $a \in \mathcal{G}(Q) \cap Dom(\sigma')$ implies $a \in Dom(\sigma)$. Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'}{=} a\sigma\gamma' \overset{\mathcal{A}(a\sigma)\subseteq\mathcal{G}'}{=} a\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}=\sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(Q) \setminus Dom(\sigma')$:
  We have $a\gamma\sigma'' \overset{a\in\mathcal{G}}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a\notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $x \in \mathcal{N}(Q) \setminus Dom(\sigma')$:
  From $x \notin Dom(\sigma')$ we know $x \notin \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$. From $\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \subseteq \mathcal{G}$ and $\mathcal{N}(Range(\gamma|_{\mathcal{G}})) = \varnothing$ we know that $x \notin \mathcal{V}(t_1\gamma)$ and $x \notin \mathcal{V}(t_2\gamma)$. Thus, we have $x \notin Dom(\sigma'')$. Then, we have $x\gamma\sigma'' \overset{x\notin Dom(\gamma)}{=} x\sigma'' \overset{x\notin Dom(\sigma'')}{=} x \overset{x\notin Dom(\gamma')}{=} x\gamma' \overset{x\notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{N}(Q) \cap Dom(\sigma')$:
  Note that $x \in \mathcal{N}(Q) \cap Dom(\sigma')$ implies $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{\gamma|_{\mathcal{A}}=\gamma}{=} x\sigma'' \overset{\sigma''=\sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}}{=} x\sigma|_{\mathcal{N}}\sigma'''|\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \overset{x\in\mathcal{N}}{=} x\sigma\sigma'''|\mathcal{A}(Range(\sigma|_{\mathcal{N}})) \overset{x\in Dom(\sigma)}{=} x\sigma\sigma''' \overset{Def.\gamma'}{=} x\sigma\gamma' \overset{(\mathcal{A}\setminus\mathcal{G}')\cap\mathcal{A}(Range(\sigma))=\varnothing}{=} x\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}\gamma' \overset{\sigma'=\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(Q)$ and, consequently, $Q\gamma\sigma'' = Q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}$.

Case 3: $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$, i.e., $\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \not\subseteq \mathcal{G}$:

Here, we can assume $Dom(\sigma) = \mathcal{A}(t_1) \cup \mathcal{A}(t_2) \cup \mathcal{N}(t_1) \cup \mathcal{N}(t_2)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma))$, $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma'(a) = \sigma\sigma'''(a)$ for $a \in (\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')$, and $\gamma'(a) = \gamma(a)$ otherwise. This is possible as all variables in the ranges of σ and $\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ are fresh.

First, we show that w.l.o.g. we can demand that σ'' is chosen in such a way that $\sigma'' = \sigma''''$ for $\sigma''''|_{\mathcal{F}} = \sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}$ and $\sigma''''|_{\mathcal{N}\setminus(\mathcal{F})} = \sigma''|_{\mathcal{N}\setminus(\mathcal{F})}$ by showing that σ'''' is a most general unifier of $t_1\gamma$ and $t_2\gamma$. That σ'''' is most general follows from σ and σ'' being most general unifiers of t_1 and t_2 resp. $t_1\gamma$ and $t_2\gamma$. To see this consider that by the definition of σ''' as $\gamma\sigma'' = \sigma\sigma'''$ we have $\sigma''|_{\mathcal{F}} = \sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{F}}))}$. Clearly, $\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}$ is more general than $\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{F}}))}$ and, consequently, σ'''' is more general than σ'' which is a most general unifier of $t_1\gamma$ and $t_2\gamma$. We now show that σ'''' is still a unifier of $t_1\gamma$ and $t_2\gamma$:

$$
\begin{aligned}
t_1\gamma\sigma'''' &\overset{\mathcal{V}(Range(\sigma'')\cup Range(\sigma)\cup Range(\sigma'''))\subseteq\mathcal{V}_{fresh}}{=} && t_1\gamma\sigma''''|_{\mathcal{F}}\sigma''''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{\mathcal{F}(Range(\gamma))=\varnothing\wedge\gamma_{\mathcal{A}}=\gamma}{=} && t_1\sigma''''|_{\mathcal{F}}\gamma\sigma''''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{Def.\sigma''''}{=} && t_1\sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F})}=\sigma|_{\mathcal{V}\setminus(\mathcal{F})}}{\overset{\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))}}{=}} && t_1\sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\sigma|_{\mathcal{V}\setminus(\mathcal{F})} \\
& && \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{Dom(\sigma)\cap\mathcal{V}(Range(\sigma'''))=\varnothing}{=} && t_1\sigma|_{\mathcal{F}}\sigma|_{\mathcal{V}\setminus(\mathcal{F})}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))} \\
& && \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} && t_1\sigma\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{\sigma=mgu(t_1,t_2)}{=} && t_2\sigma\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{\mathcal{V}(Range(\sigma))\subseteq\mathcal{V}_{fresh}}{=} && t_2\sigma|_{\mathcal{F}}\sigma|_{\mathcal{V}\setminus(\mathcal{F})}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))} \\
& && \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{Dom(\sigma)\cap\mathcal{V}(Range(\sigma'''))=\varnothing}{=} && t_2\sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\sigma|_{\mathcal{V}\setminus(\mathcal{F})} \\
& && \sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))} \\
&\overset{\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F})}=\sigma|_{\mathcal{V}\setminus(\mathcal{F})}}{\overset{\sigma'''|_{\mathcal{V}(Range(\sigma|_{\mathcal{V}\setminus(\mathcal{F})}))}}{=}} && t_2\sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}\gamma\sigma''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{Def.\sigma''''}{=} && t_2\sigma''''|_{\mathcal{F}}\gamma\sigma''''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{\mathcal{F}(Range(\gamma))=\varnothing\wedge\gamma_{\mathcal{A}}=\gamma}{=} && t_2\gamma\sigma''''|_{\mathcal{F}}\sigma''''|_{\mathcal{N}\setminus(\mathcal{F})} \\
&\overset{\mathcal{V}(Range(\sigma'')\cup Range(\sigma)\cup Range(\sigma'''))\subseteq\mathcal{V}_{fresh}}{=} && t_2\gamma\sigma''''
\end{aligned}
$$

We are left to show that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}'(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})) \uplus (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))))$. For the case that $a \in \mathcal{A}(Range(\sigma))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma)}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$ there is an $a' \notin \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$ such that $a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} = a$ and $\mathcal{A}(a\gamma') \overset{a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}=a}{=} \mathcal{A}(a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a'\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a'\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

By the identical argument as for the case of $\sigma' = \sigma$, we obtain $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$.

Now, to show that $\mathcal{F}'(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))) \uplus \mathcal{A}(Range(\sigma)) \uplus \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$. For $a \in \mathcal{A} \setminus (\mathcal{A}(Range(\sigma)) \cup \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')})))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma)$ as all variables in $\mathcal{F}' \setminus \mathcal{F}$ are fresh. This amounts to $\mathcal{F}'(a\gamma') = \mathcal{F}'(a\gamma) = \mathcal{F}(a\gamma) = \varnothing$. For $a \in \mathcal{A}(Range(\sigma)) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{A}})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}'(a\gamma')$. Then we have $x \in \mathcal{F}'(a'\sigma\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. Now, for $x \in \mathcal{F}'(a'\gamma\sigma'')$ there would have to be a $z \in \mathcal{N}(a'\gamma)$ such that $x \in \mathcal{N}(z\sigma'')$. As all variables in the range of σ'' are fresh, $x \notin \mathcal{F}$. From $\sigma'' = \sigma|_{\mathcal{N}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{N}}))}$ we get $z \in Dom(\sigma)$. As $z \in \mathcal{N}(a'\gamma)$ and $\mathcal{F}(a'\gamma) = \varnothing$, we know $z \notin \mathcal{F}$. Thus, $z \in \mathcal{N} \setminus \mathcal{F}$ and, consequently, $x \in \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}}))$. But as $\mathcal{F}' = \mathcal{F} \cup (\mathcal{N}(Range(\sigma|_{\mathcal{F}})) \setminus \mathcal{N}(Range(\sigma|_{\mathcal{N}\setminus\mathcal{F}})))$, this contradicts our assumption that $x \in \mathcal{F}'(a'\gamma\sigma'') = \mathcal{F}'(a\gamma')$. For $a \in \mathcal{A}(Range(\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}))$ there must be an $a' \in (\mathcal{A} \setminus \mathcal{G}') \cup (\mathcal{N} \setminus \mathcal{F}')$ such that $a = a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$. Now assume $x \in F'(a\gamma')$. Then $x \in \mathcal{F}'(a'\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma') \overset{Def.\gamma'}{=} \mathcal{F}'(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}'(a'\gamma\sigma'')$. By the identical argument as for the case of $a \in \mathcal{A}(Range(\sigma))$ we can show that $x \notin \mathcal{F}'(a'\gamma\sigma'')$.

Finally, we have $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ by the identical argument as the one used in the case of $\sigma' = \sigma$.

Now, we are left to show that $Q\gamma\sigma'' = Q\sigma'\gamma'$ and $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

For S, we can use the identical argument as for the case of $\sigma' = \sigma$.

Now, for Q consider the partition $\mathcal{V}(Q) = ((\mathcal{V}(Q)\setminus(\mathcal{G}\cup\mathcal{F}))\cap Dom(\sigma)) \uplus ((\mathcal{V}(Q)\setminus(\mathcal{G}\cup\mathcal{F}))\setminus Dom(\sigma)) \uplus (\mathcal{G}(Q) \cap Dom(\sigma')) \uplus (\mathcal{G}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{F}(Q) \setminus Dom(\sigma')) \uplus (\mathcal{F}(Q) \cap Dom(\sigma'))$ which leads to the following sub cases:

- $a \in (\mathcal{V}(Q) \setminus (\mathcal{G} \cup \mathcal{F})) \cap Dom(\sigma)$:
  Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma' \wedge a \in Dom(\sigma)}{=} a\sigma\gamma' \overset{\mathcal{V}(Range(\sigma)) \subseteq \mathcal{V}_{fresh}}{=} a\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} = \sigma'}{=} a\sigma'\gamma'$.

- $a \in (\mathcal{V}(Q) \setminus (\mathcal{G} \cup \mathcal{F})) \setminus Dom(\sigma)$:
  Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma' \wedge a \notin Dom(\sigma)}{=} a\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{a \notin Dom(\sigma)}{=} a\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{\mathcal{A}\setminus\mathcal{G}'}}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(Q) \cap Dom(\sigma')$:
  Note that $a \in \mathcal{G}(Q) \cap Dom(\sigma')$ implies $a \in Dom(\sigma)$. Then, we have $a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{Def.\gamma'}{=} a\sigma\gamma' \overset{\mathcal{A}(a\sigma) \subseteq \mathcal{G}'}{=} a\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} = \sigma'}{=} a\sigma'\gamma'$.

- $a \in \mathcal{G}(Q) \setminus Dom(\sigma')$:
  We have $a\gamma\sigma'' \overset{a \in \mathcal{G}}{=} a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a \notin Dom(\sigma')}{=} a\sigma'\gamma'$.

- $x \in \mathcal{F}(Q) \setminus Dom(\sigma')$:
  From $x \notin Dom(\sigma')$ we get $x \notin \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$. From $x \in \mathcal{F}(Q) \subseteq \mathcal{F}$ we get $x \notin \mathcal{N}(Range(\gamma))$. Together, we obtain $x \notin \mathcal{V}(t_1\gamma) \cup \mathcal{V}(t_2\gamma)$ and, consequently, $x \notin Dom(\sigma'')$. Then, we have $x\gamma\sigma'' \overset{x \notin Dom(\gamma)}{=} x\sigma'' \overset{x \notin Dom(\sigma'')}{=} x \overset{x \notin Dom(\gamma')}{=} x\gamma' \overset{x \notin Dom(\sigma')}{=} x\sigma'\gamma'$.

- $x \in \mathcal{F}(Q) \cap Dom(\sigma')$:
  Note that $x \in \mathcal{F}(Q) \cap Dom(\sigma')$ implies $x \in Dom(\sigma)$. Then, we have $x\gamma\sigma'' \overset{\gamma|_{\mathcal{A}} = \gamma}{=} x\sigma'' \overset{\sigma''|_{\mathcal{F}} = \sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))}}{=} x\sigma|_{\mathcal{F}}\sigma'''|_{\mathcal{A}(Range(\sigma|_{\mathcal{F}}))} \overset{Def.\gamma'}{=} x\sigma|_{\mathcal{F}}\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{x \in \mathcal{F}}{=} x\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}\gamma' \overset{\sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')} = \sigma'}{=} x\sigma'\gamma'$.

Thus, we have shown that $x\gamma\sigma'' = x\sigma'\gamma'$ for all $x \in \mathcal{V}(Q)$ and, consequently, $Q\gamma\sigma'' = Q\sigma'\gamma'$.

This concludes the case of $\sigma' = \sigma\alpha_{(\mathcal{A}\setminus\mathcal{G}')\cup(\mathcal{N}\setminus\mathcal{F}')}$ and, consequently, our proof for the soundness of the UnifyCase rule. □

The proof for the soundness of UnifySuccess is analogous to the one for OnlyEval.

Lemma 4.26 (Soundness of UnifySuccess)**.** *The rule* UnifySuccess *from Definition 4.24 is sound.*

Proof. We have to show that if there is an infinite concrete state-derivation starting in $=(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(=(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ then there is an infinite concrete state-derivation starting in $Q\sigma'\gamma' \mid S\sigma|_{\mathcal{G}}\gamma' \in \mathcal{CON}(Q\sigma' \mid S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}}))$. Since the UnifySuccess rule is identical to the UnifyCase rule if we drop the right successor state of the UnifyCase rule and all conditions of the UnifyCase rule are implied by the conditions of the UnifySuccess rule, we are left to show that there is no concretization

γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ for which we have $t_1\gamma \nsim t_2\gamma$. Then the soundness of the UnifySuccess rule is implied by the soundness of the UnifyCase rule. We show the equivalent condition: $\forall\gamma : (\gamma \text{ is a concretization w.r.t. } (\mathcal{G},\mathcal{F},\mathcal{U}) \implies t_1\gamma \sim t_2\gamma)$.
So let γ be a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$. We show $t_1\gamma \sim t_2\gamma$ by defining a unifier σ'' of $t_1\gamma$ and $t_2\gamma$. Since $\sigma|_{\mathcal{A}} : \mathit{Dom}(\sigma|_{\mathcal{A}}) \to \mathit{Range}(\sigma|_{\mathcal{A}})$ is bijective and $\mathit{Range}(\sigma|_{\mathcal{A}}) \subseteq \mathcal{A}$, there is a substitution σ^{-1} with $\sigma^{-1}(\sigma(a)) = a$ for all $a \in \mathit{Dom}(\sigma|_{\mathcal{A}})$ and $\sigma^{-1}(x) = x$ otherwise. So we have $\sigma\sigma^{-1} = \sigma|_{\mathcal{N}}$. As we have $t_1\sigma = t_2\sigma$, we obtain $t_1\sigma\sigma^{-1} = t_2\sigma\sigma^{-1} \iff t_1\sigma|_{\mathcal{N}} = t_2\sigma|_{\mathcal{N}}$. We define $\sigma'' = \sigma|_{\mathcal{N}}\gamma$. Then we have:

$$\begin{aligned}
t_1\gamma\sigma'' &\overset{Def.\sigma''}{=} && t_1\gamma\sigma|_{\mathcal{N}}\gamma \\
&\overset{Dom(\gamma)\subseteq\mathcal{A}\wedge\gamma\gamma=\gamma}{=} && t_1\sigma|_{\mathcal{N}}\gamma\gamma \\
&\overset{t_1\sigma|_{\mathcal{N}}=t_2\sigma|_{\mathcal{N}}}{=} && t_2\sigma|_{\mathcal{N}}\gamma\gamma \\
&\overset{Dom(\gamma)\subseteq\mathcal{A}\wedge\gamma\gamma=\gamma}{=} && t_2\gamma\sigma|_{\mathcal{N}}\gamma \\
&\overset{Def.\sigma''}{=} && t_2\gamma\sigma''
\end{aligned}$$

□

Likewise, the soundness proof for UnifyFail corresponds to the one for Backtrack.

Lemma 4.27 (Soundness of UnifyFail)**.** *The rule* UnifyFail *from Definition 4.24 is sound.*

Proof. Assume there is an infinite concrete state-derivation from $=(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(=(t_1,t_2), Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U}))$.

Let γ be a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$. If $t_1\gamma \nsim t_2\gamma$ then the only applicable concrete rule is UnifyFail, which results in $S\gamma$ starting an infinite concrete state-derivation. From $t_1\gamma \nsim t_2\gamma$ we know that γ is also a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U} \cup \{(t; H_i)\})$ and $S\gamma \in \mathcal{CON}(S; (\mathcal{G},\mathcal{F},\mathcal{U} \cup \{(t; H_i)\}))$

Now we are left to show that there is no concretization γ w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$ with $t_1\gamma \sim t_2\gamma$.

If $t_1 \nsim t_2$, then there is no substitution δ with $t_1\delta \sim t_2\delta$. In particular, there is no concretization γ with $t_1\gamma \sim t_2\gamma$.

So let $\sigma = \mathit{mgu}(t_1, t_2)$. If there is a variable $a \in \mathcal{G}$ with $a\sigma \notin \mathit{FinitePrologTerms}(\Sigma,\mathcal{V})$, there is no concretization γ with $t_1\gamma \sim t_2\gamma$. To see this, assume there is an mgu σ'''' of $t_1\gamma$ and $t_2\gamma$. Then there must be a substitution δ' with $\sigma\delta' = \gamma\sigma''''$. We can assume $a \in \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$, since σ is most general. As $a\sigma \notin \mathit{FinitePrologTerms}(\Sigma,\mathcal{V})$ we also have $a\sigma\delta' = a\gamma\sigma'''' \notin \mathit{FinitePrologTerms}(\Sigma,\mathcal{V})$. Since γ has to replace all abstract variables in t_1 and t_2 and $\mathcal{A}(\mathit{Range}(\gamma)) = \varnothing$, we obtain $a\gamma \notin \mathit{FinitePrologTerms}(\Sigma,\mathcal{V}) \supseteq \mathit{GroundTerms}(\Sigma)$. Contradiction.

Now let $\mathcal{V}(\mathit{Range}(\sigma)) \subseteq \mathcal{V}_{\mathit{fresh}}$ and $\exists(s,s') \in \mathcal{U} : \sigma' = \mathit{mgu}(s\sigma|_{\mathcal{G}}, s'\sigma|_{\mathcal{G}}) \wedge \mathit{Dom}(\sigma') \subseteq \mathcal{F}$. We show that $\forall\delta : (\mathit{Dom}(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(\mathit{Range}(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$ by

showing that $\sigma'\delta$ is a unifier of $s\sigma|_{\mathcal{G}}\delta$ and $s'\sigma|_{\mathcal{G}}\delta$.
Let δ be a substitution with $Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}$. We immediately have that $\delta = \delta\delta$ since $Dom(\delta) \cap \mathcal{V}(Range(\delta)) = \varnothing$. Now we have:

$$
\begin{aligned}
s\sigma|_{\mathcal{G}}\delta\sigma'\delta &\overset{Dom(\delta)\subseteq\mathcal{A}}{=} s\sigma|_{\mathcal{G}}\delta|_{\mathcal{A}}\sigma'\delta|_{\mathcal{A}} \\
&\overset{Dom(\sigma')\subseteq\mathcal{F}}{=} s\sigma|_{\mathcal{G}}\delta|_{\mathcal{A}}\sigma'|_{\mathcal{F}}\delta|_{\mathcal{A}} \\
&\overset{\mathcal{V}(Range(\delta))\subseteq\mathcal{N}\setminus\mathcal{F}\wedge\delta=\delta\delta}{=} s\sigma|_{\mathcal{G}}\sigma'|_{\mathcal{F}}\delta|_{\mathcal{A}}\delta|_{\mathcal{A}} \\
&\overset{Dom(\sigma')\subseteq\mathcal{F}}{=} s\sigma|_{\mathcal{G}}\sigma'\delta|_{\mathcal{A}}\delta|_{\mathcal{A}} \\
&\overset{\sigma'=mgu(s|_{\mathcal{G}},s'|_{\mathcal{G}})}{=} s'\sigma|_{\mathcal{G}}\sigma'\delta|_{\mathcal{A}}\delta|_{\mathcal{A}} \\
&\overset{Dom(\sigma')\subseteq\mathcal{F}}{=} s'\sigma|_{\mathcal{G}}\sigma'|_{\mathcal{F}}\delta|_{\mathcal{A}}\delta|_{\mathcal{A}} \\
&\overset{\mathcal{V}(Range(\delta))\subseteq\mathcal{N}\setminus\mathcal{F}\wedge\delta=\delta\delta}{=} s'\sigma|_{\mathcal{G}}\delta|_{\mathcal{A}}\sigma'|_{\mathcal{F}}\delta|_{\mathcal{A}} \\
&\overset{Dom(\sigma')\subseteq\mathcal{F}}{=} s'\sigma|_{\mathcal{G}}\delta|_{\mathcal{A}}\sigma'\delta|_{\mathcal{A}} \\
&\overset{Dom(\delta)\subseteq\mathcal{A}}{=} s'\sigma|_{\mathcal{G}}\delta\sigma'\delta
\end{aligned}
$$

From $\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$ it follows that $\forall\delta\exists(s,s') \in \mathcal{U} : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta$. By disjunctively adding $t_1\delta \nsim t_2\delta$ we obtain:

$$\forall\delta\exists(s,s') \in \mathcal{U} : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta \vee t_1\delta \nsim t_2\delta$$

$$\overset{Dom(\delta)\subseteq\mathcal{A}\wedge\mathcal{V}(Range(\delta))\subseteq\mathcal{N}\setminus\mathcal{F} \text{ independent from } s,s',t_1, \text{ and } t_2}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A}\wedge\mathcal{V}(Range(\delta)) \subseteq \mathcal{N}\setminus\mathcal{F}) \implies (\exists(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \sim s'\sigma|_{\mathcal{G}}\delta)\vee t_1\delta \nsim t_2\delta$$

$$\overset{\text{Double negation}}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F}) \implies \neg(\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta) \vee t_1\delta \nsim t_2\delta$$

$$\overset{\text{Definition of implication}}{\iff}$$

$$\forall\delta : \neg((Dom(\delta) \subseteq \mathcal{A}\wedge\mathcal{V}(Range(\delta)) \subseteq \mathcal{N}\setminus\mathcal{F}))\vee\neg(\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta)\vee t_1\delta \nsim t_2\delta$$

$$\overset{\text{Factor out negation}}{\iff}$$

$$\forall\delta : \neg(Dom(\delta) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\delta)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta)) \vee t_1\delta \nsim t_2\delta$$

$$\overset{\text{Definition of implication}}{\iff}$$

$$\forall\delta : (Dom(\delta) \subseteq \mathcal{A}\wedge\mathcal{V}(Range(\delta)) \subseteq \mathcal{N}\setminus\mathcal{F}\wedge(\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\delta \nsim s'\sigma|_{\mathcal{G}}\delta)) \implies t_1\delta \nsim t_2\delta$$

We continue by showing that for all concretizations γ we have $t_1\gamma \nsim t_2\gamma$.
Let γ be a concretization with $\gamma = \sigma|_{\mathcal{G}}\gamma'$ where γ' is an arbitrary substitution. Then we have $\gamma\sigma|_{\mathcal{G}} = \sigma|_{\mathcal{G}}\gamma$. By definition, γ satisfies $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\gamma \nsim s'\gamma)$. Since $\mathcal{V}(Range(\gamma)) \subseteq \mathcal{N}$ we also have $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\gamma\sigma|_{\mathcal{G}} \nsim s'\gamma\sigma|_{\mathcal{G}})$ and hence $Dom(\gamma) \subseteq \mathcal{A} \wedge \mathcal{V}(Range(\gamma)) \subseteq \mathcal{N} \setminus \mathcal{F} \wedge (\forall(s,s') \in \mathcal{U} : s\sigma|_{\mathcal{G}}\gamma \nsim s'\sigma|_{\mathcal{G}}\gamma)$ which implies $t_1\gamma \nsim t_2\gamma$.
Now let γ be a concretization with $\gamma \neq \sigma|_{\mathcal{G}}\gamma'$ for all substitutions γ'. Assume $t_1\gamma \sim t_2\gamma$. So there is a substitution σ'' with $t_1\gamma\sigma'' = t_2\gamma\sigma''$. Since $mgu(t_1,t_2) = \sigma$ and $\sigma = \sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}$ we obtain $\exists\sigma''' : \sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''' = \gamma\sigma''$. As we have $Dom(\gamma) \subseteq \mathcal{A}$ and $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$ we also have that $(\sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''')|_{\mathcal{N}} = (\sigma|_{\mathcal{N}}\sigma''')|_{\mathcal{N}} = \sigma''$ and $(\sigma|_{\mathcal{G}}\sigma|_{\mathcal{V}\setminus\mathcal{G}}\sigma''')|_{\mathcal{G}} = (\sigma|_{\mathcal{G}}\sigma''')|_{\mathcal{G}} = \gamma|_{\mathcal{G}}$.
Let $x \in \mathcal{V}$ be any variable. We perform a case analysis over the partition $(Range(\sigma|_{\mathcal{G}}) \uplus Dom(\sigma|_{\mathcal{G}}) \uplus (\mathcal{V} \setminus (Dom(\sigma|_{\mathcal{G}}) \cup Range(\sigma|_{\mathcal{G}}))))$:

- $x \in (\mathcal{V} \setminus (Dom(\sigma|_{\mathcal{G}}) \cup Range(\sigma|_{\mathcal{G}})))$:
  We define $x\gamma' = x\gamma$ and have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$.

- $x \in Range(\sigma|_{\mathcal{G}})$:
  Then there is a variable $a \in Dom(\sigma|_{\mathcal{G}})$ and a position π with $(a\sigma|_{\mathcal{G}})|_\pi = x$. Additionally we know that $x \in \mathcal{A}$ and $a\sigma|_{\mathcal{G}}\sigma''' = a\gamma$. Hence we have $x\sigma''' = a\gamma|_\pi$. W.l.o.g. we demand $x\gamma = x\sigma'''$ since x is fresh. So we define $x\gamma' = x\sigma'''$. Then we have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$.

- $x \in Dom(\sigma|_{\mathcal{G}})$:
  We define $x\gamma' = x$ as x will already be replaced by $\sigma|_{\mathcal{G}}$. So we still have $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$ since all variables in $Range(\sigma|_{\mathcal{G}})$ are properly replaced.

So we have in all cases $x\sigma|_{\mathcal{G}}\gamma' = x\gamma$ and, therefore, $\sigma|_{\mathcal{G}}\gamma' = \gamma$. Contradiction. Thus, we have $t_1\gamma \nsim t_2\gamma$ again. □

Now, the soundness proof for NoUnifyCase has a similar structure to the one for UnifyCase. But as we do not apply the mgu to the current goal in case of a successful unification, the proof is much simpler than the one for UnifyCase.

Lemma 4.28 (Soundness of NoUnifyCase)**.** *The rule* NoUnifyCase *from Definition 4.24 is sound.*

Proof. Assume $\backslash\!=\!(t_1,t_2)\gamma, Q\gamma \mid S\gamma = \backslash\!=\!(t_1\gamma,t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash\!=\!(t_1,t_2), Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U}))$ has an infinite concrete state-derivation. There are two cases depending on whether $t_1\gamma$ and $t_2\gamma$ unify.

First, if $t_1\gamma$ does not unify with $t_2\gamma$, the unique applicable concrete rule is NoUnify-Success and we obtain $Q\gamma \mid S\gamma$ which has to start an infinite concrete state-derivation. From $t_1\gamma \nsim t_2\gamma$ and γ being a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U})$, we obtain that γ is a concretization w.r.t. $(\mathcal{G},\mathcal{F},\mathcal{U} \cup \{(t_1,t_2)\})$, too. Thus, $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G},\mathcal{F},\mathcal{U} \cup \{(t_1,t_2)\}))$.

Second, if $t_1\gamma \sim t_2\gamma$, the unique applicable concrete rule is NoUnifyFail. From $t_1\gamma \sim t_2\gamma$ we directly know that t_1 also unifies with t_2. Let $mgu(t_1\gamma, t_2\gamma) = \sigma''$. Then due to $mgu(t_1, t_2) = \sigma$ there must be a substitution σ''' such that $\gamma\sigma'' = \sigma\sigma'''$. W.l.o.g., we demand that $\mathcal{V}(Range(\sigma'')) \subseteq \mathcal{N}_{fresh}$.

By application of the concrete NoUnifyFail rule we obtain $S\gamma$. We are, thus, left to show that $S\gamma \in \mathcal{CON}(S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_{\mathcal{G}}))$, i.e., that there is a concretization γ' w.r.t. $(\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_{\mathcal{G}})$ such that $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$.

We can assume that $Dom(\sigma) \subseteq \mathcal{V}(t_1) \cup \mathcal{V}(t_2)$. Define $\gamma'(a) = \sigma'''(a)$ for $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ and $\gamma'(a) = \gamma(a)$ otherwise.

We continue by showing that γ' is a concretization w.r.t. $(\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_{\mathcal{G}})$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma'$.

As γ' is only defined for $\mathcal{A}$, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

To show that $\bigcup_{a\in\mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, we perform a case analysis w.r.t. $\mathcal{A} = \mathcal{A}(Range(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A} \setminus \mathcal{A}(Range(\sigma|_{\mathcal{G}})))$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\sigma''') \overset{a\notin Dom(\sigma|_{\mathcal{G}})}{=} \mathcal{A}(a\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{A}(a\gamma\sigma'') \overset{\mathcal{V}(Range(\sigma''))\subseteq\mathcal{N}}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$. For $a \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $\mathcal{A}(a\gamma') \overset{Def.\gamma'}{=} \mathcal{A}(a\gamma) \overset{\bigcup_{a\in\mathcal{A}}\mathcal{A}(a\gamma)=\varnothing}{=} \varnothing$.

To show that $Range(\gamma'|_{\mathcal{G}'}) \subseteq GroundTerms(\Sigma)$, we make a case analysis over $a \in \mathcal{G}' = \mathcal{G} \uplus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. For $a \in \mathcal{G}$ we know that $a\gamma \in GroundTerms(\Sigma)$ and by $\gamma'|_{\mathcal{G}} \overset{\mathcal{A}(Range(\sigma))\subseteq\mathcal{V}_{fresh}\wedge Def.\gamma'}{=} \gamma|_{\mathcal{G}}$ we obtain $a\gamma' \in GroundTerms(\Sigma)$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $a\sigma|_{\mathcal{G}} \in FinitePrologTerms(\Sigma, \mathcal{A})$ and $a\gamma' = a\sigma'''$. For all $a' \in Dom(\sigma|_{\mathcal{G}})$, $a'\sigma\sigma''' \overset{Def.\sigma'''}{=} a'\gamma\sigma'' \overset{a'\in\mathcal{G}}{\in} GroundTerms(\Sigma)$. Thus, $a\gamma' \in GroundTerms(\Sigma)$.

Now, to show that $\mathcal{F}(Range(\gamma')) = \varnothing$, we perform a case analysis over $a \in \mathcal{A} = (\mathcal{A} \setminus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))) \uplus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$. For $a \in \mathcal{A} \setminus \mathcal{A}(Range(\sigma|_{\mathcal{G}}))$ we have $a\gamma' = a\gamma$ and $\mathcal{F}(a\gamma) = \varnothing$ as γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. For $a \in \mathcal{A}(Range(\sigma|_{\mathcal{G}})) \subseteq \mathcal{A}(Range(\sigma|_{\mathcal{A}}))$ there must be an $a' \in Dom(\sigma|_{\mathcal{G}})$ such that $a \in \mathcal{A}(a'\sigma)$. Now, assume $x \in \mathcal{F}(a\gamma')$. Then we have $x \in \mathcal{F}(a'\sigma\gamma') \overset{Def.\gamma'\wedge a'\in Dom(\sigma|_{\mathcal{G}})}{=} \mathcal{F}(a'\sigma\sigma''') \overset{Def.\sigma'''}{=} \mathcal{F}(a'\gamma\sigma'') \overset{a'\in G}{=} \varnothing$.

Finally, we have $\bigwedge_{(t,t')\in\mathcal{U}\sigma|_{\mathcal{G}}} t\gamma' \not\sim t'\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\sigma|_{\mathcal{G}}\gamma' \not\sim s'\sigma|_{\mathcal{G}}\gamma' = \bigwedge_{(s,s')\in\mathcal{U}} s\gamma \not\sim s'\gamma$ as $a\sigma|_{\mathcal{G}}\gamma' = a\gamma$ for all abstract variables in $a \in \mathcal{A}(\mathcal{U})$ by definition of γ'. To see this, consider the partition $\mathcal{A}(\mathcal{U}) = (\mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}}))$. If $a \in \mathcal{A}(\mathcal{U}) \setminus Dom(\sigma|_{\mathcal{G}})$ we have $a\gamma \overset{Def.\gamma'}{=} a\gamma' \overset{a\notin Dom(\sigma|_{\mathcal{G}})}{=} a\sigma|_{\mathcal{G}}\gamma'$. If $a \in \mathcal{A}(\mathcal{U}) \cap Dom(\sigma|_{\mathcal{G}})$ we have $a\gamma \overset{a\in\mathcal{G}}{=} a\gamma\sigma'' \overset{Def.\sigma'''}{=} a\sigma\sigma''' \overset{a\in\mathcal{G}}{=} a\sigma|_{\mathcal{G}}\sigma''' \overset{Def.\gamma'\wedge\mathcal{V}(Range(\sigma|_{\mathcal{A}}))\subseteq\mathcal{A}}{=} a\sigma|_{\mathcal{G}}\gamma'$.

To show that $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$ we perform a case analysis according to the partition $\mathcal{A}(S) = (\mathcal{A}(S) \setminus Dom(\sigma|_{\mathcal{G}})) \uplus (\mathcal{A}(S) \cap Dom(\sigma|_{\mathcal{G}}))$. Analogous to the analysis for $\mathcal{A}(\mathcal{U})$ above, we have $a\gamma = a\sigma|_{\mathcal{G}}\gamma'$ for both cases. With $\gamma|_{\mathcal{A}} = \gamma$ and $\gamma'|_{\mathcal{A}} = \gamma'$ we obtain $S\gamma = S\sigma|_{\mathcal{G}}\gamma'$. □

For proving the soundness of NoUnifySuccess we can refer back to the proof for UnifyFail.

Lemma 4.29 (Soundness of NoUnifySuccess)**.** *The rule* NoUnifySuccess *from Definition 4.24 is sound.*

Proof. Assume there is an infinite concrete state-derivation from $\backslash\!=\!(t_1, t_2)\gamma, Q\gamma \mid S\gamma = \backslash\!=\!(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash\!=\!(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$.

Let γ be a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$. If $t_1\gamma \nsim t_2\gamma$ then the only applicable concrete rule is NoUnifySuccess, which results in $Q\gamma \mid S\gamma$ starting an infinite concrete state-derivation. From $t_1\gamma \nsim t_2\gamma$ we know that γ is also a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t; H_i)\})$ and $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U} \cup \{(t; H_i)\}))$

Now we are left to show that there is no concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ with $t_1\gamma \sim t_2\gamma$.

As we have the same situation as in the proof of Lemma 4.27, this is already shown there. □

Likewise, we can use the proof for UnifySuccess in the proof for NoUnifyFail.

Lemma 4.30 (Soundness of NoUnifyFail)**.** *The rule* NoUnifyFail *from Definition 4.24 is sound.*

Proof. We have to show that if there is an infinite concrete state-derivation starting in $\backslash\!=\!(t_1, t_2)\gamma, Q\gamma \mid S\gamma = \backslash\!=\!(t_1\gamma, t_2\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\backslash\!=\!(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ then there is an infinite concrete state-derivation starting in $S\sigma|_{\mathcal{G}}\gamma' \in \mathcal{CON}(S\sigma|_{\mathcal{G}}; (\mathcal{G}', \mathcal{F}, \mathcal{U}\sigma|_{\mathcal{G}}))$. Since the NoUnifyFail rule is identical to the NoUnifyCase rule if we drop the left successor state of the NoUnifyCase rule and all conditions of the NoUnifyCase rule are implied by the conditions of the NoUnifyFail rule, we are left to show that there is no concretization γ w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ for which we have $t_1\gamma \nsim t_2\gamma$. Then the soundness of the NoUnifyFail rule is implied by the soundness of the NoUnifyCase rule. We show the equivalent condition: $\forall\gamma : (\gamma$ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U}) \implies t_1\gamma \sim t_2\gamma)$.

As we have the same situation as in the proof of Lemma 4.26, this is already shown there. □

4.4 Type Testing

The group of built-in predicates for type testing is especially used to check inputs and to exclude or detect error cases in a Prolog program. As Prolog is a typeless language, the types considered in the built-in predicates correspond to a template rather than a real type.[12] All the predicates in this section have only empty answer substitutions and succeed or fail according to their tests.

The built-in predicate atomic/1 tests if its argument is a constant term, i.e., its root symbol has the arity zero and it is no variable.[13]

Example 4.31. The goals atomic(p) and atomic(0) succeed while the goals atomic(f(a)) and atomic(X) fail for $X \in \mathcal{N}$.

By contrast, compound/1 tests if its argument is not a constant or variable, but a compound term with at least one argument.

Example 4.32. The goal compound(f(a)) succeeds while the goals compound(p) and compound(X) fail for $X \in \mathcal{N}$.

Combining both tests, the built-in predicate nonvar/1 tests if its argument is no variable, i.e., a constant or a compound term.

Example 4.33. The goals nonvar(f(a)) and nonvar(p) succeed while the goal nonvar(X) fails for $X \in \mathcal{N}$.

Finally, var/1 tests if its argument is a variable.

Example 4.34. The goal var(X) succeeds while the goals var(f(a)) and var(p) fail for $X \in \mathcal{N}$.

The corresponding concrete inference rules are, thus, easily defined.

[12] Prolog distinguishes, however, between integers, floats and other terms. If this distinction can be seen as real types is not discussed here as we do not consider the arithmetical features of Prolog in this thesis.

[13] For atomic/1, numbers are treated like constants. This is why we can handle this built-in predicate in our setting where we ignore the difference between numbers and constants. See Section 4.6 for the problems we have with the similar built-in predicate atom/1.

Definition 4.35 (Concrete Inference Rules for Type Testing)**.**

$$\frac{\mathsf{atomic}(c), Q \mid S}{Q \mid S}\ (\textsc{AtomicSuccess}) \quad \textit{where } c \textit{ is a constant}$$

$$\frac{\mathsf{atomic}(t'), Q \mid S}{S}\ (\textsc{AtomicFail}) \quad \textit{where } t' \textit{ is no constant}$$

$$\frac{\mathsf{compound}(t'), Q \mid S}{Q \mid S}\ (\textsc{CompoundSuccess}) \quad \textit{where } t' \textit{ is no constant and no variable}$$

$$\frac{\mathsf{compound}(t'), Q \mid S}{S}\ (\textsc{CompoundFail}) \quad \textit{where } t' \textit{ is a constant or } t' \in \mathcal{N}$$

$$\frac{\mathsf{nonvar}(t'), Q \mid S}{Q \mid S}\ (\textsc{NonvarSuccess}) \quad \textit{where } t' \notin \mathcal{N}$$

$$\frac{\mathsf{nonvar}(x), Q \mid S}{S}\ (\textsc{NonvarFail}) \qquad \frac{\mathsf{var}(x), Q \mid S}{Q \mid S}\ (\textsc{VarSuccess})$$

$$\frac{\mathsf{var}(t'), Q \mid S}{S}\ (\textsc{VarFail}) \quad \textit{where } t' \notin \mathcal{N}$$

For the abstract inference rules we must again consider that abstract variables may represent both terms which satisfy and which do not satisfy the respective tests of the built-in predicates. Thus, we have an additional abstract inference rule for each built-in predicate for type testing which is applicable if its argument is an abstract variable. Unfortunately, as long as we do not extend our knowledge bases to also contain shape information, we cannot extract much knowledge from the tests and just have to consider both cases for a successful and failing test. For atomic/1 we know at least that an abstract variable must represent a ground term if this test succeeds. For var/1 we can even obtain more knowledge by considering one successor state for each variable used in the parent state and two more successor states for a foreign variable and a failing test. Note that such a case analysis is not possible in our setting for function symbols as we would have to extend the finite signature, while we already consider an infinite set of variables.[14] Furthermore, we know that the test for var/1 must fail for abstract variables from $\mathcal{G}$ as

[14] Of course, we could perform a case analysis over the function symbols in the finite signature, but then we would have to specify the signature explicitly or assume that input terms may only be formed by the function symbols occurring in the respective program. We refrain from such assumptions and leave the analysis of programs over an infinite or growing signature to future work.

these variables cannot represent variables from $\mathcal{N}$. Likewise, the test for $\mathsf{nonvar}/1$ must succeed in such cases. This leads to the following definition of abstract inference rules for built-in predicates for type testing.

Definition 4.36 (Abstract Inference Rules for Type Testing)**.**

$$\frac{\mathsf{atomic}(c), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{AtomicSuccess}) \quad \textit{where } c \textit{ is a constant}$$

$$\frac{\mathsf{atomic}(t'), Q \mid S; KB}{S; KB}\ (\textsc{AtomicFail}) \quad \begin{array}{l}\textit{where } t' \textit{ is no constant and}\\ \textit{no abstract variable}\end{array}$$

$$\frac{\mathsf{atomic}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G} \cup \{a\}, \mathcal{F}, \mathcal{U}) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{AtomicCase}) \quad \textit{where } a \in \mathcal{A}$$

$$\frac{\mathsf{compound}(f(t_1, \ldots, t_k)), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{CompoundSuccess}) \quad \begin{array}{l}\textit{where } f/k \in \Sigma\\ \textit{and } t_i \in\\ \mathit{PrologTerms}(\Sigma, \mathcal{V})\\ \textit{for all } i \in \{1, \ldots, k\}\end{array}$$

$$\frac{\mathsf{compound}(t'), Q \mid S; KB}{S; KB}\ (\textsc{CompoundFail}) \quad \textit{where } t' \textit{ is a constant or } t' \in \mathcal{N}$$

$$\frac{\mathsf{compound}(a), Q \mid S; KB}{Q \mid S; KB \qquad S; KB}\ (\textsc{CompoundCase}) \quad \textit{where } a \in \mathcal{A}$$

$$\frac{\mathsf{nonvar}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{NonvarSuccess}) \quad \textit{where } t' \notin \mathcal{V} \setminus \mathcal{G}$$

$$\frac{\mathsf{nonvar}(x), Q \mid S; KB}{S; KB}\ (\textsc{NonvarFail})$$

$$\frac{\mathsf{nonvar}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{NonvarCase}) \quad \textit{where } a \in \mathcal{A} \setminus \mathcal{G}$$

$$\frac{\mathsf{var}(x), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{VarSuccess})$$

$$\frac{\mathsf{var}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}\ (\textsc{VarFail}) \quad \textit{where } t' \notin \mathcal{V} \setminus \mathcal{G}$$

$$\frac{\mathsf{var}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})}{Q\sigma_0 \mid S\sigma_0; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_0) \ldots Q\sigma_n \mid S\sigma_n; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_n) \qquad S; (\mathcal{G}, \mathcal{F}, \mathcal{U})} \text{ (VarCase)}$$

where $a \in \mathcal{A} \setminus \mathcal{G}$, $(\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(\mathcal{U})) \setminus \mathcal{F} = \{x_1, \ldots, x_n\}$, $\sigma_0 = [a/x]$ *for* $x \in \mathcal{N}_{fresh}$ *and* $\forall i \in \{1, \ldots, n\} : \sigma_i = [a/x_i]$.

As the changes to the knowledge bases correspond to a kind of shape analysis only and not to instantiation of non-abstract variables, we can easily prove that these abstract inference rules are sound.

Lemma 4.37 (Soundness of Abstract Inference Rules for Type Testing)**.** *The rules* AtomicSuccess, AtomicFail, AtomicCase, CompoundSuccess, CompoundFail, CompoundCase, NonvarSuccess, NonvarFail, NonvarCase, VarSuccess, VarFail *and* VarCase *from Definition 4.36 are sound.*

Proof. For AtomicSuccess assume there is an infinite concrete state-derivation from $\mathsf{atomic}(c), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{atomic}(c), Q \mid S; KB)$ where c is a constant. Then the only applicable concrete inference rule is AtomicSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation.

For AtomicFail assume there is an infinite concrete state-derivation from $\mathsf{atomic}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{atomic}(t'), Q \mid S; KB)$ where t' is no constant and no abstract variable. As γ is only defined for abstract variables, the only applicable concrete inference rule is AtomicFail leading to the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation.

For AtomicCase assume there is an infinite concrete state-derivation from $\mathsf{atomic}(a\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{atomic}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $a \in \mathcal{A}$. Then there are two cases depending on whether $a\gamma$ is a constant. If $a\gamma$ is a constant, then the only applicable concrete inference rule is AtomicSuccess leading to the state $Q\gamma \mid S\gamma$ which starts an infinite concrete state-derivation. As $a\gamma$ is a constant, we have $a\gamma \in \mathit{GroundTerms}(\Sigma)$ and, thus, $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G} \cup \{a\}, \mathcal{F}, \mathcal{U}))$. So let $a\gamma$ be no constant. Then the only applicable concrete inference rule is AtomicFail leading to the state $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation.

For CompoundSuccess assume there is an infinite concrete state-derivation from $\mathsf{compound}(f(t_1\gamma, \ldots, t_k\gamma)), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{compound}(f(t_1, \ldots, t_k)), Q \mid S; KB)$ where $f/k \in \Sigma$ and $t_i \in \mathit{PrologTerms}(\Sigma, \mathcal{V})$ for all $i \in \{1, \ldots, k\}$. Then the only applicable concrete inference rule is CompoundSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation.

For CompoundFail assume there is an infinite concrete state-derivation from $\mathsf{compound}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{compound}(t'), Q \mid S; KB)$ where t' is a constant or $t' \in \mathcal{N}$. Then the only applicable concrete inference rule is CompoundFail leading to the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation.

For CompoundCase assume there is an infinite concrete state-derivation from $\mathsf{compound}(a\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{compound}(a), Q \mid S; KB)$ where $a \in \mathcal{A}$. Then there are two cases depending on whether $a\gamma$ is no constant and no variable. If $a\gamma$ is no constant and no variable, the only applicable concrete inference rule is CompoundSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation. So let $a\gamma$ be a variable or a constant. Then the only applicable concrete inference rule is CompoundFail leading to the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation.

For NonvarSuccess assume there is an infinite concrete state-derivation from $\mathsf{nonvar}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nonvar}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $t' \notin \mathcal{V} \setminus \mathcal{G}$. Then we have $t'\gamma \notin \mathcal{V}$ and the only applicable concrete inference rule is NonvarSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation.

For NonvarFail assume there is an infinite concrete state-derivation from $\mathsf{nonvar}(x\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nonvar}(x), Q \mid S; KB)$ where $x \in \mathcal{N}$. Then we have $x\gamma = x \in \mathcal{N}$ and the only applicable concrete inference rule is NonvarFail leading to the state $S\gamma \in \mathcal{CON}(S; KB)$ having an infinite concrete state-derivation.

For NonvarCase assume there is an infinite concrete state-derivation from $\mathsf{nonvar}(a\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nonvar}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $a \in \mathcal{A} \setminus \mathcal{G}$. Then we have two cases depending on whether $a\gamma \in \mathcal{N}$. If $a\gamma \in \mathcal{N}$, the only applicable concrete inference rule is NonvarFail leading to the state $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation. So let $a\gamma \notin \mathcal{N}$. Then the only applicable concrete inference rule is NonvarSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation.

For VarSuccess assume there is an infinite concrete state-derivation from $\mathsf{var}(x\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{var}(x), Q \mid S; KB)$ where $x \in \mathcal{N}$. Then we have $x\gamma = x \in \mathcal{N}$ and the only applicable concrete inference rule is VarSuccess leading to the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ having an infinite concrete state-derivation.

For VarFail assume there is an infinite concrete state-derivation from $\mathsf{var}(t'\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{var}(t'), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $t' \notin \mathcal{V} \setminus \mathcal{G}$. Then we have $t'\gamma \notin \mathcal{N}$ and the only applicable concrete inference rule is VarFail leading to the state $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation.

For VarCase assume there is an infinite concrete state-derivation from $\mathsf{var}(a\gamma), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{var}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ where $a \in \mathcal{A} \setminus \mathcal{G}$, $(\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(\mathcal{U})) \setminus \mathcal{F} = \{x_1, \ldots, x_n\}$, $x \in \mathcal{N}_{fresh}$, $\sigma_0 = [a/x]$ and $\forall i \in \{1, \ldots, n\} : \sigma_i = [a/x_i]$. If $a\gamma \notin \mathcal{N}$, the only applicable concrete inference rule is VarFail leading to the state $S\gamma \in \mathcal{CON}(S; (\mathcal{G}, \mathcal{F}, \mathcal{U}))$ having an infinite concrete state-derivation. So let $a\gamma = y \in \mathcal{N}$. Then the only applicable concrete inference rule is VarSuccess leading to the state $Q\gamma \mid S\gamma$ which starts an infinite concrete state-derivation. We perform a case analysis over $y \in \mathcal{N} = \{x_1, \ldots, x_n\} \uplus \mathcal{N} \setminus \{x_1, \ldots, x_n\}$.

- $y \in \{x_1, \ldots, x_n\}$:
  Then there is some $i \in \{1, \ldots, n\}$ with $y = x_i$ and we have $\sigma_i\gamma = \gamma$. Thus, we obtain $Q\gamma \mid S\gamma = Q\sigma_i\gamma \mid S\sigma_i\gamma$. We are left to show that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_i)$, i.e., $\gamma|_{\mathcal{A}} = \gamma$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma) = \varnothing$, $Range(\gamma|_{\mathcal{G}}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}(Range(\gamma)) = \varnothing$, and $\bigwedge_{(t,t') \in \mathcal{U}\sigma_i} t\gamma \not\approx t'\gamma$. All these properties except for $\bigwedge_{(t,t') \in \mathcal{U}\sigma_i} t\gamma \not\approx t'\gamma$ are implied by the fact that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$.

  As we know $\bigwedge_{(t,t') \in \mathcal{U}\sigma_i} t\gamma \not\approx t'\gamma \iff \bigwedge_{(s,s') \in \mathcal{U}} s\sigma_i\gamma \not\approx s'\sigma_i\gamma \overset{\sigma_i\gamma=\gamma}{\iff} \bigwedge_{(s,s') \in \mathcal{U}} s\gamma \not\approx s'\gamma$ which is also implied by the fact that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$, we have $Q\sigma_i\gamma \mid S\sigma_i\gamma \in \mathcal{CON}(Q\sigma_i \mid S\sigma_i; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_i))$.

- $y \in \mathcal{N} \setminus \{x_1, \ldots, x_n\}$:
  Then there is a variable renaming ρ with $Dom(\rho) = \{x, y\}$, $y\rho = x$ and $x\rho = y$. We define γ' by $a\gamma' = a\gamma\rho$ for $a \in \mathcal{A}$ and $x'\gamma' = x'$ for $x' \in \mathcal{N}$. Therefore, we obtain that $\sigma_0\gamma' = \gamma'$ and $Q\gamma\rho \mid S\gamma\rho = Q\gamma' \mid S\gamma' = Q\sigma_0\gamma' \mid S\sigma_0\gamma'$ has an infinite concrete state-derivation. We are, thus, left to show that γ' is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_0)$, i.e., $\gamma'|_{\mathcal{A}} = \gamma'$, $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}}) \subseteq GroundTerms(\Sigma)$, $\mathcal{F}(Range(\gamma')) = \varnothing$, and $\bigwedge_{(t,t') \in \mathcal{U}\sigma_0} t\gamma' \not\approx t'\gamma'$.

  As γ' is only defined for abstract variables, we trivially have $\gamma'|_{\mathcal{A}} = \gamma'$.

  Since $a\gamma' = a\gamma\rho$ for $a \in \mathcal{A}$, ρ is a variable renaming and γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$, we obtain $\bigcup_{a \in \mathcal{A}} \mathcal{A}(a\gamma') = \varnothing$, $Range(\gamma'|_{\mathcal{G}}) \subseteq GroundTerms(\Sigma)$ and $\mathcal{F}(Range(\gamma')) = \varnothing$.

  As we know $\bigwedge_{(t,t') \in \mathcal{U}\sigma_0} t\gamma' \not\approx t'\gamma' \iff \bigwedge_{(s,s') \in \mathcal{U}} s\sigma_0\gamma' \not\approx s'\sigma_0\gamma' \overset{\sigma_0\gamma'=\gamma'}{\iff} \bigwedge_{(s,s') \in \mathcal{U}} s\gamma' \not\approx s'\gamma'$ which is also implied by the fact that γ is a concretization w.r.t. $(\mathcal{G}, \mathcal{F}, \mathcal{U})$, ρ is a variable renaming and $a\gamma' = a\gamma\rho$ for $a \in \mathcal{A}$, we have $Q\sigma_0\gamma' \mid S\sigma_0\gamma' \in \mathcal{CON}(Q\sigma_0 \mid S\sigma_0; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_0))$.

□

4.5 Special Cases

Finally, we handle the built-in predicates flush_output/0, nl/0, write/1, write_canonical/1 and writeq/1 used for output. As the ISO standard defines that the default output stream is a text stream and we do not consider built-in predicates being capable of changing the current output stream, we can assume that the current output stream is always a text stream. Therefore, these built-in predicates cannot generate an error according to their definition in the ISO standard and we can handle them just like the built-in predicate true/0 as we do not consider the environment of the Prolog processor in this thesis. For their effect on the current output stream we refer to [DEC96].

Definition 4.38 (Concrete Inference Rules for flush_output/0, nl/0, write/1, writeq/1 and write_canonical/1)**.**

$$\frac{\mathsf{flush\_output}, Q \mid S}{Q \mid S}\ (\textsc{FlushOutput}) \qquad \frac{\mathsf{nl}, Q \mid S}{Q \mid S}\ (\textsc{Newline}) \qquad \frac{\mathsf{write}(t'), Q \mid S}{Q \mid S}\ (\textsc{Write})$$

$$\frac{\mathsf{write\_canonical}(t'), Q \mid S}{Q \mid S}\ (\textsc{WriteCanonical}) \qquad \frac{\mathsf{writeq}(t'), Q \mid S}{Q \mid S}\ (\textsc{Writeq})$$

As for true/0, the abstract inference rules for these built-in predicates are straightforward to define.

Definition 4.39 (Abstract Inference Rules for flush_output/0, nl/0, write/1, writeq/1 and write_canonical/1)**.**

$$\frac{\mathsf{flush\_output}, Q \mid S; KB}{Q \mid S; KB}\ (\textsc{FlushOutput}) \qquad \frac{\mathsf{nl}, Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Newline})$$

$$\frac{\mathsf{write}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Write}) \qquad \frac{\mathsf{write\_canonical}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{WriteCanonical})$$

$$\frac{\mathsf{writeq}(t'), Q \mid S; KB}{Q \mid S; KB}\ (\textsc{Writeq})$$

The soundness proofs for these abstract inference rules are very easy.

Lemma 4.40 (Soundness of FlushOutput, Newline, Write, WriteCanonical and Writeq)**.** *The rules* FlushOutput*,* Newline*,* Write*,* WriteCanonical *and* Writeq *from Definition 4.39 are sound.*

Proof. For FlushOutput assume there is an infinite concrete state-derivation from $\mathsf{flush\_output}, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nl}, Q \mid S; KB)$. As the unique applicable concrete inference rule is FlushOutput, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ which starts an infinite concrete state-derivation.

For Newline assume there is an infinite concrete state-derivation from $\mathsf{nl}, Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nl}, Q \mid S; KB)$. As the unique applicable concrete inference rule is Newline, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ which starts an infinite concrete state-derivation.

For Write assume there is an infinite concrete state-derivation from $\mathsf{write}(t'), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nl}, Q \mid S; KB)$ where $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V})$. As the unique applicable concrete inference rule is Write, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ which starts an infinite concrete state-derivation.

For WriteCanonical assume there is an infinite concrete state-derivation from $\mathsf{write\_canonical}(t'), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nl}, Q \mid S; KB)$ where $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V})$. As the unique applicable concrete inference rule is WriteCanonical, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ which starts an infinite concrete state-derivation.

For Writeq assume there is an infinite concrete state-derivation from $\mathsf{writeq}(t'), Q\gamma \mid S\gamma \in \mathcal{CON}(\mathsf{nl}, Q \mid S; KB)$ where $t' \in \mathit{PrologTerms}(\Sigma, \mathcal{V})$. As the unique applicable concrete inference rule is Writeq, we reach the state $Q\gamma \mid S\gamma \in \mathcal{CON}(Q \mid S; KB)$ which starts an infinite concrete state-derivation. □

4.6 Problems with Remaining Built-in Predicates

If we try to handle more of the built-in predicates as defined in [DEC96] in our setting, we will face several problems. We will now discuss these problems and try to give ideas how to overcome them in extensions of the framework presented here. We leave the elaboration of these ideas to future work.

Arithmetical Features

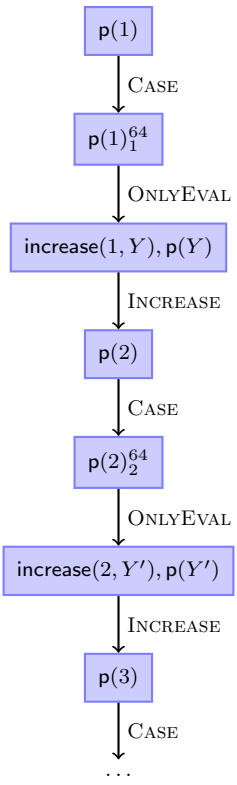
Prolog supports a number of arithmetical features. In fact, it distinguishes between normal terms, integers and floats. As we consider a finite signature, we cannot handle numbers, which are infinite. In spite of the fact that any real computer can only represent finitely many different numbers, Prolog supports unbounded integers, i.e., the only limit for represented integers is the available memory. Since we cannot assume any limit for the available memory, we virtually have to deal with infinitely many integers. Now, to have an infinite signature would not necessarily be a problem for the correctness of our approach. But the termination of the heuristic we will present in Chapter 6 relies on the finiteness of the signature. While we can define a safe criterion for generalization of non-numeric terms by looking at the nested depth of function symbols or the number of different subterms, this does not work for numbers as their size remains the same if we consider them as terms. This problem is illustrated by the following example.

Example 4.41. For simplicity, assume we would have a built-in predicate $\mathsf{increase}/2$ which is true if both arguments are integers and the second integers is the first increased by one and a suitable abstract inference rule for this predicate. Now consider the following Prolog program using this predicate

$$\mathsf{p}(X) \leftarrow \mathsf{increase}(X, Y), \mathsf{p}(Y). \tag{64}$$

and the set of queries $\mathcal{Q}$ consisting of the single query $\mathsf{p}(1)$. Then we would obtain the following infinite graph. We again omit the knowledge bases as we do not have any

abstract variables in this query.



In our setting, we can never find instances for such an abstract state-derivation. The only way to generalize the terms occurring in the abstract state-derivation according to the nested depth of symbols would be to already generalize the first occurrence of a symbol. But then we would have to generalize the whole term to an abstract variable which we cannot analyze anymore. Concerning the number of different subterms we would have to consider only terms with one subterm, i.e., constants and variables. Again, we would have to generalize the whole term which does not make sense.

The error cases for many built-in predicates are another problem related to integers. Whenever an integer is required as an argument for a built-in predicate, we would have to distinguish between integers and other terms just like Prolog.

Moreover, the order of terms in Prolog considers integers and floats differently compared to other terms. Thus, we cannot fully express the order of terms in our setting and, therefore, cannot handle built-in predicates relying on the order of terms.

Finally, the arithmetic computations performed by some built-in predicates require certain structures and mathematical conditions for the arithmetic expressions used in the calculations. These structures and conditions can only be checked successfully if we extend the handling of integers and floats according to the corresponding definitions in [DEC96].

The following built-in predicates make use of integers, consider arithmetical features or rely on the order of terms and can, therefore, not be handled in our setting.

- arg/3
- =:=/2
- =\=/2
- >/2
- >=/2
- </2
- =</2
- atom/1
- atom_codes/2
- atom_length/2
- char_code/2
- current_op/3
- float/1
- functor/3
- get_byte/1
- get_byte/2
- get_code/1
- get_code/2
- integer/1
- is/2
- number/1
- number_chars/2
- number_codes/2
- op/3
- put_byte/1
- put_byte/2
- put_code/1
- put_code/2
- sub_atom/2
- @>/2
- @>=/2
- @</2
- @=</2

To overcome the problems due to arithmetical features, we could extend our terms to be built not only from function symbols and variables, but also from integers and floats. Concerning generalization, we could extend the knowledge bases with a set for abstract variables which represent numbers (perhaps separated for integers and floats), possibly supplemented by information like intervals where the number might be in or comparisons with some values. Then we could generalize numbers outside of an interval and exceeding a limit on the precision of floats respectively. To handle arithmetic computations more precisely, it would be useful to adapt the definitions from integer term rewriting [FGP+09] to integer logic programming or integer dependency triples, for example. See also, e.g., the approach from [SD04] for termination analysis of Prolog programs using numerical features.

Characters

Prolog distinguishes not only numbers from other terms, but also characters. Especially, the identifiers for function symbols in our setting are built from characters in Prolog. As we do not consider characters that explicitly, we are not able to express the effects of some built-in predicates manipulating characters and function symbols. Also we cannot check for a character as an argument of a built-in predicate. Hence, we cannot detect some error cases of built-in predicates. Similar to characters is the use of character codes. While this is also problematic due to arithmetical features as these codes are integers, we have an additional problem due to characters which are represented by these codes. Furthermore, characters are also important for the order of terms. Hence, the built-in predicates relying on this order are also problematic due to characters. The following list shows the built-in predicates which we cannot handle in our setting due to their explicit usage of characters or character codes.

- atom_chars/2
- atom_codes/2
- atom_concat/3
- atom_length/2
- char_code/2
- char_conversion/2
- current_char_conversion/2
- get_char/1
- get_char/2
- get_code/1
- get_code/2
- number_chars/2
- number_codes/2
- peek_char/1
- peek_char/2
- peek_code/1
- peek_code/2
- put_char/1
- put_char/2
- put_code/1
- put_code/2
- @>/2
- @>=/2
- @</2
- @=</2

To handle the problems related to characters, we would need to represent function symbols as compositions of characters. Also, we would need a relation between such characters and their character codes.

Context

Some built-in predicates work in context of other built-in predicates. Therefore, we would need to adapt the definitions of our inference rules to also consider such contexts around the execution we have handled so far. While this problem is probably one of the easiest ones to solve, it will need some efforts to cover all combinations of contexts and to adapt the setting to the presence of contexts. Especially the rules INSTANCE and GENERALIZATION would need to be carefully adapted. While throw/1 belongs to the predicates working inside of contexts, we can handle it in our setting by exploiting the fact that we do not use the corresponding built-in predicate catch/3 which modifies the context relevant for throw/1. Built-in predicates which we cannot handle due to the context they are introducing are bagof/3, catch/3, findall/3 and setof/3.

Environment

A Prolog processor as defined in [DEC96] does not only consider the Prolog program and a query, but also an environment establishing elements for input and output as well as Prolog flags, operator tables and character conversion tables. In fact, even the program is not static, but parsed into a dynamic database which can be modified during the execution of the query. As we do not handle all these additional elements in our setting, we cannot handle built-in predicates accessing such elements correctly. As an exception, we can handle a few built-in predicates for output by exploiting the default configuration of the environment and the fact, that we do not consider possibilities to modify this configuration. The built-in predicates accessing the environment which we cannot handle in our setting are listed below.

- abolish/1
- asserta/1
- assertz/1
- at_end_of_stream/0
- at_end_of_stream/1
- char_conversion/2
- clause/2
- close/1
- close/2
- current_char_conversion/2
- current_input/1
- current_op/3
- current_output/1
- current_predicate/1
- current_prolog_flag/2
- flush_output/1
- get_byte/1
- get_byte/2
- get_char/1
- get_char/2
- get_code/1
- get_code/2
- nl/1
- op/3
- open/3
- open/4
- peek_byte/1
- peek_byte/2
- peek_char/1
- peek_char/2
- peek_code/1
- peek_code/2
- put_byte/2
- put_char/2
- put_code/2
- read/1
- read/2
- read_term/2
- read_term/3
- retract/1
- set_input/1
- set_output/1
- set_prolog_flag/2
- set_stream_position/2
- stream_property/2
- write/2
- write_canonical/2
- write_term/2
- write_term/3
- writeq/2

Ideas to solve the problems faced with the environment would be to add descriptions of elements of the environment (like the current database) to the states. But then we would need adaptions for INSTANCE and GENERALIZATION as instances in the current setting would not necessarily allow for the same concrete state-derivations if the underlying program is modified.

Rational Terms and Shape Information

The remaining built-in predicates we do not handle in this thesis are copy_term/2, unify_with_occurs_check/2 and =../2. While we could handle these predicates in a setting using unification with occurs-check, their behavior according to the ISO standard is undefined for infinite terms. First experiments with real implementations show that these predicates may not terminate in case their arguments are infinite terms. Also, for finite terms we would have problems in the abstract case for copy_term/2 and =../2 as these predicates access the shape of terms explicitly. In addition to that, we would still have to perform a unification for all three predicates which needs complex conditions in the abstract case as we have seen in Section 4.3. Thus, we refrain from handling these built-in predicates in this thesis and leave the experimental verification of their behavior with rational terms and their integration into this framework to future work.

4.7 Adaptions for New Inference Rules

As the result of equivalent concrete state-derivations for scope variants from Chapter 3 is based on a case analysis over the applicability of concrete inference rules, we have to extend the corresponding proofs by the new concrete inference rules.

Lemma 4.42 (Equivalent Concrete State-Derivations for Concrete Scope Variants)**.** *Given a concrete state S and a scope variant S' of S, all concrete state-derivations possible for S are also possible for S'.*

Proof. To show Lemma 4.42 it is sufficient to show that for all concrete rules the applicability of a rule for S implies the applicability for S' and after application of the rule the resulting states are still scope variants of each other. We perform a case analysis over the applicability of the concrete inference rules for S.

- For Success, Failure, VariableError, UndefinedError, Cut, CutAll, Case, Eval, Backtrack and Call the lemma is already shown in the proof of Lemma 3.33.

- AtomicFail is applicable:
  Then we have $S = \mathsf{atomic}(t'), Q \mid S''$ with t' not being a constant and as S' is a scope variant of S, we also have $S' = \mathsf{atomic}(t''), Q' \mid S'''$ with t'' not being a constant. Thus, AtomicFail is applicable for S', too. After application of AtomicFail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{atomic}(t'), Q \mid S''$ and $\mathsf{atomic}(t''), Q' \mid S'''$ are scope variants.

- AtomicSuccess is applicable:
  Then we have $S = \mathsf{atomic}(c), Q \mid S''$ with c being a constant and as S' is a scope variant of S, we also have $S' = \mathsf{atomic}(c), Q' \mid S'''$. Thus, AtomicSuccess is applicable for S', too. After application of AtomicSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{atomic}(c), Q \mid S''$ and $\mathsf{atomic}(c), Q' \mid S'''$ are scope variants.

- CompoundFail is applicable:
  Then we have $S = \mathsf{compound}(t'), Q \mid S''$ with t' being a constant or a variable and as S' is a scope variant of S, we also have $S' = \mathsf{compound}(t'), Q' \mid S'''$. Thus, CompoundFail is applicable for S', too. After application of CompoundFail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{compound}(t'), Q \mid S''$ and $\mathsf{compound}(t'), Q' \mid S'''$ are scope variants.

- CompoundSuccess is applicable:
  Then we have $S = \mathsf{compound}(t'), Q \mid S''$ with t' not being a constant or variable and as S' is a scope variant of S, we also have $S' = \mathsf{compound}(t''), Q' \mid S'''$ with

t'' not being a constant or variable. Thus, CompoundSuccess is applicable for S', too. After application of CompoundSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{compound}(t'), Q \mid S''$ and $\mathsf{compound}(t''), Q' \mid S'''$ are scope variants.

- Conjunction is applicable:
  Then we have $S = ,(t_1, t_2), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ,(t'_1, t'_2), Q' \mid S'''$. Thus, Conjunction is applicable for S', too. After application of Conjunction we obtain the states $t_1, t_2, Q \mid S''$ and $t'_1, t'_2, Q' \mid S'''$ which are scope variants of each other as $,(t_1, t_2), Q \mid S''$ and $,(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- Disjunction is applicable:
  Then we have $S = ;(t_1, t_2), Q \mid S''$ where $root(t_1) \neq \text{->}/2$ and as S' is a scope variant of S, we also have $S' = ;(t'_1, t'_2), Q' \mid S'''$ where $root(t'_1) \neq \text{->}/2$. Thus, Disjunction is applicable for S', too. After application of Disjunction we obtain the states $t_1, Q \mid t_2, Q \mid S''$ and $t'_1, Q' \mid t'_2, Q' \mid S'''$ which are scope variants of each other as $;(t_1, t_2), Q \mid S''$ and $;(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- EqualsFail is applicable:
  Then we have $S = ==(t_1, t_2), Q \mid S''$ with $t_1 \neq t_2$ and as S' is a scope variant of S, we also have $S' = ==(t'_1, t'_2), Q' \mid S'''$ with $t'_1 \neq t'_2$. Thus, EqualsFail is applicable for S', too. After application of EqualsFail we obtain the states S'' and S''' which are scope variants of each other as $==(t_1, t_2), Q \mid S''$ and $==(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- EqualsSuccess is applicable:
  Then we have $S = ==(t_1, t_1), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ==(t'_1, t'_1), Q' \mid S'''$. Thus, EqualsSuccess is applicable for S', too. After application of EqualsSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $==(t_1, t_1), Q \mid S''$ and $==(t'_1, t'_1), Q' \mid S'''$ are scope variants.

- Fail is applicable:
  Then we have $S = \mathsf{fail}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{fail}, Q' \mid S'''$. Thus, Fail is applicable for S', too. After application of Fail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{fail}, Q \mid S''$ and $\mathsf{fail}, Q' \mid S'''$ are scope variants.

- FlushOutput is applicable:
  Then we have $S = \mathsf{flush\_output}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{flush\_output}, Q' \mid S'''$. Thus, FlushOutput is applicable for S', too.

After application of FLUSHOUTPUT we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{flush\_output}, Q \mid S''$ and $\mathsf{flush\_output}, Q' \mid S'''$ are scope variants.

- HALT is applicable:
  Then we have $S = \mathsf{halt}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{halt}, Q' \mid S'''$. Thus, HALT is applicable for S', too. After application of HALT we obtain the states ε and ε which clearly are scope variants of each other.

- HALT1 is applicable:
  Then we have $S = \mathsf{halt}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{halt}(t''), Q' \mid S'''$. Thus, HALT1 is applicable for S', too. After application of HALT1 we obtain the states ε and ε which clearly are scope variants of each other.

- IFTHEN is applicable:
  Then we have $S = \text{->}(t_1, t_2), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \text{->}(t'_1, t'_2), Q' \mid S'''$. Thus, IFTHEN is applicable for S', too. After application of IFTHEN we obtain the states $\mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S''$ and $\mathsf{call}(t'_1), !_{m'}, t'_2, Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached states are scope variants of each other as $\text{->}(t_1, t_2), Q \mid S''$ and $\text{->}(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- IFTHENELSE is applicable:
  Then we have $S = ;(\text{->}(t_1, t_2), t_3), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = ;(\text{->}(t'_1, t'_2), t'_3), Q' \mid S'''$. Thus, IFTHENELSE is applicable for S', too. After application of IFTHENELSE we obtain the states $\mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m \mid S''$ and $\mathsf{call}(t'_1), !_{m'}, t'_2, Q' \mid t'_3, Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached states are scope variants of each other as $;(\text{->}(t_1, t_2), t_3), Q \mid S''$ and $;(\text{->}(t'_1, t'_2), t'_3), Q' \mid S'''$ are scope variants.

- NEWLINE is applicable:
  Then we have $S = \mathsf{nl}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{nl}, Q' \mid S'''$. Thus, NEWLINE is applicable for S', too. After application of NEWLINE we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{nl}, Q \mid S''$ and $\mathsf{nl}, Q' \mid S'''$ are scope variants.

- NONVARFAIL is applicable:
  Then we have $S = \mathsf{nonvar}(x), Q \mid S''$ with $x \in \mathcal{N}$ and as S' is a scope variant of S, we also have $S' = \mathsf{nonvar}(x), Q' \mid S'''$. Thus, NONVARFAIL is applicable for S', too. After application of NONVARFAIL we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{nonvar}(x), Q \mid S''$ and $\mathsf{nonvar}(x), Q' \mid S'''$ are scope variants.

- NonvarSuccess is applicable:
  Then we have $S = \mathsf{nonvar}(t'), Q \mid S''$ with t' not being a variable and as S' is a scope variant of S, we also have $S' = \mathsf{nonvar}(t''), Q' \mid S'''$ with t'' not being a variable. Thus, NonvarSuccess is applicable for S', too. After application of NonvarSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{nonvar}(t'), Q \mid S''$ and $\mathsf{nonvar}(t''), Q' \mid S'''$ are scope variants.

- Not is applicable:
  Then we have $S = \backslash+(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \backslash+(t''), Q' \mid S'''$. Thus, Not is applicable for S', too. After application of Not we obtain the states $\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S''$ and $\mathsf{call}(t''), !_{m'}, \mathsf{fail} \mid Q' \mid ?_{m'} \mid S'''$. As m and m' are fresh, we can demand that $m' = f(m)$. So the reached states are scope variants of each other as $\backslash+(t'), Q \mid S''$ and $\backslash+(t''), Q' \mid S'''$ are scope variants.

- NoUnifyFail is applicable:
  Then we have $S = \backslash=(t_1, t_2), Q \mid S''$ where $t_1 \sim t_2$ and as S' is a scope variant of S, we also have $S' = \backslash=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \sim t'_2$. Thus, NoUnifyFail is applicable for S', too. After application of NoUnifyFail we obtain the states S'' and S''' which are scope variants of each other as $\backslash=(t_1, t_2), Q \mid S''$ and $\backslash=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- NoUnifySuccess is applicable:
  Then we have $S = \backslash=(t_1, t_2), Q \mid S''$ where $t_1 \nsim t_2$ and as S' is a scope variant of S, we also have $S' = \backslash=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \nsim t'_2$. Thus, NoUnifySuccess is applicable for S', too. After application of NoUnifySuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\backslash=(t_1, t_2), Q \mid S''$ and $\backslash=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- Once is applicable:
  Then we have $S = \mathsf{once}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{once}(t''), Q' \mid S'''$. Thus, Once is applicable for S', too. After application of Once we obtain the states $\mathsf{call}(,(t', !)), Q \mid S''$ and $\mathsf{call}(,(t'', !)), Q' \mid S'''$ which are scope variants of each other as $\mathsf{once}(t'), Q \mid S''$ and $\mathsf{once}(t''), Q' \mid S'''$ are scope variants.

- Repeat is applicable:
  Then we have $S = \mathsf{repeat}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{repeat}, Q' \mid S'''$. Thus, Repeat is applicable for S', too. After application of Repeat we obtain the states $Q \mid \mathsf{repeat}, Q \mid S''$ and $Q' \mid \mathsf{repeat}, Q' \mid S'''$ which are scope variants of each other as $\mathsf{repeat}, Q \mid S''$ and $\mathsf{repeat}, Q' \mid S'''$ are scope variants.

- THROW is applicable:
  Then we have $S = \mathsf{throw}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{throw}(t''), Q' \mid S'''$. Thus, THROW is applicable for S', too. After application of THROW we obtain the states ε and ε which clearly are scope variants of each other.

- TRUE is applicable:
  Then we have $S = \mathsf{true}, Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{true}, Q' \mid S'''$. Thus, TRUE is applicable for S', too. After application of TRUE we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{true}, Q \mid S''$ and $\mathsf{true}, Q' \mid S'''$ are scope variants.

- UNEQUALSFAIL is applicable:
  Then we have $S = \backslash\!\!=\!\!=(t_1, t_1), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \backslash\!\!=\!\!=(t'_1, t'_1), Q' \mid S'''$. Thus, UNEQUALSFAIL is applicable for S', too. After application of UNEQUALSFAIL we obtain the states S'' and S''' which are scope variants of each other as $\backslash\!\!=\!\!=(t_1, t_1), Q \mid S''$ and $\backslash\!\!=\!\!=(t'_1, t'_1), Q' \mid S'''$ are scope variants.

- UNEQUALSSUCCESS is applicable:
  Then we have $S = \backslash\!\!=\!\!=(t_1, t_2), Q \mid S''$ where $t_1 \neq t_2$ and as S' is a scope variant of S, we also have $S' = \backslash\!\!=\!\!=(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \neq t'_2$. Thus, UNEQUALSSUCCESS is applicable for S', too. After application of UNEQUALSSUCCESS we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\backslash\!\!=\!\!=(t_1, t_2), Q \mid S''$ and $\backslash\!\!=\!\!=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- UNIFYFAIL is applicable:
  Then we have $S = =(t_1, t_2), Q \mid S''$ with $t_1 \nsim t_2$ and as S' is a scope variant of S, we also have $S' = =(t'_1, t'_2), Q' \mid S'''$ with $t'_1 \nsim t'_2$. Thus, UNIFYFAIL is applicable for S', too. After application of UNIFYFAIL we obtain the states S'' and S''' which are scope variants of each other as $=(t_1, t_2), Q \mid S''$ and $=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- UNIFYSUCCESS is applicable:
  Then we have $S = =(t_1, t_2), Q \mid S''$ where $t_1 \sim t_2$ and as S' is a scope variant of S, we also have $S' = =(t'_1, t'_2), Q' \mid S'''$ where $t'_1 \sim t'_2$. Thus, UNIFYSUCCESS is applicable for S', too. After application of UNIFYSUCCESS we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $=(t_1, t_2), Q \mid S''$ and $=(t'_1, t'_2), Q' \mid S'''$ are scope variants.

- VARFAIL is applicable:
  Then we have $S = \mathsf{var}(t'), Q \mid S''$ with t' not being a variable and as S' is a scope

variant of S, we also have $S' = \mathsf{var}(t''), Q' \mid S'''$ with t'' not being a variable. Thus, VarFail is applicable for S', too. After application of VarFail we obtain the states S'' and S''' which are scope variants of each other as $\mathsf{var}(t'), Q \mid S''$ and $\mathsf{var}(t''), Q' \mid S'''$ are scope variants.

- VarSuccess is applicable:
  Then we have $S = \mathsf{var}(x), Q \mid S''$ with $x \in \mathcal{N}$ and as S' is a scope variant of S, we also have $S' = \mathsf{var}(x), Q' \mid S'''$. Thus, VarSuccess is applicable for S', too. After application of VarSuccess we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{var}(x), Q \mid S''$ and $\mathsf{var}(x), Q' \mid S'''$ are scope variants.

- Write is applicable:
  Then we have $S = \mathsf{write}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{write}(t''), Q' \mid S'''$. Thus, Write is applicable for S', too. After application of Write we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{write}(t'), Q \mid S''$ and $\mathsf{write}(t''), Q' \mid S'''$ are scope variants.

- WriteCanonical is applicable:
  Then we have $S = \mathsf{write\_canonical}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{write\_canonical}(t''), Q' \mid S'''$. Thus, WriteCanonical is applicable for S', too. After application of WriteCanonical we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{write\_canonical}(t'), Q \mid S''$ and $\mathsf{write\_canonical}(t''), Q' \mid S'''$ are scope variants.

- Writeq is applicable:
  Then we have $S = \mathsf{writeq}(t'), Q \mid S''$ and as S' is a scope variant of S, we also have $S' = \mathsf{writeq}(t''), Q' \mid S'''$. Thus, Writeq is applicable for S', too. After application of Writeq we obtain the states $Q \mid S''$ and $Q' \mid S'''$ which are scope variants of each other as $\mathsf{writeq}(t'), Q \mid S''$ and $\mathsf{writeq}(t''), Q' \mid S'''$ are scope variants.

□

Lemma 4.43 (Equivalent Concrete State-Derivations for Abstract Scope Variants)**.** *Given an abstract state S and a scope variant S' of S, for every concrete state S_c represented by S there exists a concrete state S'_c represented by S' such that all concrete state-derivations possible for S_c are also possible for S'_c.*

Proof. As concretizations only replace abstract variables, we have for every concretization γ that $S'\gamma$ is a scope variant of $S\gamma$. By Lemma 4.42 we obtain that all concrete state-derivations possible for $S\gamma$ are also possible for $S'\gamma$. □

4.8 Summary

We explained the effect of 24 additional built-in predicates in our setting and introduced 33 additional concrete inference rules to simulate these effects in our approach. Then we introduced 41 additional abstract inference rules such that we can handle the additional built-in predicates for sets of queries, too. All new abstract inference rules are proved to be sound and we adapted all proofs from Chapter 3 relying on the set of used concrete inference rules to the extended set. Furthermore, we discussed problems and possible solutions for the remaining built-in predicates not handled in this thesis.

5 Operational Semantics with Concrete Inference Rules

In this chapter we consider the operational semantics of Prolog as defined in the ISO standard [DEC96] and prove that we can simulate every computation according to the operational semantics with our concrete inference rules as long as the computation does not use built-in predicates which we cannot handle or has to perform transformations for infinite terms which are not defined according to the ISO standard. Thus, this chapter shows that our approach is in fact capable of analyzing real Prolog applications, since the termination graphs we construct by applying abstract inference rules to abstract states represent concrete state-derivations for concrete states. By proving termination of an abstract state we prove termination of every concrete state represented by the respective abstract state. By the theorem we prove in this chapter, this implies termination of the Prolog program w.r.t. all queries represented by the concrete states shown to be terminating.

Structure of the Chapter

We start in Section 5.1 by establishing some summarizing definitions used in the remainder of the thesis.

Then we continue by stating the operational semantics for Prolog as given in the ISO standard [DEC96] which is described by Prolog search-trees and an algorithm constructing such trees in Section 5.2.

Finally, we prove that we can simulate the construction of Prolog search-trees with concrete state-derivations in Section 5.3. To this end, we define which Prolog search-trees are represented by concrete state-derivations and show that for every Prolog search-tree there is a concrete state-derivation representing this tree.

In Section 5.4 we summarize the contributions of this chapter.

5.1 Complete Rule Set

We have introduced all concrete and abstract inference rules used in this thesis. This is stated by the following two definitions.

Definition 5.1 (Set of Concrete Inference Rules)**.** *The set ConcreteInferenceRules is defined as the set containing exactly the following 43 concrete inference rules:*

- AtomicFail
- AtomicSuccess
- Backtrack
- Call
- Case
- CompoundFail
- CompoundSuccess
- Conjunction
- Cut
- CutAll
- Disjunction
- EqualsFail
- EqualsSuccess
- Eval
- Fail
- Failure
- FlushOutput
- Halt
- Halt1
- IfThen
- IfThenElse
- Newline
- NonvarFail
- NonvarSuccess
- Not
- NoUnifyFail
- NoUnifySuccess
- Once
- Repeat
- Success
- Throw
- True
- UndefinedError
- UnequalsFail
- UnequalsSuccess
- UnifyFail
- UnifySuccess
- VariableError
- VarFail
- VarSuccess
- Write
- WriteCanonical
- Writeq

Definition 5.2 (Set of Abstract Inference Rules)**.** *The set AbstractInferenceRules is defined as the set containing exactly the following 56 abstract inference rules:*

- AtomicCase
- AtomicFail
- AtomicSuccess
- Backtrack
- Call
- Case
- CompoundCase
- CompoundFail
- CompoundSuccess
- Conjunction
- Cut
- CutAll
- Disjunction
- EqualsCase
- EqualsFail
- EqualsSuccess
- Eval
- Fail
- Failure
- FlushOutput
- Generalization
- Halt
- Halt1
- IfThen
- IfThenElse
- Instance
- Newline
- NonvarCase
- NonvarFail
- NonvarSuccess
- Not
- NoUnifyCase
- NoUnifyFail
- NoUnifySuccess
- Once
- OnlyEval
- Parallel
- Repeat
- Split
- Success
- Throw
- True
- UndefinedError
- UnequalsCase
- UnequalsFail
- UnequalsSuccess
- UnifyCase
- UnifyFail
- UnifySuccess
- VarCase
- VariableError
- VarFail
- VarSuccess
- Write
- WriteCanonical
- Writeq

Additionally, we now introduce some denotations which we will use in the remainder of this thesis according to the built-in predicates we can handle in our setting.

Definition 5.3 (Handled Built-in Predicates)**.** *The set HandledBuiltInPredicates is defined as the set containing exactly the following 26 function symbols:*

- atomic/1
- call/1
- compound/1
- ,/2
- !/0
- ;/2
- fail/0
- flush_output/0
- halt/0
- halt/1
- ->/2
- nl/0
- nonvar/1
- \+/1
- once/1
- repeat/0
- ==/2
- \==/2
- throw/1
- true/0
- \=/2
- =/2
- var/1
- write/1
- write_canonical/1
- writeq/1

The set UnhandledBuiltInPredicates is defined by

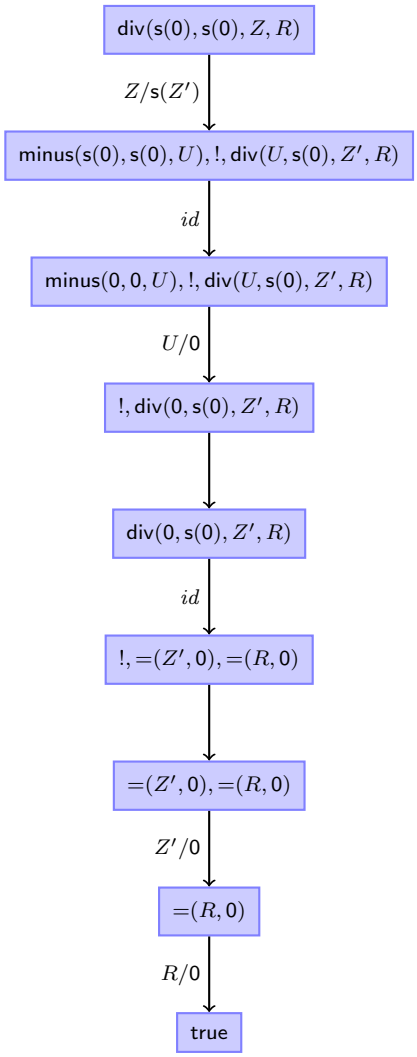
$$UnhandledBuiltInPredicates = BuiltInPredicates \setminus HandledBuiltInPredicates.$$

5.2 Operational Semantics of the ISO Standard

The ISO standard for Prolog [DEC96] defines the *operational semantics* of Prolog programs in terms of Prolog search-trees. These trees represent the computation executed by a standard conforming Prolog processor. Therefore, we first state the definition of Prolog search-trees as given in [DEC96] using the notations established in this thesis.

Definition 5.4 (Prolog Search-Tree [DEC96])**.** *A* Prolog search-tree *is a tree whose nodes are labeled with a current goal from $\mathcal{T}^{rat}(\Sigma, \mathcal{N})^*$ and a local substitution. Additionally it has a set of unvisited nodes which is a subset of the nodes in the tree. If the tree is finished, it has no current node and no unvisited nodes. Otherwise it has exactly one current node which belongs to the tree, but not to the set of unvisited nodes.*

Example 5.5. Consider the Prolog program $\mathcal{P}$ for division with remainder from Example 4.22 and the query $\mathsf{div}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z, R)$. The Prolog search-tree corresponding to the complete evaluation of this query is depicted below. Here, we placed the labels corresponding to the local substitutions next to the edges leading to the node where this label belongs to while we omit those parts of the local substitutions belonging to the fresh variables from the program only. As the evaluation is finished, there are no unvisited nodes and no current node. We will see examples of Prolog search-trees where the computation is still in progress after the next definition.



Now, for definite logic programs, the operational semantics is defined by means of resolution and SLD-trees. Instead of separately defining what trees can be inferred and how such trees are visited, the ISO standard for Prolog describes the operational semantics of Prolog by only one algorithm combining the construction and visiting of Prolog search-trees during the execution of a Prolog program w.r.t. a query. Thus, we state this algorithm in the following definition.[15]

Definition 5.6 (Search-Tree Visit and Construction Algorithm [DEC96])**.** *For a standard conforming Prolog processor with the flag* unknown *set to* error *and no other built-in predicates than the ones in BuiltInPredicates, we define the following algorithm which constructs a Prolog search-tree for the computation of the processor as defined in [DEC96].*

Given a Prolog program $\mathcal{P}$ and a transformed goal Q, the search-tree visit and construction algorithm *works as follows:*

[15] Note that we omit the flattening of conjunctions as this is equivalent to executing the built-in predicate ,/2. This flattening is not necessarily required by the standard, but just an equivalent behavior is demanded.

1. *Start from the root as current node, labeled by the initial goal Q, which is a sequence of predications, as current goal, and by the empty substitution as local substitution.*

2. *If the goal Q of the current node is* true *then backtrack to the first unvisited node n w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search-tree and continue with step 2 where the current node is n and n is dropped from the set of unvisited nodes.*

3. *Otherwise let t be the first predication in Q.*

4. *If t is* true *delete it, and proceed to step 2 with the new current goal being the tail of the sequence Q.*

5. *If t corresponds to a user-defined procedure which exists in the database, i.e., $Slice(\mathcal{P}, t) \neq \varnothing$:*

   (a) *If no renamed clause in $\mathcal{P}$ has a head which unifies with t then backtrack to the first unvisited node n w.r.t. the depth-first, left-to-right ordering of the nodes in the* Prolog *search-tree and continue with step 2 where the current node is n and n is dropped from the set of unvisited nodes.*

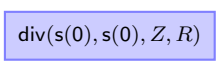
   (b) *Otherwise add to the current node as many children as there are freshly renamed clauses $H \leftarrow B \in \mathcal{P}$ whose head is unifiable with t with the same order as the clauses in $\mathcal{P}$. The child nodes are labeled with a local substitution $\sigma = mgu(t, H)$ ($H \leftarrow B$ being the corresponding freshly renamed clause), and the current goal Q' which is the instance by σ of Q in which t has been previously replaced by B. The current node becomes the first child and proceed to step 2.*

6. *Else if t corresponds to a built-in predicate: The specific side effects described with the built-in predicate in [DEC96] are performed and the execution continues at step 2 with or without preceding backtracking or generates an error according to the description of the built-in predicate in [DEC96].*

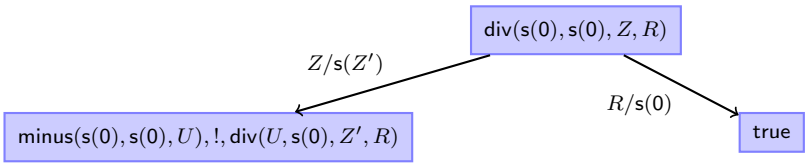
7. *Otherwise t does not correspond to any existing procedure and an error is generated whose effect corresponds to the execution at the same node of the built-in predicate* throw(existence_error(procedure, $root(t)$)).

Example 5.7. Consider again the Prolog program $\mathcal{P}$ for division with remainder from Example 4.22 and the query $\mathsf{div}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), Z, R)$. Now we show step by step how this algorithm works to built the tree from Example 5.5.

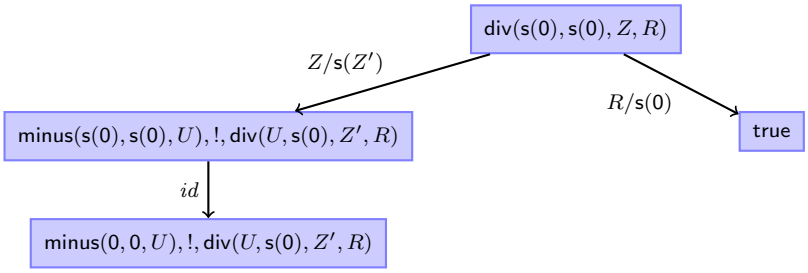
We start with the initial node as the current node labeled with the initial goal and the empty local substitution and no unvisited nodes.



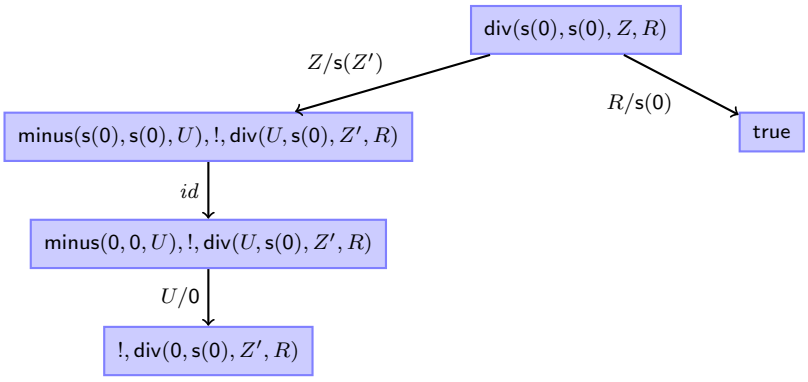
As the goal is not true, we check if the first predication in this goal is a user-defined, built-in or undefined predicate. Since div/4 is user-defined, we add nodes for all clauses where the head of the clause unifies with the first predication in the current goal. In this case, there are two clause heads unifying with the predication and we reach the following tree.



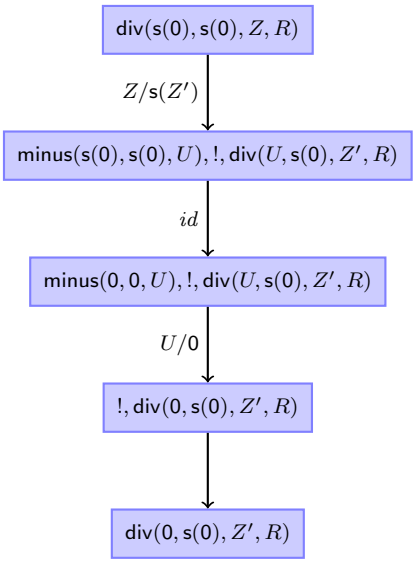
The second successor node is the only node in the set of unvisited nodes while the first successor node is the current node. Now we repeat the preceding steps for the new current goal and find that we have only one unifying clause for the predication with the user-defined predicate minus. Hence, we reach the following tree, where the set of unvisited nodes remains unchanged while the current node becomes the new node.



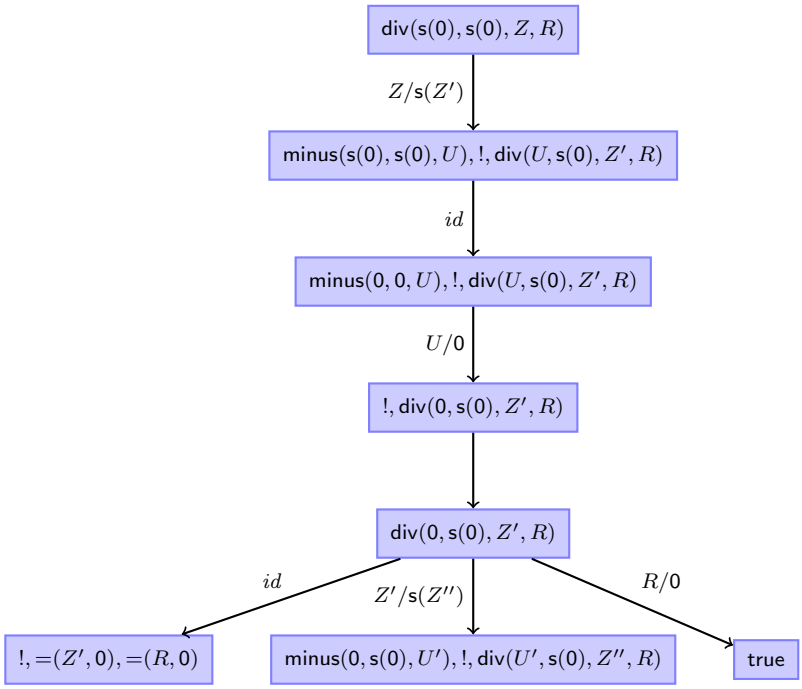
Again, we only have one successor node and obtain the next tree analogously to the previous one.



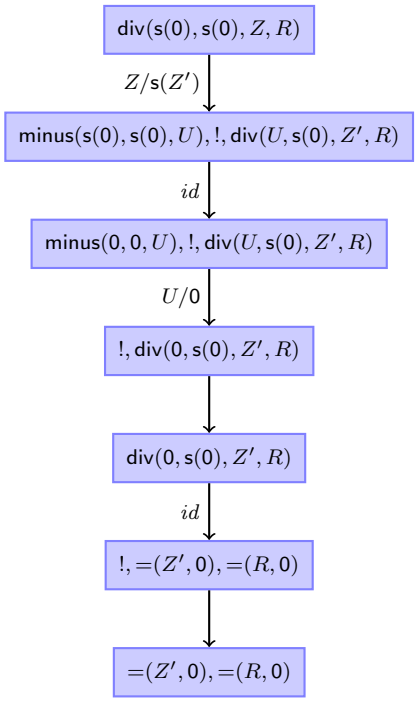
Now, we have a built-in predicate as the first predication of the current goal and execute its side effect which is to cut off all unvisited nodes between the current node and the node where it was introduced (which is the first node). Hence, we drop the right successor node of the first node and add a successor without the cut to the current node. This new node becomes the next current node.



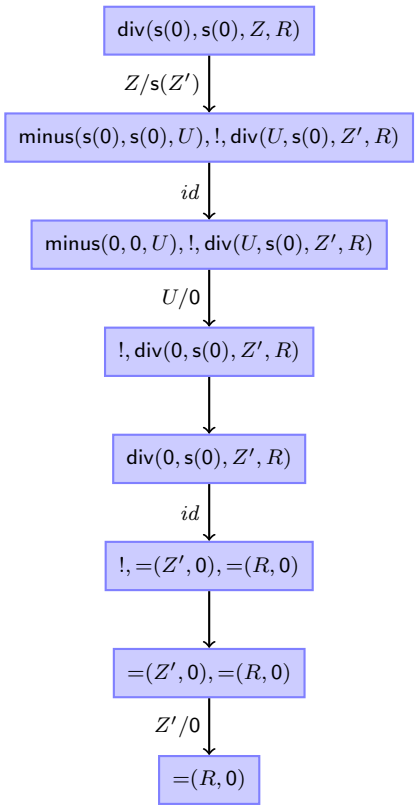
Next, we have three unifying clauses for the first predication in the current goal and add the respective nodes to the current node. The first new node becomes the current node while the other new nodes form the set of unvisited nodes.



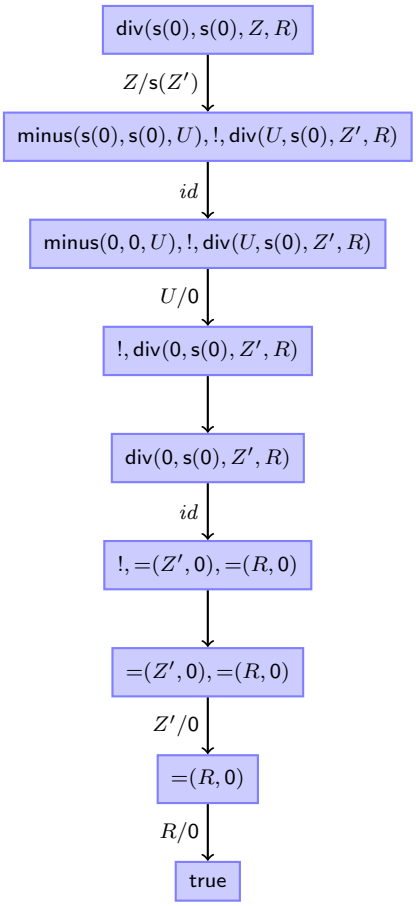
We evaluate a cut again and drop the just introduced unvisited nodes. We add another node to the current node which becomes the new current node.



Now we execute another built-in predicate to unify Z' with $\mathsf{0}$.



We repeat this for R and $\mathsf{0}$.



Finally, the current goal is true and we backtrack to the next unvisited node. Since the set of unvisited nodes is empty, the computation stops.

5.3 Representing Prolog Search-Trees

After establishing the operational semantics of Prolog according to the ISO standard by means of an algorithm, we show how to simulate the execution of this algorithm by concrete state-derivations. To this end, we define which Prolog search-tree is represented by a concrete state-derivation. We use an inductive definition for the represented Prolog search-trees as this kind of definition is useful for the inductive proof of the central theorem of this chapter. The general idea of the representation is that state elements in our states correspond to nodes in the Prolog search-tree. The first state element represents the current node while the remaining state elements correspond to the unvisited nodes in the tree. Unfortunately, due to the separation of the evaluation in the rules Case, Backtrack, Eval and Failure, this general idea models not exactly the connection between concrete state-derivations and Prolog search-trees. Instead, we have some state elements in our states which do not correspond to any node as they will be just dropped by applying the Backtrack or Failure rule. Likewise, the nodes in a Prolog search-tree do

not contain goals labeled with clauses, but contain only the evaluated goals we will reach after the application of the EVAL rule. To overcome this difference, we pre-perform the applications of the BACKTRACK, EVAL and FAILURE rules when applying the CASE rule for the represented goals in the nodes. Then, we just ignore the applications of the former three rules for the represented Prolog search-trees. For built-in predicates introducing question marks or additional state elements we use similar adaptions in the representation to only represent those nodes which will really be constructed by the algorithm from Definition 5.6. Finally, a further small difference is that the empty goal $\square$ in our setting represents the successful proved goal true in a Prolog search-tree.

Definition 5.8 (Represented Prolog Search-Tree)**.** *Given a concrete state-derivation from an initial concrete state S_0 for a goal as defined in Definition 3.9 leading to the concrete states $S_1, S_2, \ldots$, the Prolog search-tree T represented by the concrete state-derivation is inductively defined as follows:*

- *The root node of T is labeled with the unlabeled term in S_0 and the empty substitution. The set of unvisited nodes is initially empty. For concrete state-derivations not using any rules from TreeRules, the root node is also the current node.*

- *If the state S_i is reached from S_{i-1} by applying the* ATOMICFAIL, COMPOUNDFAIL, EQUALSFAIL, FAIL, NONVARFAIL, NOUNIFYFAIL, SUCCESS, UNEQUALSFAIL, UNIFYFAIL *or* VARFAIL *rule, consider the set of unvisited nodes. If it is not empty, let n be the first node in this set w.r.t. the depth-first left-to-right ordering of the nodes in T. Drop n from the set of unvisited nodes. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i and the set of unvisited nodes was not empty before, the current node is n. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i and the set of unvisited nodes was empty before, there is no current node.*

- *If the state S_i is reached from S_{i-1} by applying the* ATOMICSUCCESS *rule where the first state element of S_{i-1} is $\mathsf{atomic}(t), Q$, add a child to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* CALL *rule where the first state element of S_{i-1} is $\mathsf{call}(t), Q$, $t \notin \mathcal{N}$ and t has only finitely many predication positions, add a child to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with $Transformed(t, m)[!_m/!], Q$ and the empty substitution. If the first label of the child*

is $\Box$, replace this label with true*. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Case *rule which introduced the labeled goals $(t, Q)^{c_1}_m, \ldots, (t, Q)^{c_k}_m$ for the freshly renamed clauses $H_{c_1} \leftarrow B_{c_1}, \ldots, H_{c_k} \leftarrow B_{c_k}$ and the first state element of S_{i-1} is the unlabeled goal t, Q for $t \in PrologTerms(\Sigma, \mathcal{V})$ and Q being a sequence of terms, let $\{j_1, \ldots, j_l\} \subseteq \{c_1, \ldots, c_k\}$ with $j_1 < \cdots < j_l$ and $\forall j \in \{c_1, \ldots, c_k\} : t \sim H_j \implies j \in \{j_1, \ldots, j_l\}$. If $l > 0$, add l children to the current node of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1}. The n-th child is labeled with $B'_{j_n}\sigma, Q\sigma$ and σ for $n \in \{1, \ldots, l\}$ where B'_{j_n} is $Transformed(B_{j_n}, m)[!_m/!]$ and $\sigma = mgu(t, H_{j_n})$. If the first label of a child is $\Box$, replace this label with* true*. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the first child added unless $l = 0$. Add all children except for the first to the set of unvisited nodes. If $l = 0$, consider the set of unvisited nodes. If it is not empty, let n be the first node in this set w.r.t. the depth-first, left-to-right ordering of the nodes in T. Drop n from the set of unvisited nodes. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i and the set of unvisited nodes was not empty before, the current node is n. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i and the set of unvisited nodes was empty before, there is no current node and the represented* Prolog *search tree is finished.*

- *If the state S_i is reached from S_{i-1} by applying the* CompoundSuccess *rule where the first state element of S_{i-1} is* compound$(t), Q$*, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \Box$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Conjunction *rule where the first state element of S_{i-1} is $,(t_1, t_2), Q$, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with t_1, t_2, Q and the empty substitution. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Cut *rule where the first state element of S_{i-1} is $!_m, Q$, there must be a rule from the set {*Call, Case, IfThen,

IfThenElse, Not} *in the concrete state-derivation just leading to* S_{i-1} *which introduced this labeled cut* $!_m$ *and where we added some nodes to the node* n *in* T*. Delete all nodes in the set of unvisited nodes in* T *between the current node* n' *of the* Prolog *search-tree represented by the concrete state-derivation just leading to* S_{i-1} *and the node* n *(inclusive). Also, drop these nodes from the set of unvisited nodes. Then add a node to* n' *labeled with* Q *if* $Q \neq \Box$ *and* true *otherwise and the empty substitution. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child of* n'*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* CutAll *rule where the first state element of* S_{i-1} *is* $!_m, Q$*, delete all nodes in the set of unvisited nodes in* T*, let the set of unvisited nodes be empty and add a node to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *which is labeled by* Q *if* $Q \neq \Box$ *and* true *otherwise and the empty substitution. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child of* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* Disjunction *rule where the first state element of* S_{i-1} *is* $;(t_1, t_2), Q$*, add two children to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* t_1, Q *and the empty substitution and* t_2, Q *and the empty substitution. Add the second child to the set of unvisited nodes. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the first child added to* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* EqualsSuccess *rule where the first state element of* S_{i-1} *is* $==(t_1, t_1), Q$*, add a child to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* Q *and the empty substitution. If* $Q = \Box$*, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child added to* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* Halt*,* Halt1*,* UndefinedError *or* VariableError *rule, there is no current node, there are no unvisited nodes and the represented* Prolog *search tree is finished.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* IfThen *rule where the first state element of* S_{i-1} *is* $->(t_1, t_2), Q$*, add a child to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* $\mathsf{call}(t_1), !, t_2, Q$ *and the empty substitution. If the concrete state-*

derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.

- *If the state S_i is reached from S_{i-1} by applying the* IfThenElse *rule where the first state element of S_{i-1} is $;(->(t_1, t_2), t_3), Q$, add two children to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with $\mathsf{call}(t_1), !, t_2, Q$ and the empty substitution and t_3, Q and the empty substitution. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the first child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Newline *rule where the first state element of S_{i-1} is nl, Q, add a child to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* NonvarSuccess *rule where the first state element of S_{i-1} is $\mathsf{nonvar}(t), Q$, add a child to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Not *rule where the first state element of S_{i-1} is $\backslash+(t), Q$, add two children to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with $\mathsf{call}(t), !, \mathsf{fail}$ and the empty substitution and Q and the empty substitution. If $Q = \square$, the second child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the first child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* NoUnifySuccess *rule where the first state element of S_{i-1} is $\backslash=(t_1, t_2), Q$, add a child to the current node n of the Prolog search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Once *rule where the first state element of S_{i-1} is* $\mathsf{once}(t), Q$*, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with* $\mathsf{call}(t, !), Q$ *and the empty substitution. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* Repeat *rule where the first state element of S_{i-1} is* repeat, Q*, add two children to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution and* repeat, Q *and the empty substitution. If $Q = \square$, the first child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the first child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* True *rule where the node n represented by the prefix of the concrete state-derivation just leading to S_{i-1} starts with a state element having* true *as its first term, delete this term from the first state element in n. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is n.*

- *If the state S_i is reached from S_{i-1} by applying the* UnequalsSuccess *rule where the first state element of S_{i-1} is* $\backslash==(t_1, t_2), Q$*, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* UnifySuccess *rule where the first state element of S_{i-1} is* $=(t_1, t_2), Q$*, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state S_i, the current node is the child added to n.*

- *If the state S_i is reached from S_{i-1} by applying the* VarSuccess *rule where the first state element of S_{i-1} is* $\mathsf{var}(x), Q$*, add a child to the current node n of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to S_{i-1} labeled with Q and the empty substitution. If $Q = \square$, the child is labeled with*

true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child added to* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* Write *rule where the first state element of* S_{i-1} *is* $\mathsf{write}(t), Q$*, add a child to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* Q *and the empty substitution. If* $Q = \square$*, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child added to* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* WriteCanonical *rule where the first state element of* S_{i-1} *is* $\mathsf{write\_canonical}(t), Q$*, add a child to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* Q *and the empty substitution. If* $Q = \square$*, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child added to* n*.*

- *If the state* S_i *is reached from* S_{i-1} *by applying the* Writeq *rule where the first state element of* S_{i-1} *is* $\mathsf{writeq}(t), Q$*, add a child to the current node* n *of the* Prolog *search-tree represented by the prefix of the concrete state-derivation just leading to* S_{i-1} *labeled with* Q *and the empty substitution. If* $Q = \square$*, the child is labeled with* true *and the empty substitution instead. If the concrete state-derivation does not use any rules from TreeRules after reaching the state* S_i*, the current node is the child added to* n*.*

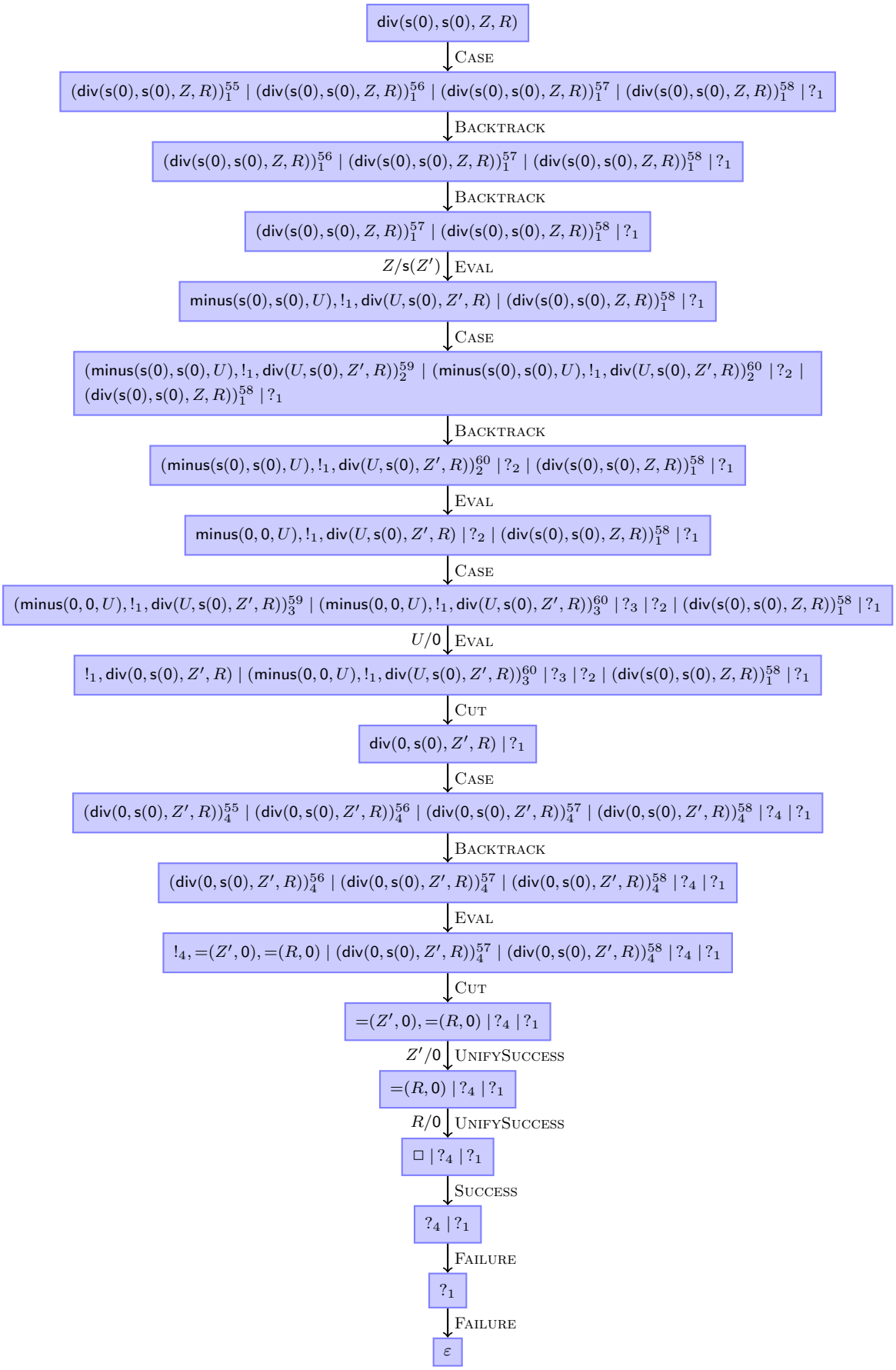
- *If the state* S_i *is reached from* S_{i-1} *by applying the* Backtrack, Eval *or* Failure *rule, the represented* Prolog *search-tree remains unchanged.*

Here, the labels for the nodes correspond to the current goals and local substitutions of the nodes in a Prolog *search-tree.*

The set $\mathit{TreeRules} \subset \mathit{ConcreteInferenceRules}$ *is defined as:*

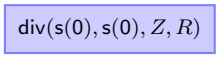
$$\mathit{ConcreteInferenceRules} \setminus \{\textsc{Backtrack}, \textsc{Eval}, \textsc{Failure}\}$$

Example 5.9. Consider once more the Prolog program $\mathcal{P}$ for division with remainder from Example 4.22 and the query $\mathsf{div}(\mathsf{s}(0), \mathsf{s}(0), Z, R)$. The concrete state-derivation for this query is as follows.

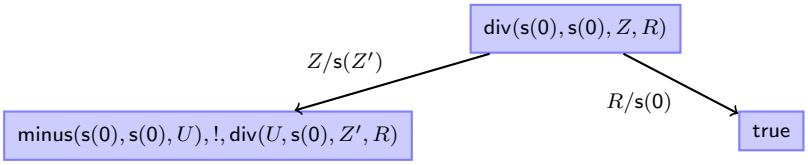


The Prolog search-tree represented by this concrete state-derivation is exactly the Prolog

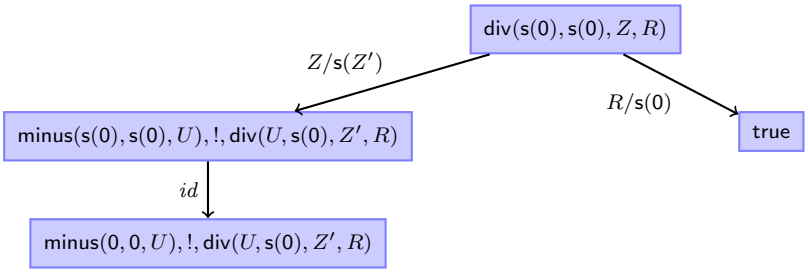
search-tree from Example 5.5. To see this, we now follow the steps of the inductive definition. We start with the initial node as the current node labeled with the initial goal and the empty local substitution and no unvisited nodes as for Prolog search-trees. Thus, the empty concrete state-derivation represents the Prolog search-tree consisting only of the initial node.



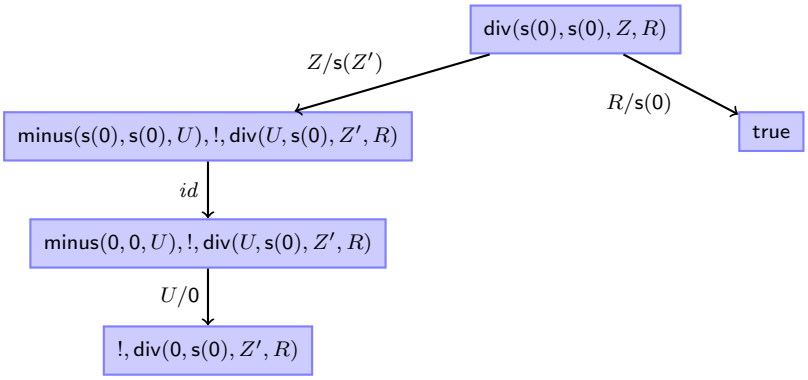
Now the concrete state-derivation applies the Case rule. Hence, we add nodes to the tree corresponding to unifying clauses in the program. The first added node becomes the current node and the second one forms the set of unvisited nodes.



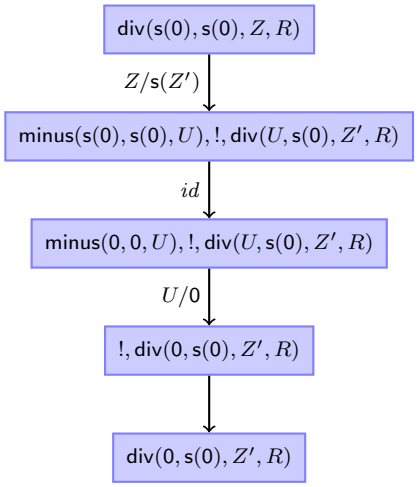
The next three applications of the rules Backtrack and Eval do not modify the represented Prolog search-tree. Then we apply the Case rule again and add a node corresponding to the only unifying clause to the tree. This node becomes the current node.



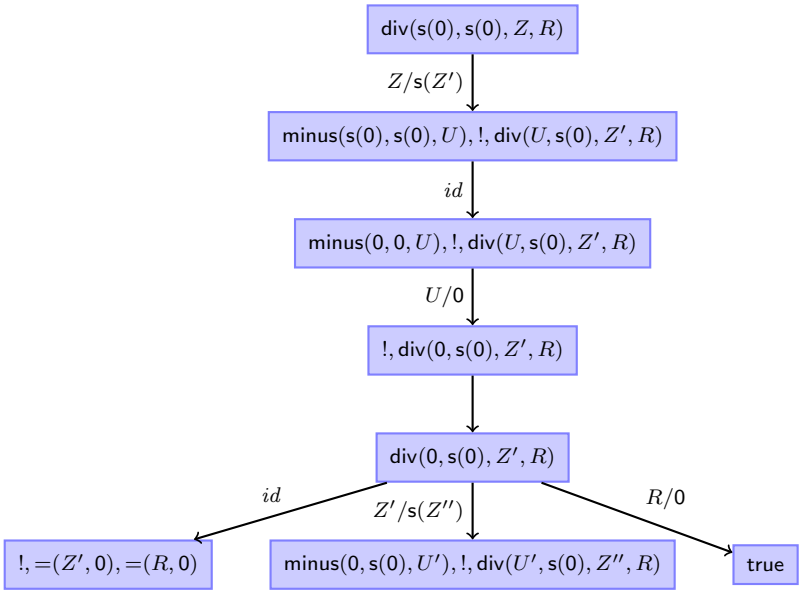
The next two applications of the rules Backtrack and Eval do not modify the represented Prolog search-tree again. We apply the Case rule once more and add a node corresponding to the only unifying clause to the tree which becomes the current node.



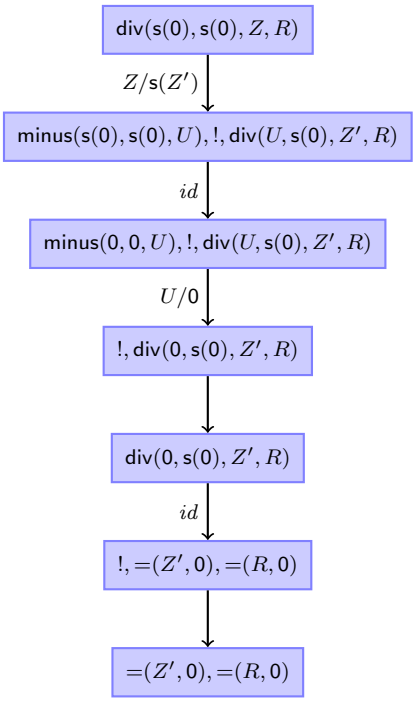
The following application of the Eval rule has no effect on the represented Prolog search-tree. Now we apply the Cut rule and delete all unvisited nodes between the current node and the node where the respective cut was introduced. Here, this is the first node. Then we add a new node without the cut to the current node. This new node becomes the current node.



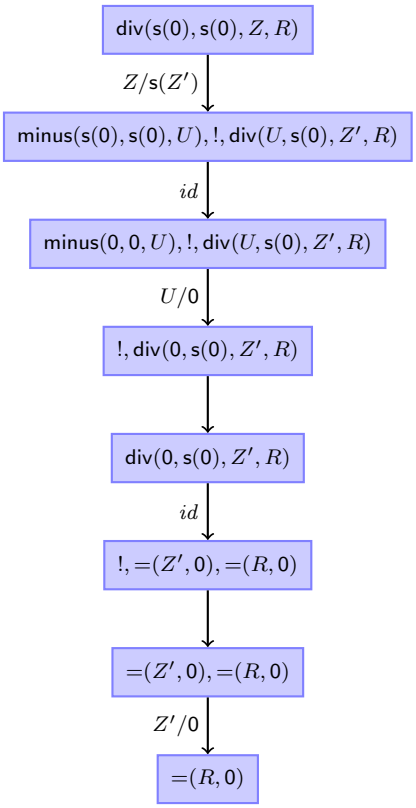
Next, we apply the Case rule again and obtain three new nodes for three unifying clauses. The first new node becomes the current node while the other new nodes are added to the (currently empty) set of unvisited nodes.



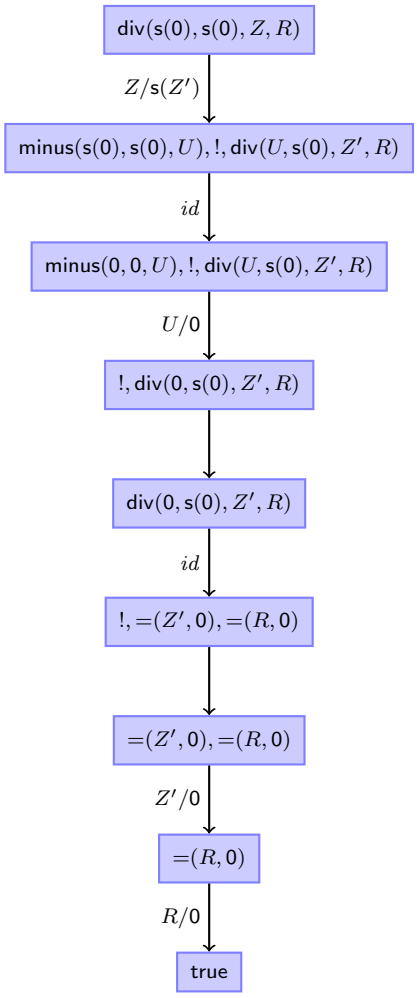
The following two applications of the Backtrack and Eval rules do not mandate any changes to the represented Prolog search-tree. Again, we apply the Cut rule and delete the recently introduced unvisited nodes from the tree. We add another node to the current node which becomes the next current node.



Now we apply the UnifySuccess rule and add a node to the tree labeled with the answer substitution $[Z'/0]$.



We repeat this for the unification of R and 0. As the remaining goal is $\square$, we replace the node label with true.



Now we apply the Success rule and backtrack to the next unvisited node. Since the set of unvisited nodes is empty, the represented Prolog search-tree is finished and there is no current node. The following two applications of the Failure rule do not modify the represented Prolog search-tree anymore. As we can see, the construction of the Prolog search-tree is absolutely identical to Example 5.7. Also, note that after application of the Backtrack and Eval rules we have that the first state elements of the states in the concrete state-derivation correspond exactly to the nodes in the Prolog search-tree if we omit the scopes for the cuts.

Before we state the central theorem of this chapter, we prove some useful properties of our representation. As described before Definition 5.8, the state elements which are not dropped by the Backtrack or Failure rule correspond to the unvisited nodes in the represented Prolog search-tree, possibly after applying the Eval rule to them. Hence, we reach the next unvisited node from a state where the current node has been evaluated, i.e., the first state element has been dropped, by only applying these three rules which do not change the represented Prolog search-tree. Likewise, we also prove the intended connection between the empty goal $\square$ and the successful proved goal true.

Lemma 5.10 (First State Elements for Current Nodes)**.** *Given a Prolog search-tree T and a concrete state-derivation w.r.t. a Prolog program $\mathcal{P}$ leading to a state S which represents T, then we can continue the concrete state-derivation from S to a state S' such that the following conditions hold:*

- *If the current goal of the current node of T is* true*, then the first state element S_0 of S' is $\square$.*
- *Otherwise the first state element S_0 of S' contains a goal Q which is the current goal of the current node of T when all labeled cuts in Q are replaced by unlabeled cuts.*
- *The continued concrete state-derivation still represents T, i.e., the continuation only uses rules from the set* {Backtrack, Eval, Failure}*.*
- *For all unvisited nodes $n_1, \ldots, n_u$ in T there are state elements $S_1, \ldots, S_u \in$ StateElements such that $S' = S_0 \mid S_{r,0} \mid S_1 \mid S_{r,1} \mid \ldots \mid S_u \mid S_{r,u}$ where S_i can be evaluated by only using the rules from the set* {Backtrack, Eval, Failure} *to a goal Q_i which is the current goal of n_i when all labeled cuts in Q_i are replaced by unlabeled cuts and $S'_i = S_i \mid S_{r,i} \mid \ldots \mid S_u \mid S_{r,u}$ is reachable from $S_{r,i-1} \mid S'_i$ by only applying rules from the set* {Backtrack, Eval, Failure}*.*
- *If there are no unvisited nodes, let $S' = S_0 \mid S_{r,0}$. Then the state ε is reachable from $S_{r,0}$ by only applying rules from the set* {Backtrack, Eval, Failure}*.*

Proof. We perform the proof by induction over the length k of the concrete state-derivation to the state S.

For $k = 0$ we have that S contains the initial goal which is the current goal of the current node of T. Hence, we have $S = S'$ and the lemma trivially holds as there are no unvisited nodes and S' contains only one state element.

For $k > 0$ we can assume the lemma holds for concrete state-derivations of length at most $k - 1$. We perform a case analysis over the last concrete rule in the concrete state-derivation.

- For Case we have that $S = (t, Q)^{i_1}_m \mid \ldots \mid (t, Q)^{i_j}_m \mid ?_m \mid S_r$ with $|\mathit{Slice}(\mathcal{P}, t)| = j$. There are two cases depending on whether there is an index $l \in \{1, \ldots, j\}$ with $H_{i_l} \leftarrow B_{i_l} \in \mathcal{P}$ and $H_{i_l} \sim t$. If there is at least one such index, let l be the smallest index with this property. Then we have to apply the Backtrack rule for the first $l - 1$ state elements of S. Thus, we reach the state $(t, Q)^{i_l}_m \mid \ldots \mid (t, Q)^{i_j}_m \mid ?_m \mid S_r$. Now we apply the Eval rule and obtain the state $S' = B'_{i_l}\sigma, Q\sigma \mid \ldots \mid (t, Q)^{i_j}_m \mid ?_m \mid S_r$ where $B'_{i_l} = \mathit{Transformed}(B_{i_l}, m)$ and $\sigma = \mathit{mgu}(t, H_{i_l})$. If we added new nodes to the set of unvisited nodes in T by applying the Case rule, there are state elements $(t, Q)^{i_{l'}}_m$ in S' which can be evaluated to the desired state elements by applying the

Eval rule. As all other state elements introduced by the Case rule can be dropped using the rules Backtrack and Failure and we have suitable state elements for the not newly introduced unvisited nodes in S_r by the induction hypothesis or we reach the state ε from S_r by the induction hypothesis if no unvisited nodes exist, the lemma still holds.

If there is no such index l, we have to apply the Backtrack rule to the first j state elements of S. Afterwards, we apply the Failure rule to $?_m$. There are two cases depending on whether there are unvisited nodes in T. If there are unvisited nodes, we obtain by the induction hypothesis that we reach the desired state S' for the first unvisited node being the next current node from S_r using no other rules than those from the set $\{\text{Backtrack}, \text{Eval}, \text{Failure}\}$. If there are no unvisited states, we know by the induction hypothesis that S_r can be evaluated to ε using only the rules from $\{\text{Backtrack}, \text{Eval}, \text{Failure}\}$. Hence, the lemma still holds.

- For AtomicFail, CompoundFail, EqualsFail, Fail, NonvarFail, NoUnifyFail, Success, UnequalsFail, UnifyFail and VarFail we just drop the first state element from the state we reached just before S. Thus, all conditions are implied by the induction hypothesis.

- For AtomicSuccess, CompoundSuccess, Conjunction, EqualsSuccess, Newline, NonvarSuccess, NoUnifySuccess, Once, UnequalsSuccess, UnifySuccess, VarSuccess, Write, WriteCanonical and Writeq we have $S = S'$ and all conditions are implied by the induction hypothesis.

- For Backtrack, Eval and Failure we obtain all conditions trivially by the induction hypothesis as these rules do not modify T.

- For Call we have that $S = t, Q \mid ?_m \mid S_r = S'$. Additionally, we reach the state S_r from $?_m \mid S_r$ by applying the Failure rule. We obtain the remaining conditions by the induction hypothesis.

- For Cut we have that $S = Q \mid ?_m \mid S_r = S'$. As we dropped all unvisited nodes having corresponding state elements before S_r, we know that the possibly remaining unvisited nodes have corresponding state elements in S_r. As we reach S_r from $?_m \mid S_r$ by applying the Failure rule, the remaining conditions follow by the induction hypothesis.

- For CutAll we have $S = Q = S'$ and we dropped all unvisited nodes. As S' contains only one state element, the lemma trivially holds.

- For Disjunction we have that $S = t_1, Q \mid t_2, Q \mid S_r = S'$. We directly reach a suitable state for the first unvisited node from $t_2, Q \mid S_r$. The remaining conditions follow by the induction hypothesis.

- For HALT, HALT1, THROW, UNDEFINEDERROR and VARIABLEERROR we have $S = \varepsilon = S'$ and no current node or unvisited nodes. Thus, the lemma trivially holds.
- For IFTHEN we have that $S = \mathsf{call}(t_1), !_m, t_2, Q \mid ?_m \mid S_r = S'$. Additionally, we reach the state S_r from $?_m \mid S_r$ by applying the FAILURE rule. We obtain the remaining conditions by the induction hypothesis.
- For IFTHENELSE we have that $S = \mathsf{call}(t_1), !_m, t_2, Q \mid t_3, Q \mid ?_m \mid S_r = S'$. We directly reach a suitable state for the first unvisited node from $t_3, Q \mid ?_m \mid S_r$. The state S_r is reachable from $?_m \mid S_r$ by applying the FAILURE rule. The remaining conditions follow by the induction hypothesis.
- For NOT we have that $S = \mathsf{call}(t), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S_r = S'$. We directly reach a suitable state for the first unvisited node from $Q \mid ?_m \mid S_r$. The state S_r is reachable from $?_m \mid S_r$ by applying the FAILURE rule. The remaining conditions follow by the induction hypothesis.
- For REPEAT we have that $S = Q \mid \mathsf{repeat}, Q \mid S_r = S'$. We directly reach a suitable state for the first unvisited node from $\mathsf{repeat}, Q \mid S_r$. The remaining conditions follow by the induction hypothesis.
- For TRUE we have that $S = S'$ emerges from the state reached before by just dropping the first term true from the goal in the first state element. Thus, all conditions are implied by the induction hypothesis.

□

Now we are ready to prove our central theorem of this chapter which connects executions of Prolog programs according to the ISO standard with concrete state-derivations represented by termination graphs. The assumptions we make in this theorem correspond to the default configurations of most Prolog implementations.

Theorem 5.11 (Operational Semantics for Standard Prolog without Unhandled Predicates)**.** *Given a standard conforming Prolog processor using unification without occurs-check with the flag* unknown *set to* error *as defined in [DEC96], a Prolog program $\mathcal{P}$ not using any built-in predicates from UnhandledBuiltInPredicates or other built-in predicates than those defined in [DEC96] and a goal Q, for the execution of Q w.r.t. $\mathcal{P}$ by the processor, i.e., for the Prolog search-tree as defined for the execution model in [DEC96], there is a concrete state-derivation using the rules from Definition 5.1 representing this Prolog search-tree, assuming that an error or the achieved execution of Q lead to the termination of the execution without manipulating the search-tree obtained before achieving the execution or throwing the error and that we never have to transform terms with infinitely many predication positions due to meta-calls during the execution.*

Proof. We perform the proof by induction over the number k of execution steps in the execution model.

For $k = 1$ we start from the root as current node, labeled by the initial goal and the empty substitution with no unvisited nodes. Thus, the empty concrete state-derivation from the state S_0 containing the initial goal represents this Prolog search-tree.

For $k > 1$ we can assume the theorem holds for executions with at most $k - 1$ steps. In particular, we obtain a concrete state-derivation leading from the state S_0 to a state S representing the Prolog search-tree T reached before executing the last step of the execution by the induction hypothesis. By Lemma 5.10 we can w.l.o.g. assume that the first state element of S contains the current goal of the current node of T where the goal $\square$ as the first state element of S corresponds to the current goal true of the current node of T and labeled cuts are replaced by unlabeled cuts. We perform a case analysis over the last execution step.

- If the last execution step corresponds to step 2 of the search-tree visit and construction algorithm, the current node of T is labeled with true. Therefore, the first state element of S must be $\square$. Thus, we apply the SUCCESS rule to S and the represented Prolog search-tree changes its current node to the next unvisited node in T while dropping this node from the set of unvisited nodes. Hence, we represent the Prolog search-tree reached by the last execution step.

- If the last execution step corresponds to step 4 of the search-tree visit and construction algorithm, the current node of T is labeled with a goal having true as its first predication and at least two predications in total. Therefore, the first state element of S must start with the term true. Thus, we apply the TRUE rule to S and the current node of the represented Prolog search-tree is changed according to step 4 of the search-tree visit and construction algorithm. Hence, we represent the Prolog search-tree reached by the last execution step.

- If the last execution step corresponds to step 5a of the search-tree visit and construction algorithm, the current node of T is labeled with a goal of the form t, Q with $\mathit{Slice}(\mathcal{P}, t) \neq \varnothing$ and there is no clause $H \leftarrow B \in \mathit{Slice}(\mathcal{P}, t)$ with $t \sim H$. Therefore, the first state element of S must be the goal t, Q. Thus, we apply the CASE rule to S and the current node of the represented Prolog search-tree is changed to the first node n in the set of unvisited nodes and n is dropped from this set. Hence, we represent the Prolog search-tree reached by the last execution step.

- If the last execution step corresponds to step 5b of the search-tree visit and construction algorithm, the current node of T is labeled with a goal of the form t, Q with $\mathit{Slice}(\mathcal{P}, t) \neq \varnothing$ and there is at least one clause $H \leftarrow B \in \mathit{Slice}(\mathcal{P}, t)$ with $t \sim H$. Therefore, the first state element of S must be the goal t, Q. Thus, we apply the

CASE rule to S and the represented Prolog search-tree is modified by adding children to the current node of T for each clause $H \leftarrow B \in \mathit{Slice}(\mathcal{P}, t)$ with $t \sim H$ in the order of the clauses with the label $B'\sigma, Q\sigma$ and σ where $B' = \mathit{Transformed}(B, m)[!_m/!]$ for some $m \in \mathbb{N}$ and $\sigma = \mathit{mgu}(t, H)$ while the current node becomes the first child of the current node of T. Hence, we represent the Prolog search-tree reached by the last execution step.

- If the last execution step corresponds to step 6 of the search-tree visit and construction algorithm, the current node of T is labeled with a goal of the form t, Q with $\mathit{root}(t) \in \mathit{BuiltInPredicates}$. As we assume that no built-in predicates other than those in *HandledBuiltInPredicates* are used in $\mathcal{P}$ and the initial query, we obtain that $\mathit{root}(t) \in \mathit{HandledBuiltInPredicates}$. By inspection of Definition 5.8 we obtain that each rule for a built-in predicate from $\mathit{HandledBuiltInPredicates} \setminus \{\mathsf{nl/0}, \mathsf{throw/1}, \mathsf{write/1}, \mathsf{write\_canonical/1}, \mathsf{writeq/1}\}$ modifies T as defined in [DEC96]. For throw/1 note that we do not use the built-in predicate catch/3. Thus, the execution of throw/1 will always cause a system error or an instantiation error. In both cases the computation will terminate without modifying the Prolog search-tree according to our assumptions. For nl/0, write/1, write_canonical/1 and writeq/1 note that we do not use any built-in predicates capable of changing the current output stream. Thus, it must be the default output stream which is a text stream. Hence, these built-in predicates cannot cause an error according to their definition in [DEC96] and they modify the Prolog search-tree as defined in Definition 5.8.

- If the last execution step corresponds to step 7 of the search-tree visit and construction algorithm, the current node of T is labeled with a goal of the form t, Q with $\mathit{Slice}(\mathcal{P}, t) = \varnothing$ and $\mathit{root}(t) \notin \mathit{BuiltInPredicates}$. Therefore, the first state element of S must be the goal t, Q and we apply the UNDEFINEDERROR rule to S. As we assume that T is not modified in case of an error and the computation terminates, we represent the Prolog search-tree reached by the last execution step.

As step 3 of the search-tree visit and construction algorithm does not perform an execution step, we have considered all cases for the last execution step and, thus, proved the theorem. □

5.4 Summary

We stated the operational semantics of Prolog as defined in the ISO standard and introduced a representation of Prolog search-trees by concrete state-derivations. Using this representation we showed that we can simulate any execution of a Prolog program w.r.t. a query not using built-in predicates we cannot handle in our setting, assuming the default configuration of most Prolog implementations. Thus, we have shown that our approach is capable of analyzing real Prolog applications.

6 Deterministic Construction of Termination Graphs

So far, we have introduced concrete and abstract inference rules to construct termination graphs for Prolog programs w.r.t. a class of queries. Furthermore, we showed that the concrete inference rules can be used to simulate the operational semantics of Prolog according to the ISO standard [DEC96]. However, up to now we have applied the abstract rules non-deterministically to obtain termination graphs. The reason for this is that the rules for obtaining finite graphs overlap with the rules simulating the concrete state-derivations. In order to have a fully mechanizable method, we need to make the application of abstract inference rules deterministic, i.e., we need an algorithm which decides when to use which abstract inference rule. In other words, we need to decide when to evaluate an abstract state and when to try to find instances, i.e., close the graph. For the latter we also have to give a strategy how to find instances for the current abstract state. In particular, this strategy must decide how and when to generalize or split abstract states to find instances for its successors if there is no instance father for the respective abstract state itself.

Structure of the Chapter

In Section 6.1 we introduce the notion of (partial) termination graphs and give some example graphs for the two example programs even.pl and divremain.pl from the beginning of Chapter 3 where we equivalently modified the latter by using built-in predicates.

Section 6.2 deals with the deterministic construction of termination graphs for a given Prolog program and a class of queries by using a heuristic. After introducing the relevant notions and illustrating them with examples we present the standard heuristic which has shown to be successful on a great number of examples from the TPDB.

After its presentation we prove in Section 6.3 that this heuristic is always terminating and constructs either a termination graph or fails due to the situations where our abstract inference rules are stuck (see the explanations after Definition 3.11 and Definition 3.21).

Finally, we summarize the contributions of this chapter in Section 6.4.

6.1 Termination Graphs

For a graph G and a rule RULE, we use the notation $Rule(G)$ to denote all nodes of G to which RULE has been applied. We denote by $Succ(i, n)$ the i-th child of n and by $Succ(i, Rule(G))$ the set of i-th children of nodes from $Rule(G)$.

We start by defining what graphs are considered being termination graphs.

Definition 6.1 (Termination Graph)**.** *A finite graph built from an initial state* $(s; KB)$ *using the rules from Definitions 3.21 – 3.50 is called a* termination graph *if, and only if, there is no cycle only consisting of* INSTANCE *and* GENERALIZATION *nodes and all leaves are of the form* $(\varepsilon; KB')$ *for some arbitrary knowledge bases* KB'*.*

First, we must not have cycles consisting only of INSTANCE and GENERALIZATION nodes as this would lead to a cycle without any concrete evaluation and, thus, this cycle could trivially be repeated infinitely often.

Second, we must have applied some rule to all nodes of the graph except for empty states, since every concrete evaluation must stop at such a state if it stops at all.

Example 6.2. Consider again the graph from Example 3.39. All leaves are abstract states of the form $\varepsilon; (\varnothing, \varnothing, \varnothing)$. The only cycle traversing the only INSTANCE node $\mathsf{c}(\mathsf{f}(\mathsf{e}, \mathsf{f}(\mathsf{o}, **)), T_4); (\{T_4\}, \varnothing, \varnothing)$ also contains, for instance, the CASE node $\mathsf{c}(\mathsf{f}(\mathsf{e}, \mathsf{f}(\mathsf{o}, **)), T_2); (\{T_2\}, \varnothing, \varnothing)$. Thus, this graph is indeed a termination graph.

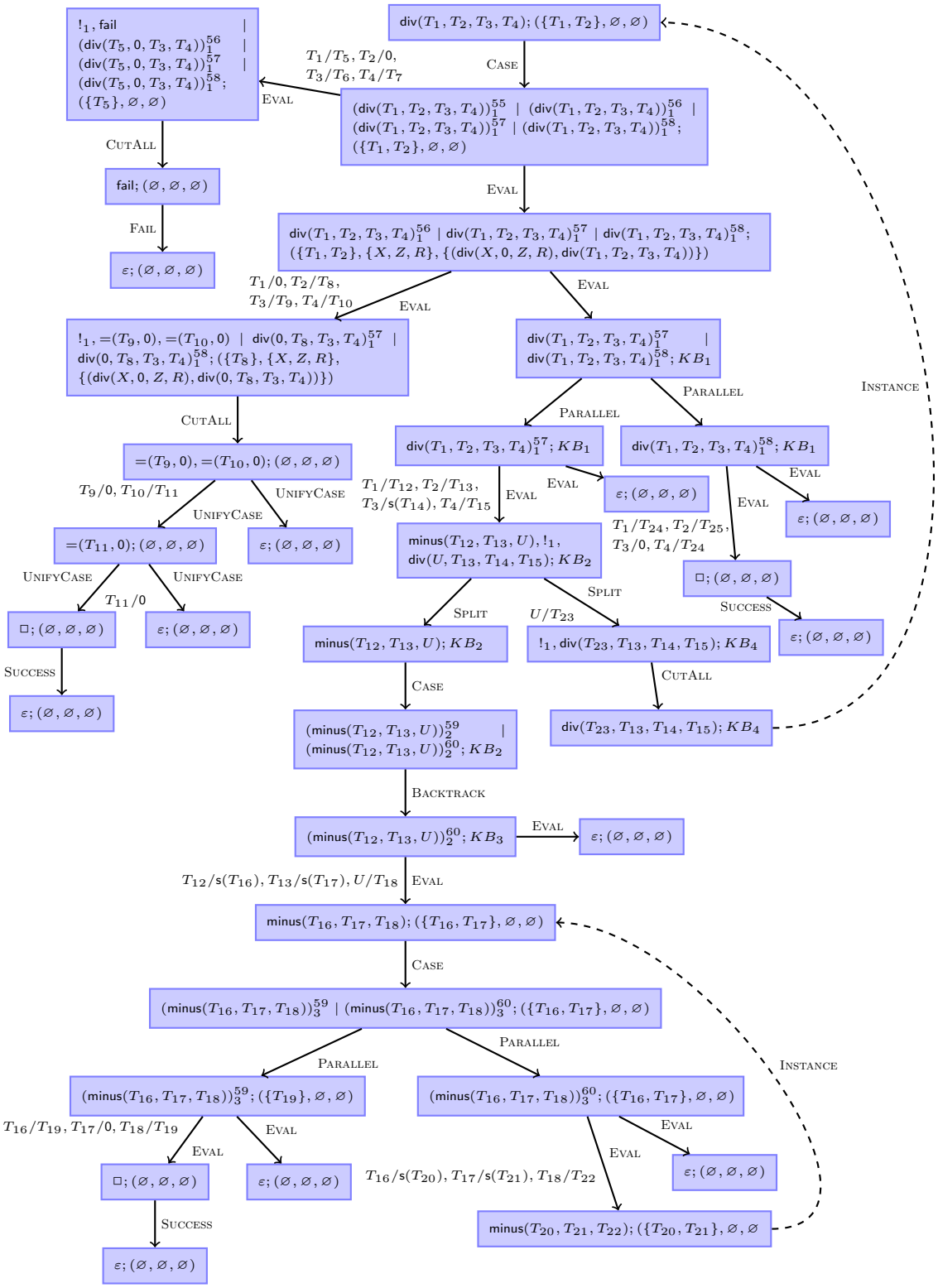
Example 6.3. However, the termination graph from Example 6.2 does not use the rules PARALLEL and SPLIT. We give a more complex example of a termination graph where we have to use these rules to obtain a finite graph. Consider again the Prolog program divremain.pl in its modified form according to built-in predicates from Example 4.22. We can construct the following termination graph for the query set $\mathcal{Q}$ as defined in Example 3.1. To save space, we define the following knowledge bases used in this graph.

$KB_1 = (\{T_1, T_2\}, \{X, Y, Z, R\}, \{(\mathsf{div}(X, \mathsf{0}, Z, R), \mathsf{div}(T_1, T_2, T_3, T_4)), (\mathsf{div}(\mathsf{0}, Y, Z, R), \mathsf{div}(T_1, T_2, T_3, T_4))\})$,

$KB_2 = (\{T_{12}, T_{13}\}, \{U, X, Y, Z, R\}, \{(\mathsf{div}(X, \mathsf{0}, Z, R), \mathsf{div}(T_{12}, T_{13}, T_3, T_4)), (\mathsf{div}(\mathsf{0}, Y, Z, R), \mathsf{div}(T_{12}, T_{13}, T_3, T_4))\})$,

$KB_3 = (\{T_{12}, T_{13}\}, \{U, X, Y, Z, R\}, \{(\mathsf{div}(X, \mathsf{0}, Z, R), \mathsf{div}(T_{12}, T_{13}, T_3, T_4)), (\mathsf{div}(\mathsf{0}, Y, Z, R), \mathsf{div}(T_{12}, T_{13}, T_3, T_4)), (\mathsf{minus}(X, \mathsf{0}, X), \mathsf{minus}(T_{12}, T_{13}, U))\})$,

$KB_4 = (\{T_{13}, T_{23}\}, \{X, Y, Z, R\}, \{(\mathsf{div}(X, \mathsf{0}, Z, R), \mathsf{div}(T_{23}, T_{13}, T_3, T_4)), (\mathsf{div}(\mathsf{0}, Y, Z, R), \mathsf{div}(T_{23}, T_{13}, T_3, T_4))\})$



Now, for the construction of such a termination graph, we start with an initial state given by the query set and expand this state according to the abstract inference rules from the preceding chapters. During this process, the graph is not a termination graph as in Definition 6.1, but a graph which can possibly be extended to a termination graph.

Definition 6.4 (Partial Termination Graph)**.** *A finite graph G built from an initial state $(s; KB)$ using the rules from Definitions 3.21 – 3.50 is called a* partial termination graph *if, and only if, there is no cycle only consisting of* Instance *and* Generalization *nodes.*

We denote the root state $(s; KB)$ of a (partial) termination graph G by $root(G)$.

Example 6.5. Every termination graph is also a partial termination graph. If we take only the finite part of the infinite graph from Example 3.30 shown there (without the dots), this is in fact a partial termination graph as it does not have any cycles at all.

6.2 The Standard Heuristic

In this section we show how to use the abstract inference rules deterministically to obtain a termination graph for a given Prolog program and query set.

Before we start to define the heuristic for the construction of a termination graph, we need the notion of *recursive symbols*. These are exactly the symbols for which we can potentially have infinite abstract state-derivations. To define recursive symbols in the presence of meta-programming, we first need a generalized notion for predications.

Definition 6.6 (Meta-Predication)**.** *Given a term $t \in \mathcal{T}^{rat}(\Sigma, \mathcal{V})$ and a position $pos \in Occ(t)$, we call $t|_{pos}$ a* meta-predication *w.r.t. t iff for all positions $pos' \in Occ(t)$ with $pos' \lhd pos$ we have $root(t|_{pos'}) \in GoalJunctors \cup \{\mathsf{call}/1, \backslash+/1, \mathsf{once}/1\}$. For a finite list L of terms $t_1, \ldots, t_k$ we also call $t_i|_{pos_i}$ a meta-predication w.r.t. L if $t_i|_{pos_i}$ is a meta-predication w.r.t. t_i.*

Definition 6.7 (Recursive Function Symbols and Clauses)**.** *We call a function symbol $f \in \Sigma \setminus BuiltInPredicates$* recursive *w.r.t. a Prolog program $\mathcal{P}$, if there is a sequence of clauses $H_0 \leftarrow B_0, H_1 \leftarrow B_1, \ldots, H_k \leftarrow B_k$ with $H_i \leftarrow B_i \in \mathcal{P}$ for all $i \in \{0, \ldots, k\}$, $root(H_0) = f$, there is a meta-predication t in B_{i-1} with $root(H_i) = root(t)$ for all $i \in \{1, \ldots, k\}$ and $((\exists j \in \{0, \ldots, k-1\} : root(H_j) = root(H_k)) \vee$ (there is a meta-predication t' in B_k with $root(t') = \mathsf{repeat}/0 \vee B_k$ has infinitely many meta-predications)). The built-in predicate $\mathsf{repeat}/0$ is additionally considered being recursive.*

We call a clause $H \leftarrow B \in \mathcal{P}$ recursive *w.r.t. $\mathcal{P}$ if B has a meta-predication t with $root(t)$ being recursive or B has infinitely many meta-predications.*

Example 6.8. Consider the following Prolog program $\mathcal{P}$:

$$\begin{aligned} \mathsf{p} &\leftarrow \mathsf{q}. \\ \mathsf{q} &\leftarrow \mathsf{q}. \\ \mathsf{a} &\leftarrow \mathsf{b}. \\ \mathsf{b} &\leftarrow \mathsf{c}. \\ \mathsf{c} &\leftarrow \square. \end{aligned}$$

As we have the sequence $\mathsf{p} \leftarrow \mathsf{q}, \mathsf{q} \leftarrow \mathsf{q}, \mathsf{q} \leftarrow \mathsf{q}$, the function symbols p and q are recursive w.r.t. $\mathcal{P}$, while the function symbols a, b and c are not recursive w.r.t. $\mathcal{P}$.

Also, the clauses $\mathsf{p} \leftarrow \mathsf{q}$ and $\mathsf{q} \leftarrow \mathsf{q}$ are recursive w.r.t. $\mathcal{P}$, while the clauses $\mathsf{a} \leftarrow \mathsf{b}$, $\mathsf{b} \leftarrow \mathsf{c}$ and $\mathsf{c} \leftarrow \square$ are not recursive w.r.t. $\mathcal{P}$.

We will now describe a heuristic which has shown to be successful on a great number of examples from the TPDB. Since it is designed to work on arbitrary examples and not for special purposes, we call it the *standard heuristic*. Some parts of this heuristic rely on parameters for which we will state values found to be advantageous in the empirical results section from Chapter 8.

Several ideas belong to the intuition behind this heuristic. First, we try to use shortening rules, i.e., rules which reduce the number of terms or state elements in a state without introducing any new terms or state elements, without looking for instances as these rules cannot lead to infinite abstract state-derivations alone. Also, we try to evaluate terms which cannot lead to infinite abstract state-derivations completely. Additionally, we continue to evaluate states as long as we did not perform enough evaluating steps. The corresponding parameter of the heuristic is used to ensure that we gather enough knowledge due to evaluations before we close the graph. The more knowledge we gather the better are our chances to synthesize DT problems which are easy to analyze in the next section. On the other hand we might continue the evaluation (and need computation time) without gathering additional knowledge. In fact, we might even obtain DT problems which are more difficult to analyze due to their growing size. Thus, the parameter for the required number of evaluation steps has to be chosen heuristically.

Example 6.9. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{q}(X) \leftarrow \mathsf{p}(\mathsf{s}(\mathsf{s}(0)), X), \mathsf{r}(X). \tag{65}$$

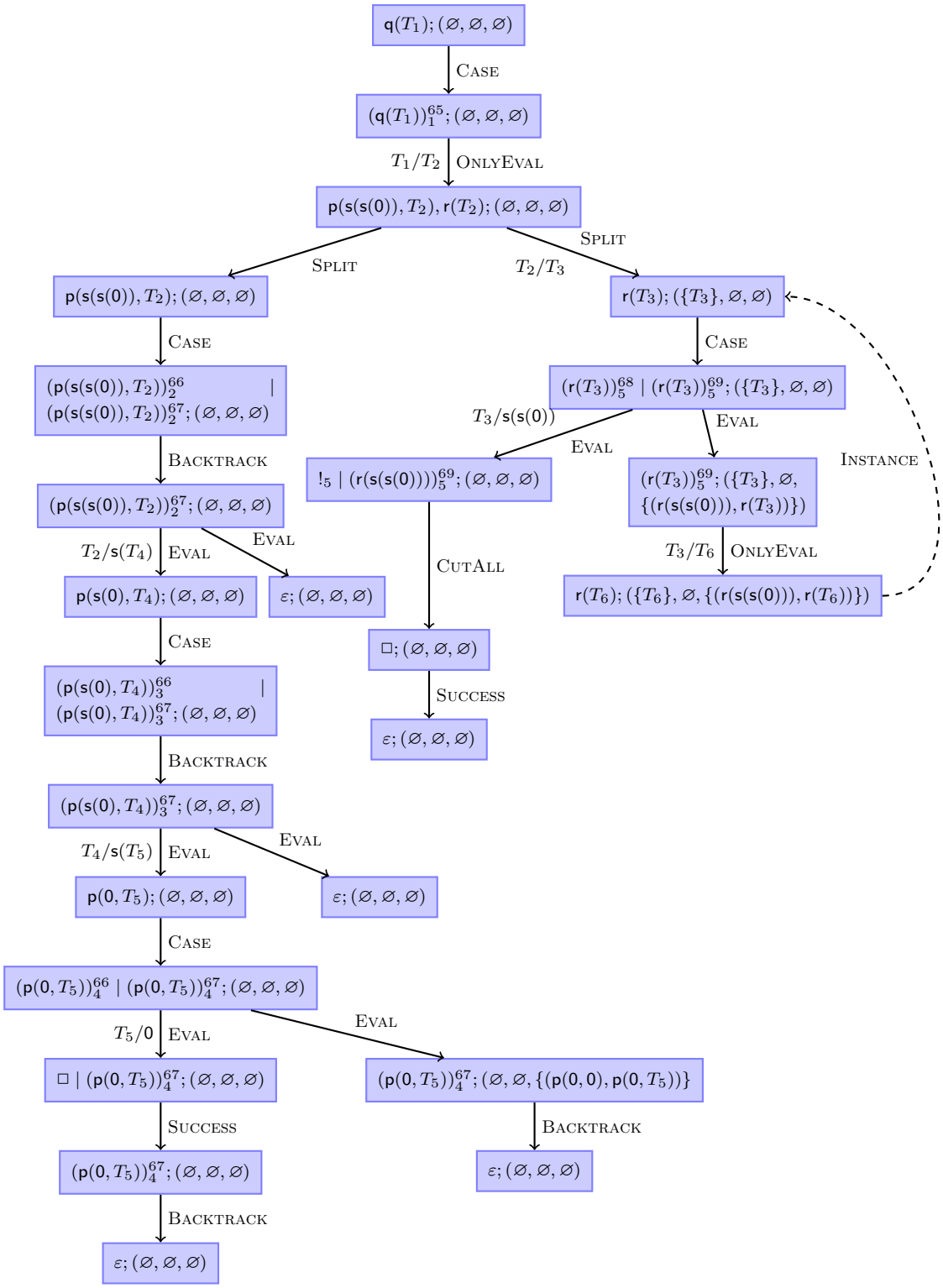
$$\mathsf{p}(0,0) \leftarrow \square. \tag{66}$$

$$\mathsf{p}(\mathsf{s}(X), \mathsf{s}(Y)) \leftarrow \mathsf{p}(X, Y). \tag{67}$$

$$\mathsf{r}(\mathsf{s}(\mathsf{s}(0))) \leftarrow !. \tag{68}$$

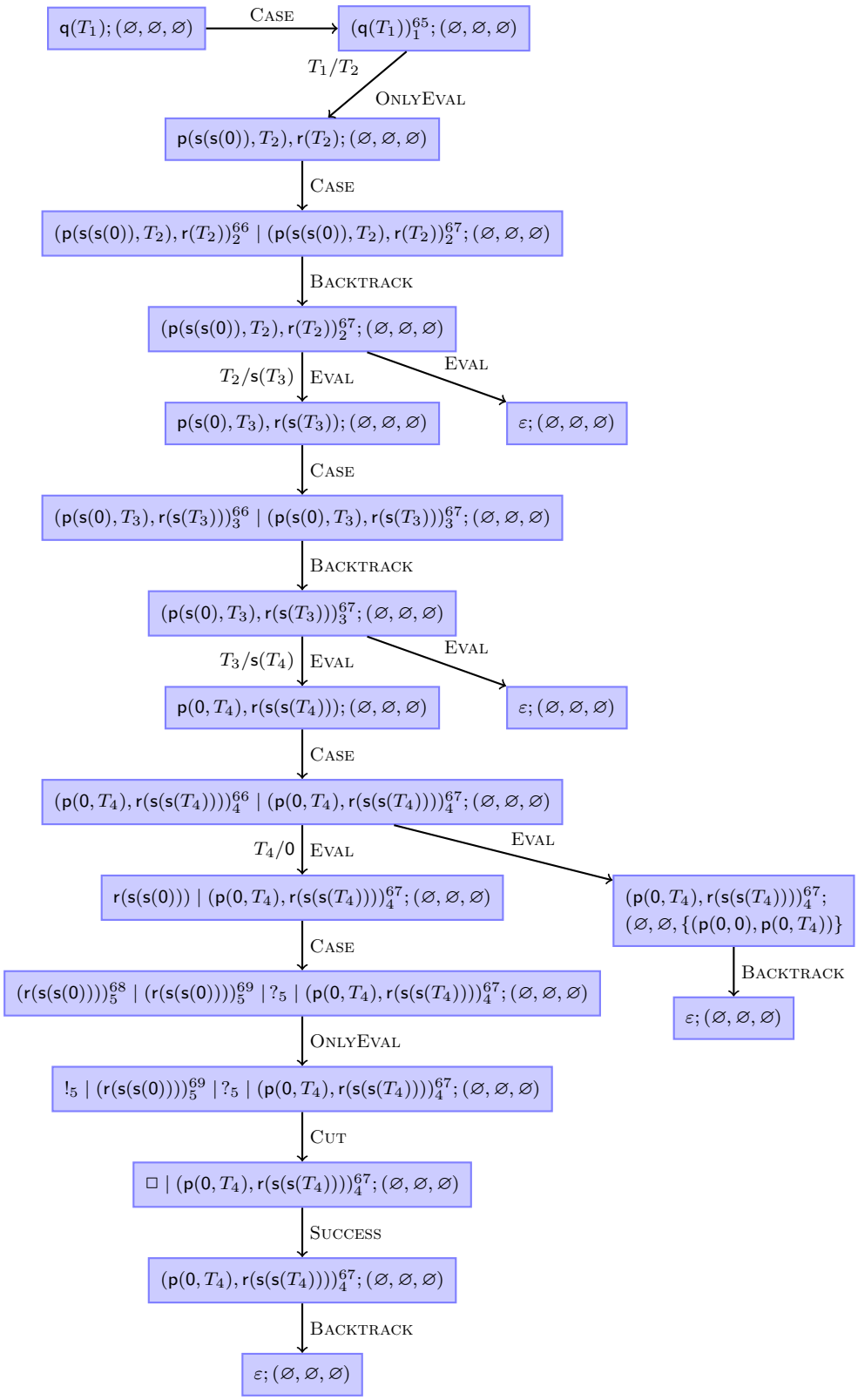
$$\mathsf{r}(X) \leftarrow \mathsf{r}(X). \tag{69}$$

and the query set $\mathcal{Q} = \{\mathsf{q}(t) \mid \mathsf{q}(t) \in \mathit{PrologTerms}(\Sigma, \mathcal{N})\}$. This program is terminating w.r.t. $\mathcal{Q}$. Now consider the following graph we would build if we try to find instances for recursive predicates as soon as possible.



The cycle in this graph can be traversed infinitely often since the information that the second argument of p is instantiated with a ground term is not sufficient to ensure unification with the first clause for r.

If we demand to evaluate at least four times using Case before trying to find instances, we obtain the following acyclic graph because we keep the shape information about the second argument of p.



If we already have applied enough evaluating rules, no shortening rule is applicable and we have a recursive term, we try to find an instance father for the current state starting with this term and, therefore, to close the graph in order to obtain a finite analysis. To this end, we need to use the INSTANCE rule instead of expanding the graph infinitely often. But since the INSTANCE rule may lose precision (or even violate the conditions for a termination graph), we try to delay its application as long as possible to gather more knowledge about the evaluation of the states in our graph. In particular, we can exclude some states from the set of possible candidates for the application of the INSTANCE rule. On the one hand, we have to exclude states which would lead to a cycle of only INSTANCE and GENERALIZATION edges. On the other hand, we may also exclude states for which we will safely find better candidates.

First, we demand that an instance candidate is no INSTANCE node itself and, thus, an instance child. Second, for every path from the instance candidate to the respective node, we must have applied an abstract inference rule other than INSTANCE and GENERALIZATION. This last condition ensures that we do not build cycles only consisting of INSTANCE and GENERALIZATION edges. In addition to that there is no reason to choose a state as an instance candidate which is an instance child itself, since we can choose its instance father instead.

The last condition is a heuristical one. If we draw an INSTANCE edge to a state with more different variables than the state under consideration, then we will unnecessarily lose information about the structure of the already evaluated terms in our states, since we have to instantiate different variables with terms containing no or some equal variables. If we delay the instantiation until we find an instance candidate with less or an equal number of different variables, such information can be preserved. If this condition is already true for the complete set of variables, we check it again for the abstract variables only. In particular, the way we will transform the termination graphs into DT problems relies on the different variables occurring in the states used for instantiation. Therefore, it is a good idea to keep their number as small as possible (cf. Chapter 7).

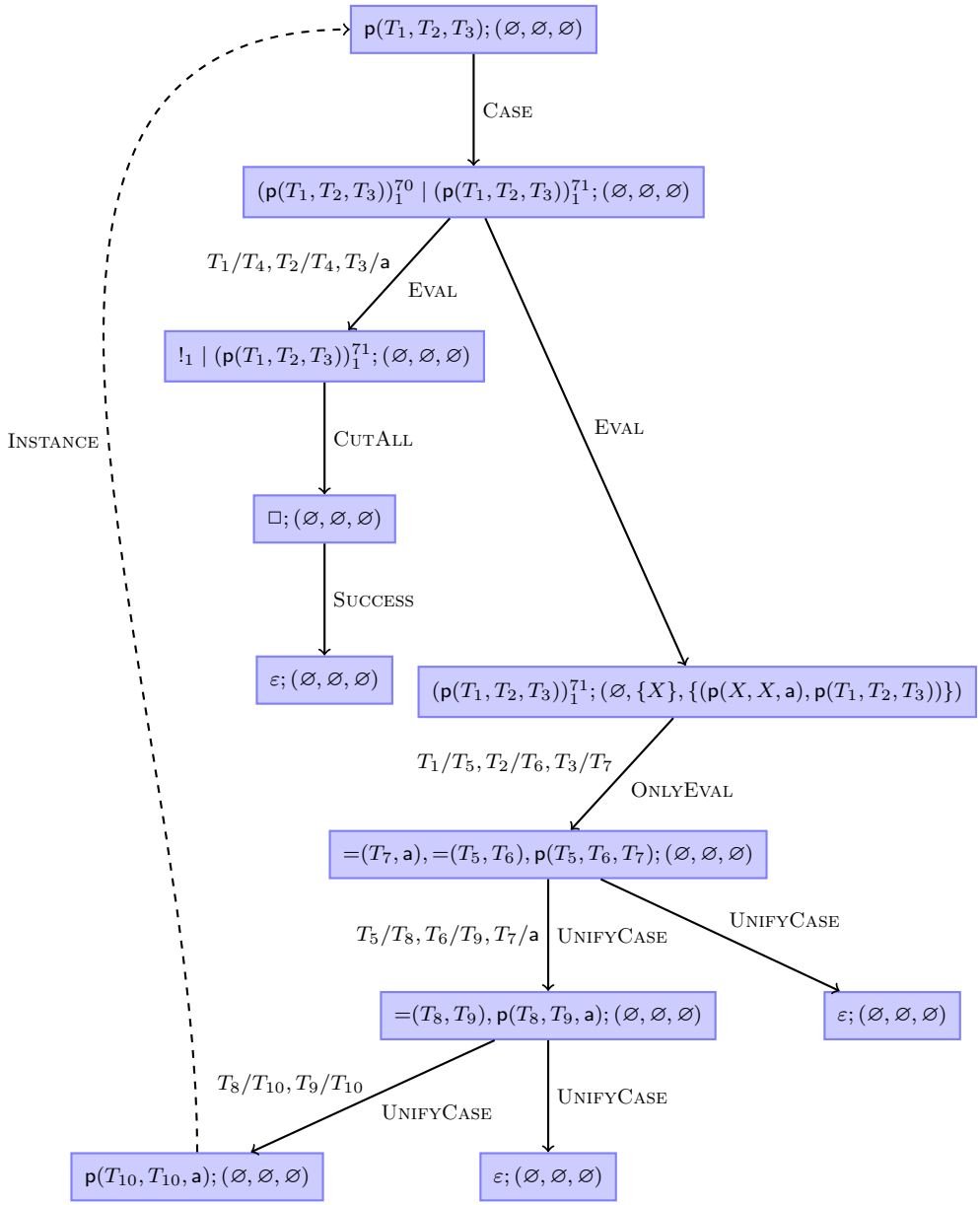
Example 6.10. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{p}(X, X, \mathsf{a}) \leftarrow !. \tag{70}$$

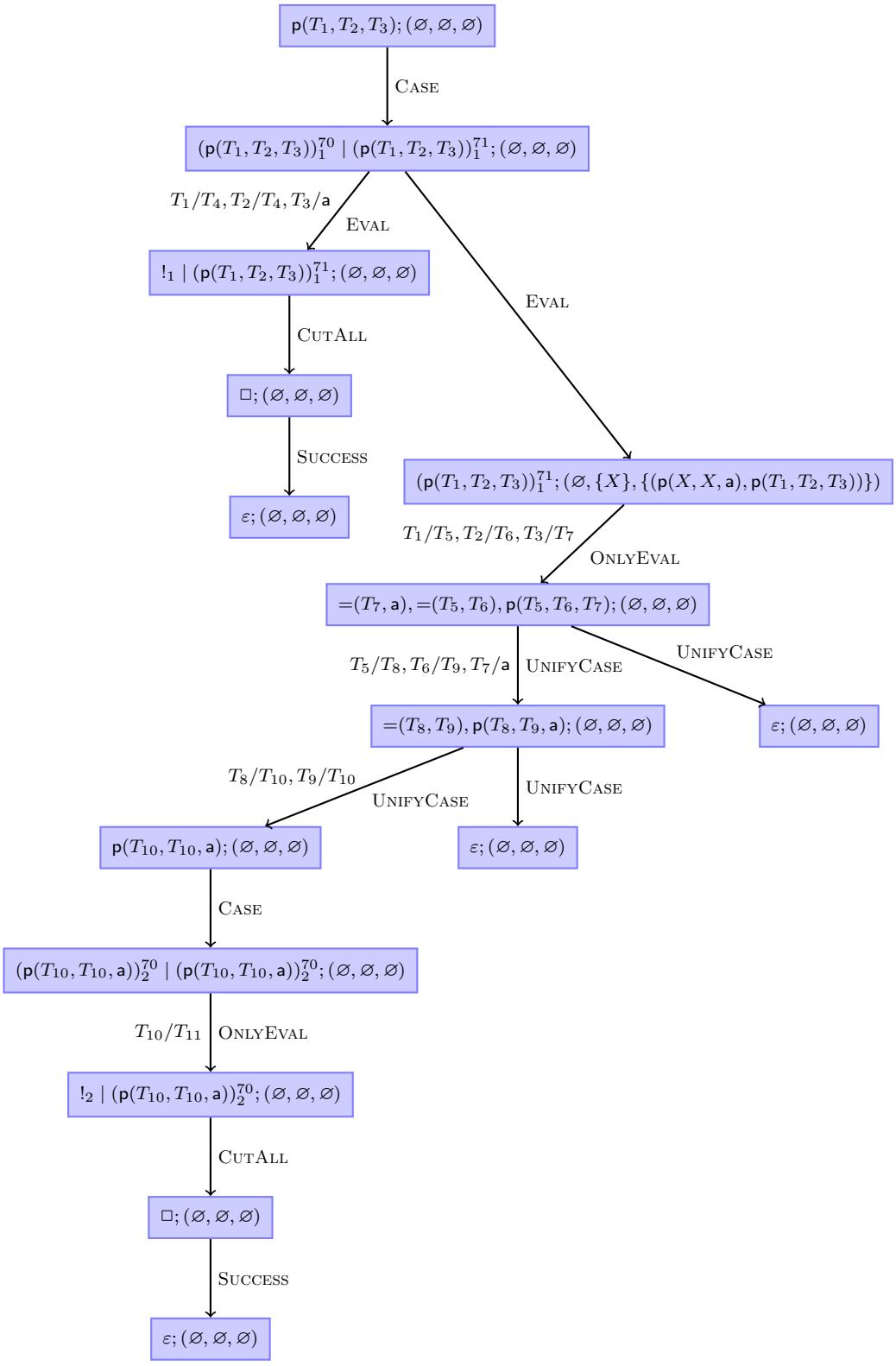
$$\mathsf{p}(X, Y, Z) \leftarrow =(Z, \mathsf{a}), =(X, Y), \mathsf{p}(X, Y, Z). \tag{71}$$

and the query set $\mathcal{Q} = \{\mathsf{p}(t_1, t_2, t_3) \mid \mathsf{p}(t_1, t_2, t_3) \in \mathit{PrologTerms}(\Sigma, \mathcal{N})\}$. $\mathcal{P}$ is terminating w.r.t. $\mathcal{Q}$.

Now consider the following two termination graphs where we try to use every matching state as an instance father.



While in fact the cycle in this graph cannot be traversed infinitely often, we will read off a DT problem for this graph which is not terminating. The reason is that the cycle contains a right successor of an EVAL node which is not reachable anymore after one traversal of the cycle. Unfortunately, the DT problem cannot recognize this. Anyway, this cycle is not necessary in this graph. By closing the graph too early we lose important information. Here, we match a state with three different variables to a state containing only one variable. Thus, we lose the information that the first two arguments of $\mathcal{P}$ are equal and that the third argument is in fact a and not an arbitrary term. By preventing to close the graph in such situations due to the last condition for instance candidates, we obtain a more precise analysis. This can be seen in the following acyclic termination graph which we obtain if we adhere to this condition.



However, me make an exception for those terms having a recursive root symbol whose branching factor is too big as the graph may grow exponentially with the number of applicable rules for the terms. The parameter determining the allowed value for the branching factor is again chosen heuristically as we would lose too much precision for too small values while obtaining infeasible big graphs for too big values.

Example 6.11. Consider the Prolog program soundex.pl from the TPDB

$$
\begin{array}{rcl}
\mathsf{goal}(\mathit{Name}, \mathit{Code}) & \leftarrow & \mathsf{eq}(\mathit{Name}, .(\mathit{First}, \mathit{Others})), \\
 & & \mathsf{reduce}(\mathit{Others}, \mathsf{s}(0), \mathit{First}, \mathit{Reduced}), \\
 & & \mathsf{eq}(\mathit{Code}, \mathit{Reduced}). \\
\mathsf{reduce}(X, \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))), Z, []) & \leftarrow & !. \\
\mathsf{reduce}([], Y, Z, []) & \leftarrow & !. \\
\mathsf{reduce}(.(\mathit{Current}, \mathit{Others}), & & \\
\quad \mathit{Count}, & & \\
\quad \mathit{Current}, & & \\
\quad \mathit{Code}) & \leftarrow & \mathsf{vowel\_h\_w\_y}(\mathit{Current}), \\
 & & !, \\
 & & \mathsf{reduce}(\mathit{Others}, \mathit{Count}, \mathit{Current}, \mathit{Code}). \\
\mathsf{reduce}(.(\mathit{Letter}, \mathit{Others}), & & \\
\quad \mathit{Count}, & & \\
\quad \mathit{Letter}, & & \\
\quad \mathit{Code}) & \leftarrow & !, \\
 & & \mathsf{reduce}(\mathit{Others}, \mathit{Count}, \mathit{Letter}, \mathit{Code}). \\
\mathsf{reduce}(.(\mathit{Current}, \mathit{Others}), & & \\
\quad \mathit{Count}, & & \\
\quad Z, & & \\
.(\mathit{Current}, \mathit{Code})) & \leftarrow & \mathsf{reduce}(\mathit{Others}, \mathsf{s}(\mathit{Count}), \mathit{Current}, \mathit{Code}). \\
\mathsf{vowel\_h\_w\_y}(97) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(101) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(105) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(111) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(117) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(104) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(119) & \leftarrow & \square. \\
\mathsf{vowel\_h\_w\_y}(121) & \leftarrow & \square. \\
\mathsf{eq}(X, X) & \leftarrow & \square.
\end{array}
$$

and the set of queries $\mathcal{Q} = \{\mathsf{goal}(t_1, t_2) \mid t_1 \text{ is ground}\}$. For a maximal branching factor of 5 or higher, we are not able to construct a termination graph for this program within 60 seconds with our fully automated termination prover AProVE running on a 2.67 GHz

Intel Core i7. However, for a maximal branching factor of 4 or less we can construct a termination graph and prove termination of this program w.r.t. $\mathcal{Q}$ in about 3.9 seconds.

If no instance father can be found, we next try to generalize or simplify the state by splitting it with Parallel or Split until we find an instance father for it or we have reduced the state to contain only a single goal with a single term. In the latter case we continue to evaluate the state, but as we will see in the proofs following the definition of the heuristic, this cannot happen infinitely often.

For the simplification with Parallel, we can detect cases where we can safely simplify the current state without losing precision. This is the case if the first state element of the current state does not contain any active cuts. Then it is impossible to get rid of the following state elements during the evaluation of the first and we have to analyze them anyway. Thus, we can split off the first state element using Parallel. This safe simplification can be performed even before we check for instances.

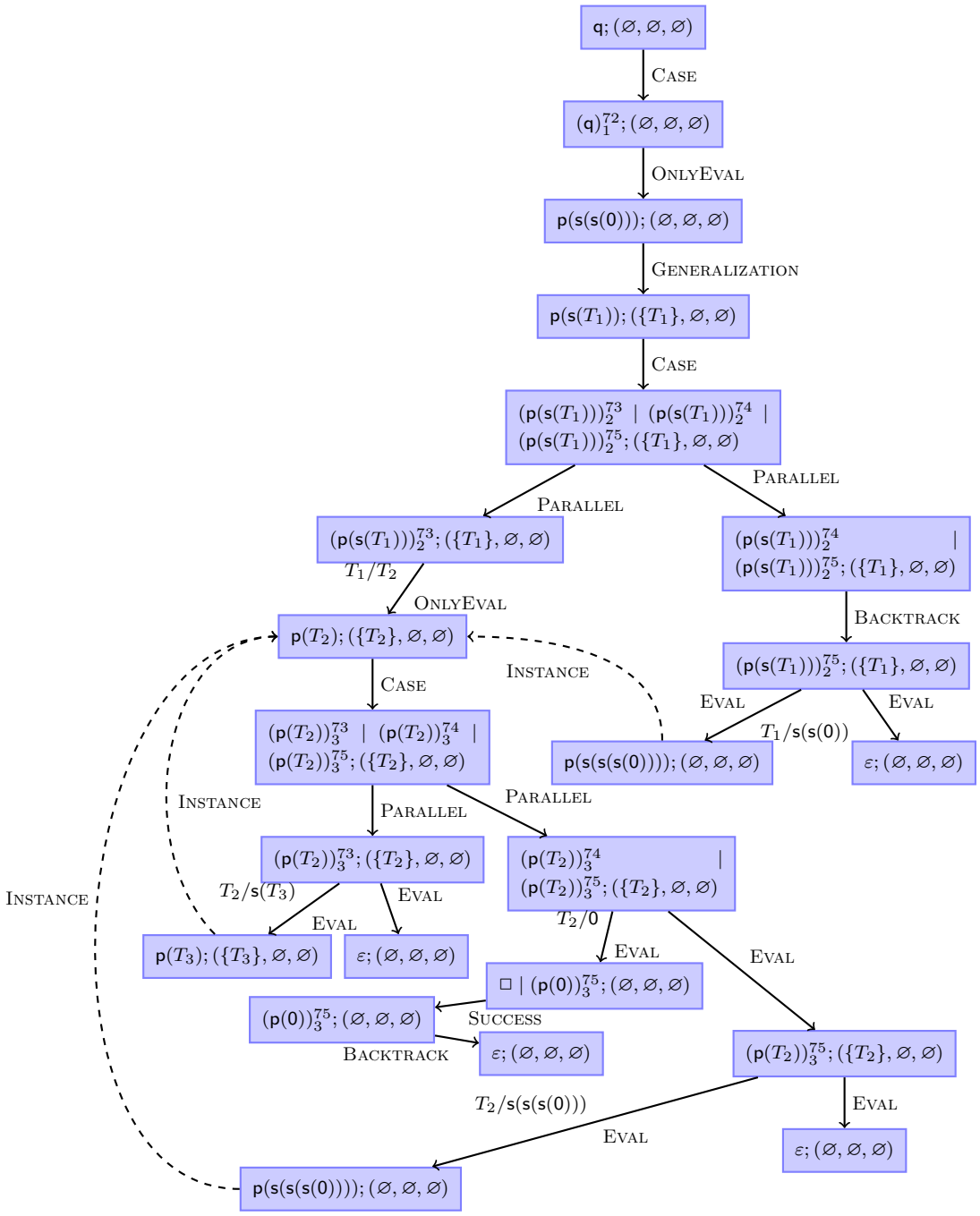
The generalization is controlled by four parameters. While Generalization virtually always significantly loses precision, it is needed to limit the number of different terms modulo scope variants and variable renaming we might encounter during evaluation. Then we can guarantee to find instances for any state after finitely many steps. For finite terms, we can limit the number of different terms modulo scope variants and variable renaming by the nested depth of function symbols occurring in the terms. Then the length of the finite paths in the terms is limited by the number of different function symbols in Σ. However, for infinite rational terms this idea is not as useful as the nested depth of some function symbol is always infinity. We would therefore have to generalize all infinite terms. Instead, we exploit the property of rational terms to have only a finite number of different subterms. Hence, we can limit the number of different infinite rational terms modulo scope variants and variable renaming by limiting the number of different subterms. While this would also be possible for finite terms, limiting the nested depth instead of the number of different subterms allows for more different terms in case of finite terms and we have to use Generalization less often. Still, the controlling parameters have to be chosen heuristically as too high values lead to infeasible big graphs.

Example 6.12. Consider again the termination graph from Example 3.42. There we have generalized terms with a nested depth of function symbols greater than one. Even for such a small generalization depth and short program the graph is comparatively big. If we raise the allowed nested depth of function symbols by one, the resulting termination graph grows by a complete subgraph consisting of seven nodes. Termination graphs for more complex programs may even grow exponentially in the maximal nested depth of function symbols.

Consider on the other hand the following Prolog program $\mathcal{P}$

$$\mathsf{q} \leftarrow \mathsf{p}(\mathsf{s}(\mathsf{s}(0))). \tag{72}$$
$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{73}$$
$$\mathsf{p}(0) \leftarrow \square. \tag{74}$$
$$\mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) \leftarrow \mathsf{p}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))). \tag{75}$$

and the query set $\mathcal{Q}$ consisting of the single query q. $\mathcal{P}$ is obviously terminating w.r.t. $\mathcal{Q}$. However, if we use a maximal nested depth of one here, we obtain the following termination graph.



The cycles in this termination graph do not terminate. Due to the early generalization we lose too much information about the shape of p's argument.

Additionally, we have to determine at which position we want to generalize the terms satisfying the conditions for generalization. While generalizing the terms at a position close to the root allows to find instances quickly, this loses more precision than generalizing at deeper positions as we keep more knowledge about the shape of the generalized terms. Thus, we have a trade-off between speed and precision again.

We make another exception for the simplification of states by Parallel and Split if we detect that a cut must be reached by further evaluating the state. Then we just evaluate it until we reach the cut as this will simplify the state without losing precision. To have a description for states where we definitely reach cuts, we introduce the notion of *cuttable* goals and states.

Definition 6.13 (Cuttable Goals and States)**.** *Let $\mathcal{P}$ be a Prolog program and KB a knowledge base. We call an unlabeled goal $t_1, \ldots, t_n$* cuttable *w.r.t. $\mathcal{P}$ and KB iff there is an index $i \in \{1, \ldots, n\}$ such that $t_i = !_m$ for some scope $m \in \mathbb{N}$ and for all $j \in \{1, \ldots, i-1\}$ we have that ($root(t_j)$ is not recursive w.r.t. $\mathcal{P}$ or $root(t_j) = \mathsf{repeat}/0$ or $root(t_j)$ is no built-in predicate and we have $Slice(\mathcal{P}, t_j) = \{H_{c_1} \leftarrow B_{c_1}, \ldots, H_{c_k} \leftarrow B_{c_k}\}$ with $k > 0$ such that there is an index i' where the abstract* OnlyEval *rule is applicable to the state $(t_j)_m^{c_{i'}}; KB$ while the abstract* Backtrack *rule is applicable to the state $(t_j)_m^{c_{j'}}; KB$ for all $j' \in \{1, \ldots, i'-1\}$ and $B_{c_{i'}} = \Box$).*

We call a labeled goal $(t, Q)_m^c$ cuttable *w.r.t. $\mathcal{P}$ and KB iff the abstract* OnlyEval *rule is applicable to the state $(t, Q)_m^c; KB$ and the goal $B'_c\sigma', Q\sigma'$ is cuttable w.r.t. $\mathcal{P}$ and $KB\sigma|_{\mathcal{G}}$ where B'_c, σ' and $\sigma|_{\mathcal{G}}$ are defined as in the abstract* OnlyEval *rule for the clause $H_c \leftarrow B_c$.*

We call a state $S; KB'$ cuttable *w.r.t. $\mathcal{P}$ iff the first state element of S is cuttable w.r.t. $\mathcal{P}$ and KB'.*

Example 6.14. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{q} \leftarrow \mathsf{p}(\mathsf{s}(0)), !. \tag{76}$$
$$\mathsf{p}(0) \leftarrow \mathsf{r}. \tag{77}$$
$$\mathsf{p}(X) \leftarrow \Box. \tag{78}$$
$$\mathsf{r} \leftarrow \mathsf{r}. \tag{79}$$

and the knowledge base $KB = (\varnothing, \varnothing, \{(\mathsf{p}(0), \mathsf{p}(T_1))\})$.

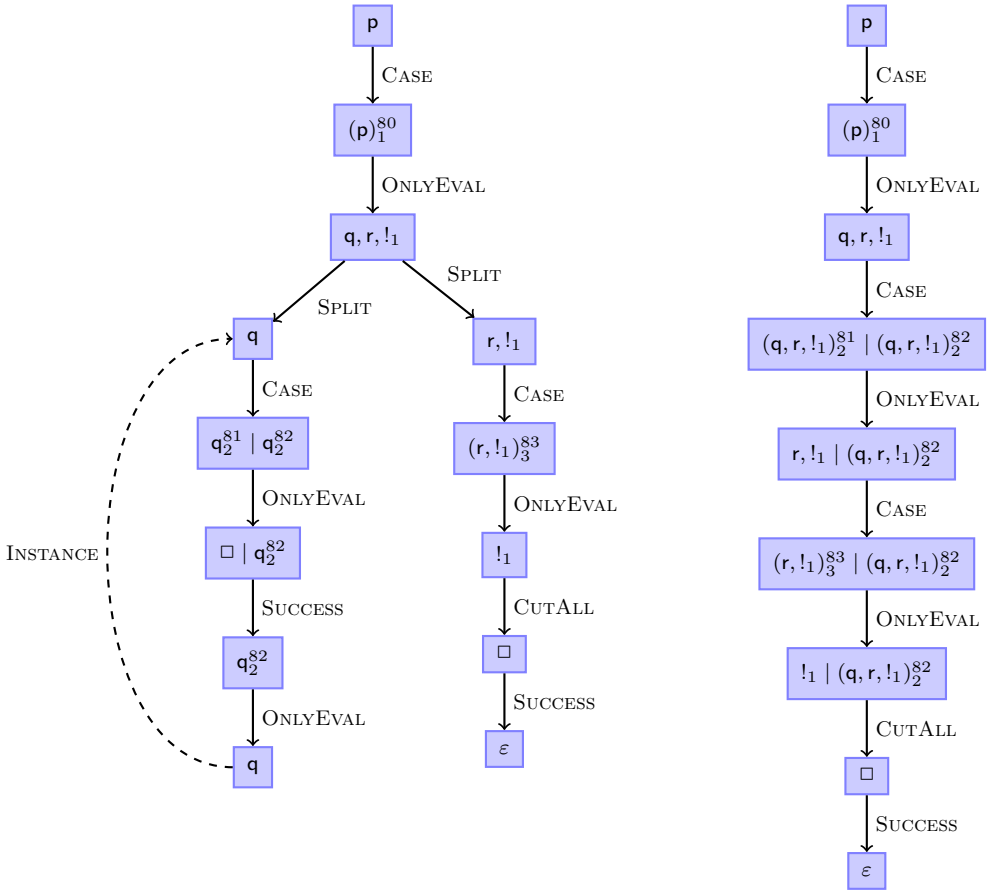
The goals $(\mathsf{p}(T_1), !_1)$ and $(\mathsf{q})_1^{76}$ are cuttable w.r.t. $\mathcal{P}$ and KB while the goals $(\mathsf{p}(0), !_1)$, $(\mathsf{p}(T_2), !_1)$, $(\mathsf{p}(\mathsf{s}(0)), \mathsf{p}(T_1))$ and $(\mathsf{p}(\mathsf{s}(0)), \mathsf{q})$ are not cuttable w.r.t. $\mathcal{P}$ and KB. Note however that the last two goals would not cause any problems if we would evaluate them instead of trying to simplify them.

The notion of cuttable goals is a heuristical one. The last goal $(\mathsf{p}(\mathsf{s}(0)), \mathsf{q})$ is not cuttable w.r.t. $\mathcal{P}$ and KB in spite of the fact that we would definitely reach a cut. We leave a more sophisticated notion of cuttable goals to future work.

Example 6.15. Consider the following Prolog program ts09.pl which is terminating for the query p:

$$\mathsf{p} \leftarrow \mathsf{q}, \mathsf{r}, !. \tag{80}$$
$$\mathsf{q} \leftarrow \square. \tag{81}$$
$$\mathsf{q} \leftarrow \mathsf{q}. \tag{82}$$
$$\mathsf{r} \leftarrow \square. \tag{83}$$

If we use the SPLIT rule after the first evaluation of p, we obtain the first of the following termination graphs, where we omit the knowledge bases as we do not have any variables in this Prolog program:



As we can see, the cycle in this termination graph does not terminate since we cannot use the cut for q after splitting this first recursive predicate from the remaining goal.

By detecting that we will reach the cut because the first defining clause for q is not recursive, we obtain the second acyclic termination graph instead.

Now, the standard heuristic is defined as follows.

Definition 6.16 (Standard Heuristic for Termination Graph Conctruction)**.** *Given a partial termination graph $G = (V, E)$ and a Prolog program $\mathcal{P}$, we perform the following algorithm for each leaf n in V which is not empty until we either abort the heuristic or G is a termination graph, i.e., all leaves are empty. This algorithm is called* standard heuristic*. The parameters for the standard heuristic are:*

- MinExSteps $\in \mathbb{N}$
- MaxBranchingFactor $\in \mathbb{N}$
- FiniteGeneralizationDepth $\in \mathbb{N} \setminus \{0, 1\}$
- InfiniteGeneralizationDepth $\in \mathbb{N} \setminus \{0, 1\}$
- FiniteGeneralizationPosition $\in \{1, \ldots,$ FiniteGeneralizationDepth$\}$
- InfiniteGeneralizationPosition $\in \mathbb{N} \setminus \{0\}$

In the following algorithm, we demand that the algorithm ends for the node n and proceeds from the beginning with the next non-empty leaf of G whenever a rule is applied.

(i) Try to apply a rule from ShorteningRules to n

(ii) If there is a path π from $root(G)$ to n with no nodes from $Instance(G)$ and at least MinExSteps *nodes from $Case(G) \cup Call(G) \cup Repeat(G)$ and ($head(n)$ is recursive w.r.t. $\mathcal{P}$ or n is starting with a labeled goal whose corresponding clause is recursive w.r.t. $\mathcal{P}$) and n is not cuttable w.r.t. $\mathcal{P}$:*

(a) If $n = S_1 \mid \ldots \mid S_k; KB$ with $k > 1$, $\forall i \in \{1, \ldots, k\} : S_i \in StateElements$ and $AC(S_1) = \varnothing$: Apply the Parallel *rule to obtain the states $S_1; KB$ and $S_2 \mid \ldots \mid S_k; KB$*

(b) If n is starting with an unlabeled goal: Try to apply the Instance *rule to the current node n and a node out of $InstanceCandidates(n, G, \mathcal{P})$*

(c) If there is a finite term t in n with $k \geq$ FiniteGeneralizationDepth *positions $pos_1 \lhd pos_2 \lhd \ldots \lhd pos_k \in Occ(t)$ with $\forall i \in \{1, \ldots, k\} : root(t|_{pos_i}) = f$ for a single $f \in \Sigma$ or there is an infinite term t' in n with $k' \geq$* InfiniteGeneralizationDepth *different subterms: Apply a generalization step to n*

(d) If $n = S_1 \mid \cdots \mid S_k; KB$ with $k > 1$, $\forall i \in \{1, \ldots, k\} : S_i \in StateElements$: Apply a parallel step to n

(e) If $n = S; KB = t_1, \ldots, t_k; KB$ with $k > 1$: Apply the Split *rule to n*

(iii) Try to apply a rule from EvaluationRules to n

(iv) Abort the heuristic

The set EvaluationRules contains exactly the rules Call, Case, Conjunction, Disjunction, Eval, IfThen, IfThenElse, Not, Once, OnlyEval *and* Repeat *while the set ShorteningRules is defined as AbstractInferenceRules* $\setminus$ (*EvaluationRules* $\cup$ {Generalization, Instance, Parallel, Split}).

For a node n or a state S we denote by $head(n) = f$ and $head(S) = f$ respectively that n and S are starting with an unlabeled goal as their first state element where this goal has a first term t with $root(t) = f$.

The set of instance candidates *(denoted $InstanceCandidates(n, G, \mathcal{P})$) for n w.r.t. G and $\mathcal{P}$ contains all nodes $n_c \in V$ for which the following conditions hold:*

- $n_c \notin Instance(G)$
- *for all paths π from n_c to n we have $\exists n' \in \pi : n' \in Rule(G)$ where $Rule \in AbstractInferenceRules \setminus$* {Instance, Generalization}
- $(|\mathcal{V}(n)| \geq |\mathcal{V}(n_c)|$ *and* $|\mathcal{V}(n)| = |\mathcal{V}(n_c)| \implies |\mathcal{A}(n)| \geq |\mathcal{A}(n_c)|)$ *or* ($head(n)$ *is recursive w.r.t. $\mathcal{P}$ and $branchingFactor(head(n), \mathcal{P}) >$* MaxBranchingFactor)

The generalization step for a node $n = S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$ works as follows:

(i) Create a new node $n' = S'; (\mathcal{G}', \mathcal{F}', \mathcal{U}')$ where $S' = S, \mathcal{G}' = \mathcal{G}, \mathcal{F}' = \mathcal{F}$ and $\mathcal{U}' = \mathcal{U}$

(ii) Initialize the set $R = \varnothing$

(iii) While there is an infinite term t in n' with $k \geq$ InfiniteGeneralizationDepth *different subterms do:*

(a) For all positions $pos \in Occ(t)$ with length InfiniteGeneralizationPosition *and $t|_{pos}$ is infinite do:*

i. If R contains an element $(t|_{pos}, T)$: Replace the term $t|_{pos}$ with T

ii. Else: Add the pair $(t|_{pos}, T)$ to R and replace the term $t|_{pos}$ with T for a fresh abstract variable T

(iv) While there is a finite term t in n' with $k \geq$ FiniteGeneralizationDepth *positions $pos_1 \lhd \ldots \lhd pos_k \in Occ(t)$ with $\exists f \in \Sigma : \forall i \in \{1, \ldots, k\} : root(t|_{pos_i}) = f$ do:*

(a) If R contains an element $(t|_{pos_{\text{FiniteGeneralizationPosition}}}, T)$: Replace the term $t|_{pos_{\text{FiniteGeneralizationPosition}}}$ with T

(b) Else if $\mathcal{V}(t|_{pos_{\text{FiniteGeneralizationPosition}}}) \subseteq \mathcal{G}$: Add the pair $(t_{pos_{\text{FiniteGeneralizationPosition}}}, T)$ to R, replace the term $t_{pos_{\text{FiniteGeneralizationPosition}}}$ with T and add T to $\mathcal{G}'$ for a fresh abstract variable T

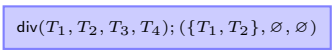
(c) Else: Add the pair $(t_{pos_{\text{FiniteGeneralizationPosition}}}, T)$ to R and replace the term $t_{pos_{\text{FiniteGeneralizationPosition}}}$ with T for a fresh abstract variable T

(v) Apply the GENERALIZATION *rule to n and n'*

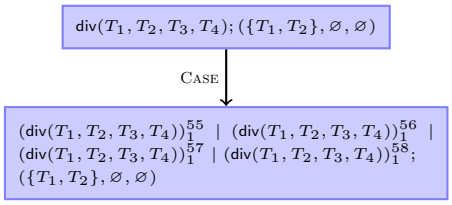
The parallel step for a node $n = S_1 \mid \cdots \mid S_k; KB$ with $k > 1$ and $\forall i \in \{1, \ldots, k\} : S_i \in$ StateElements is defined by applying the PARALLEL *rule to n in such a way that we reach the states $S_1 \mid \cdots \mid S_j$ and $S_{j+1} \mid \cdots \mid S_k$ with the smallest $j \in \{1, \ldots, k-1\}$ for which the application of* PARALLEL *is possible.*

Example 6.17. In fact, the termination graph from Example 6.3 was constructed using this heuristic. To see this, we follow some steps of the heuristic during the construction of this termination graph. We refer to the program as $\mathcal{P}$. As parameters, we choose $\mathit{MinExSteps} = 1$, $\mathit{MaxBranchingFactor} = 3$, $\mathit{FiniteGeneralizationDepth} = 7$ and $\mathit{FiniteGeneralizationPosition} = 2$. The remaining parameters have no effect on this example as no infinite terms occur there. Thus, they can be arbitrarily chosen. We choose the next unprocessed node according to the depth-first, left-to-right order w.r.t. the current partial termination graph.

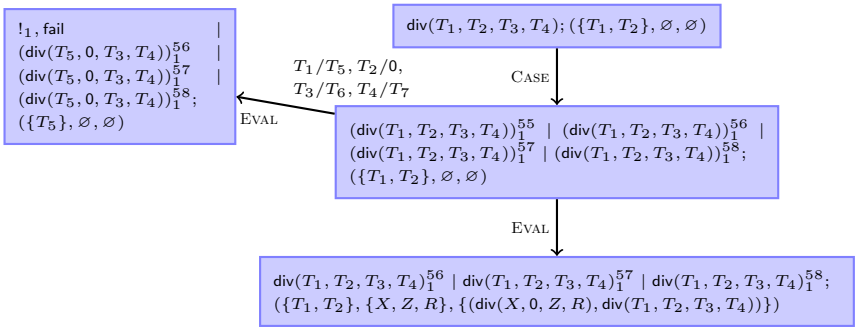
We start with the initial state as the first partial termination graph.



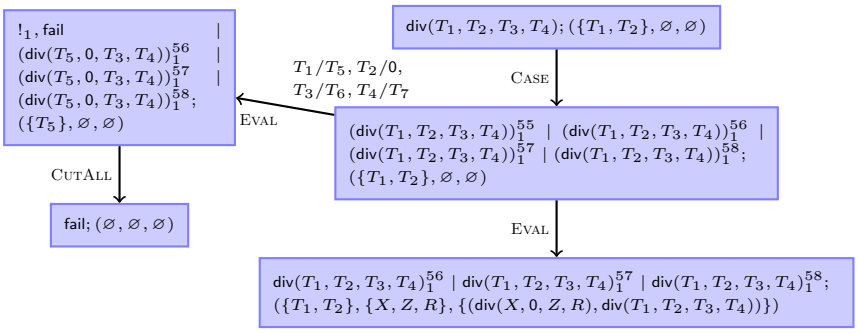
Since we cannot apply any rules from instruction (i), we only have one state element and we have not yet applied one evaluating rule, we proceed with instruction (iii) and apply the CASE rule.



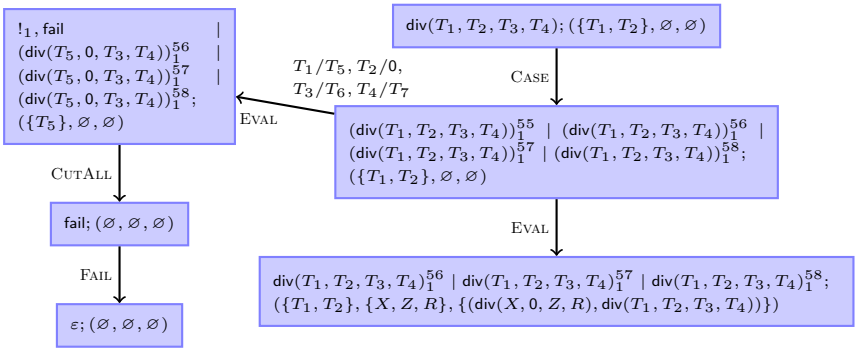
Again, no rule from instruction (i) is applicable. The test in instruction $(ii - -a)$ fails as the first state element has an active cut and the test in instruction (ii) fails as this state is cuttable w.r.t. $\mathcal{P}$. Thus, we apply the EVAL rule in instruction (iii).



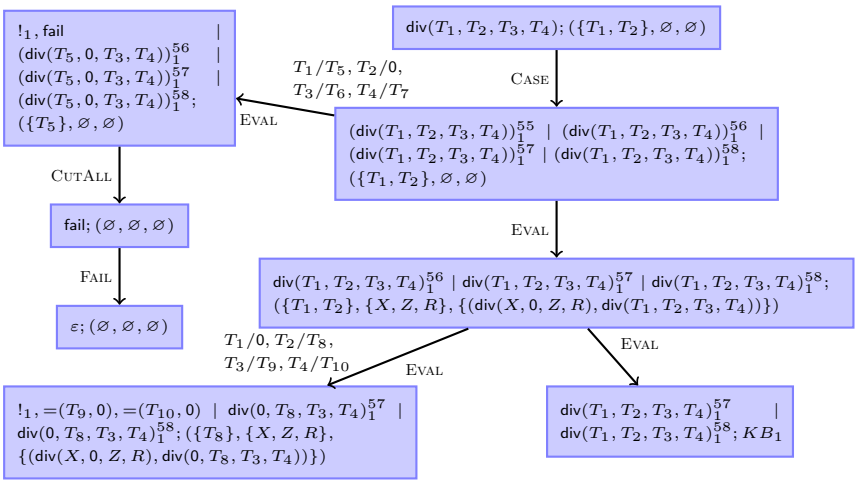
For the next current node, we apply the CUTALL rule in instruction (i).



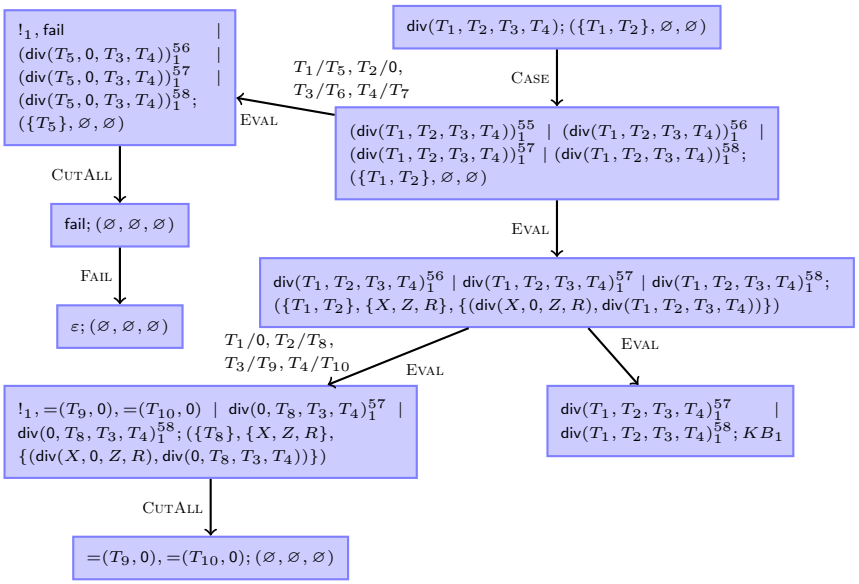
For this node we apply the Fail rule in instruction (i).



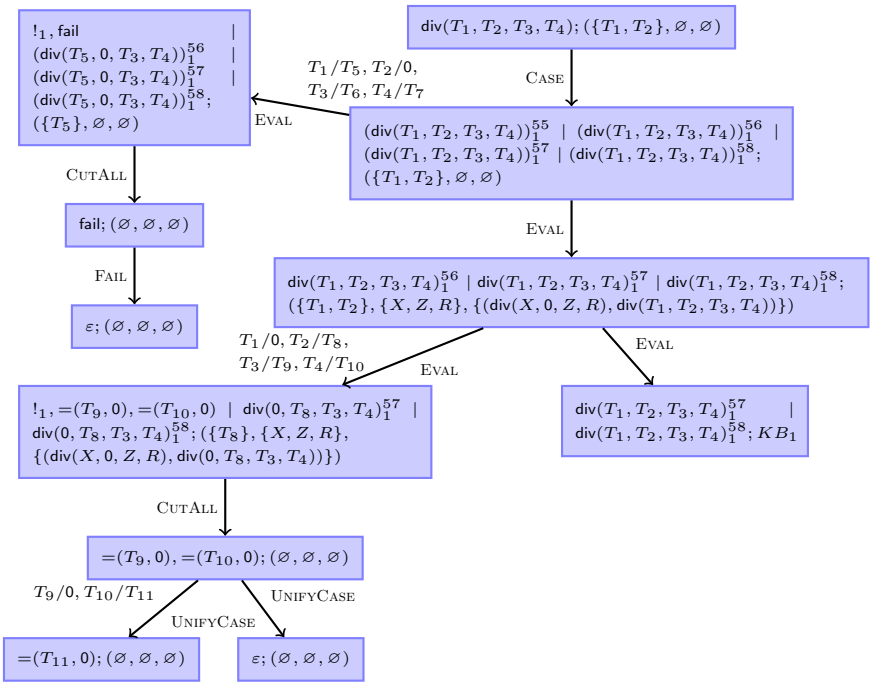
Now this state is empty and we proceed with the next unprocessed state. Again, this state is cuttable and we apply the Eval rule.



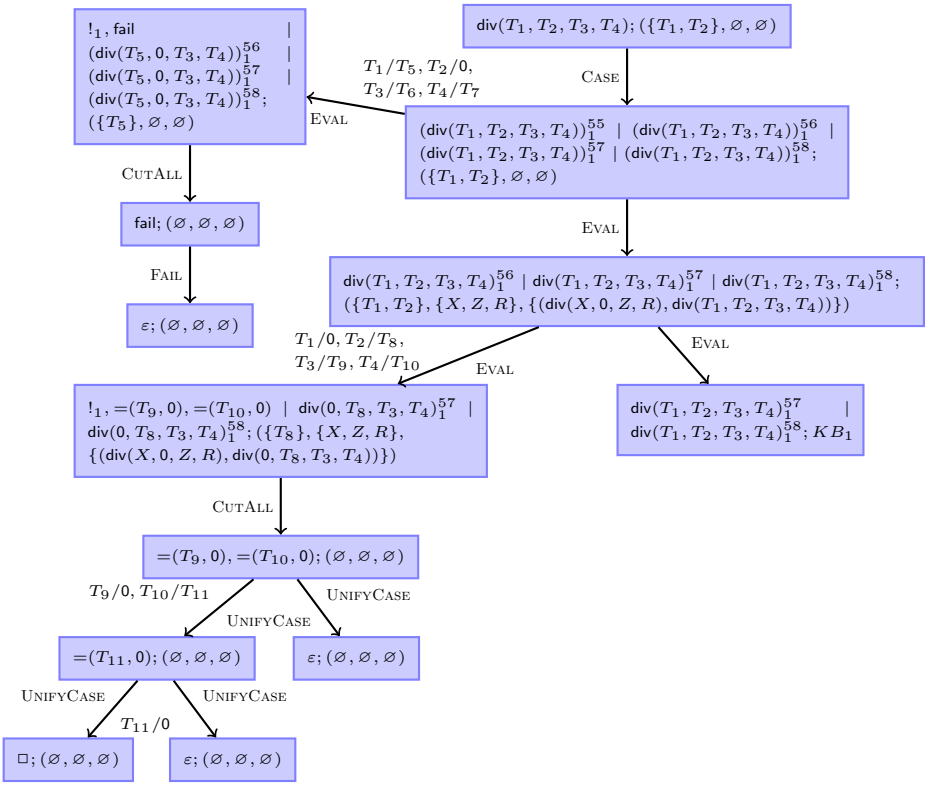
We once more apply the CutAll rule.



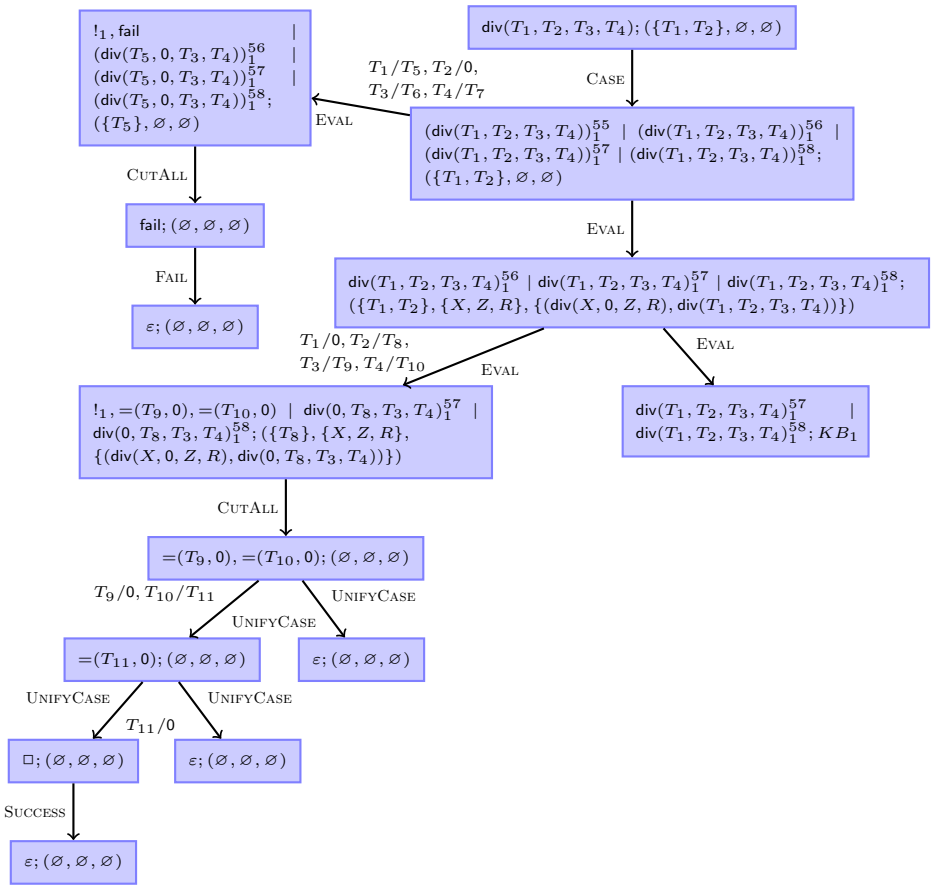
Now we apply the UnifyCase rule.



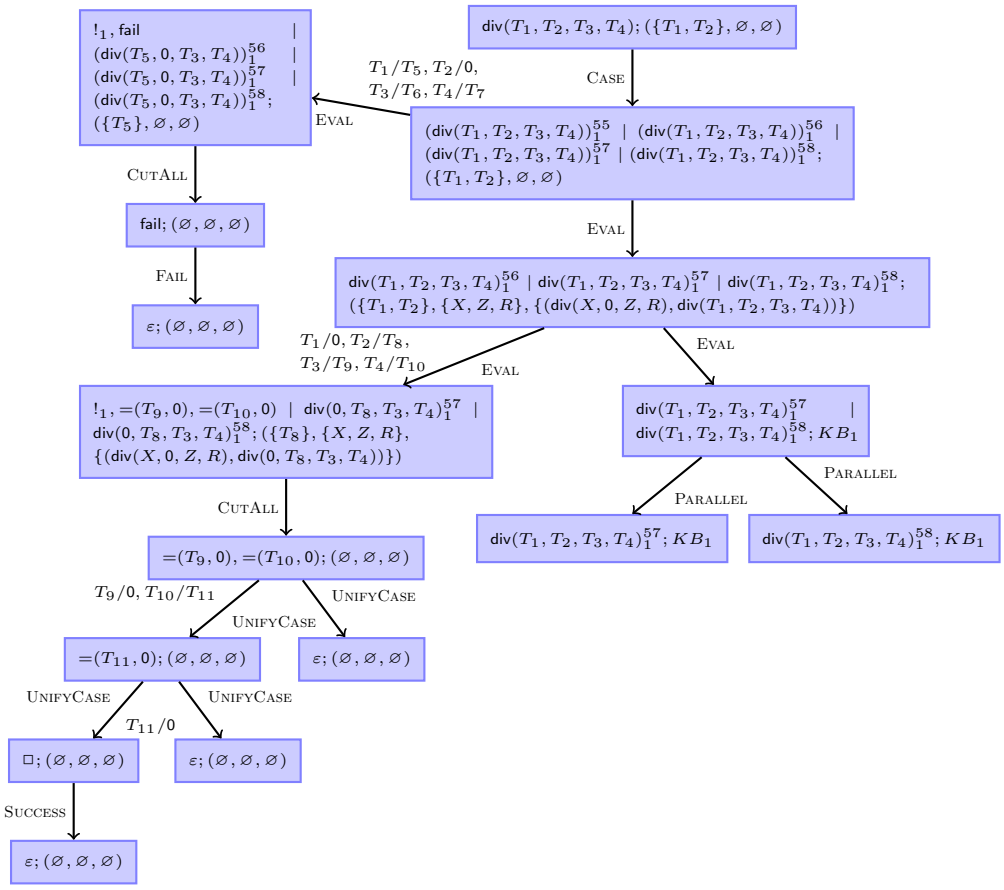
We apply the UnifyCase rule once more.



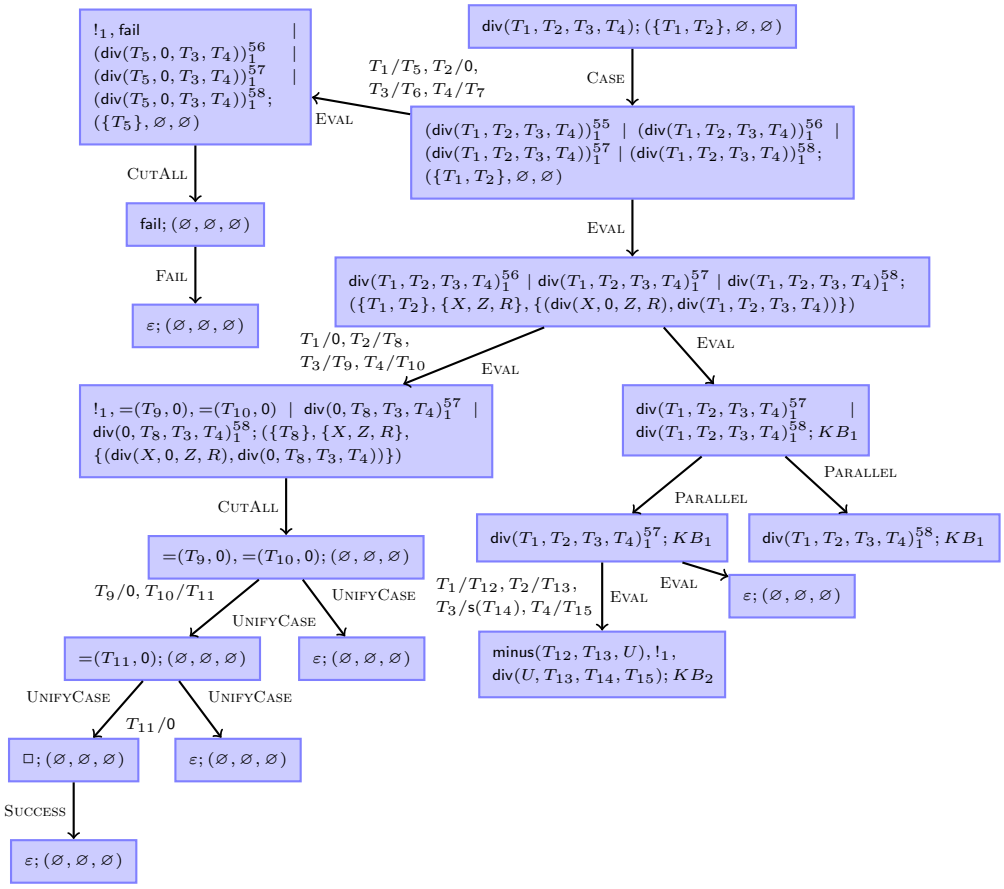
Finally, we apply the Success rule to obtain another empty state.



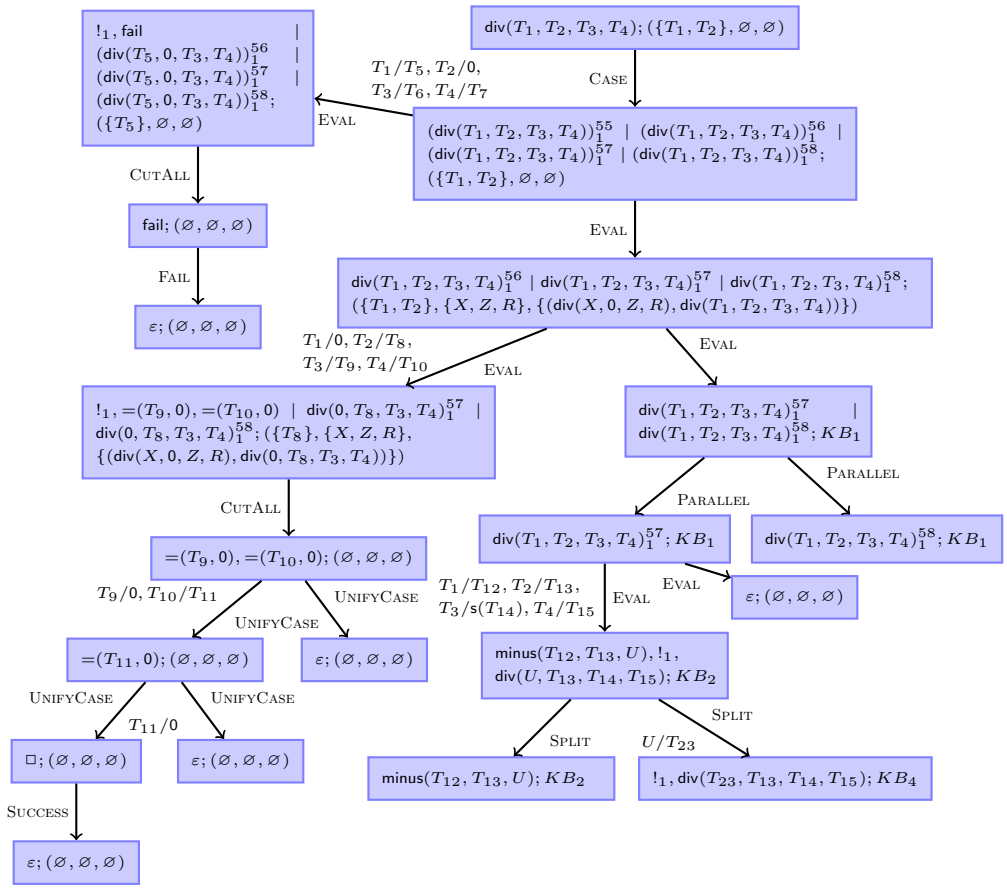
The next unprocessed state is not cuttable and, thus, we perform a parallel step.



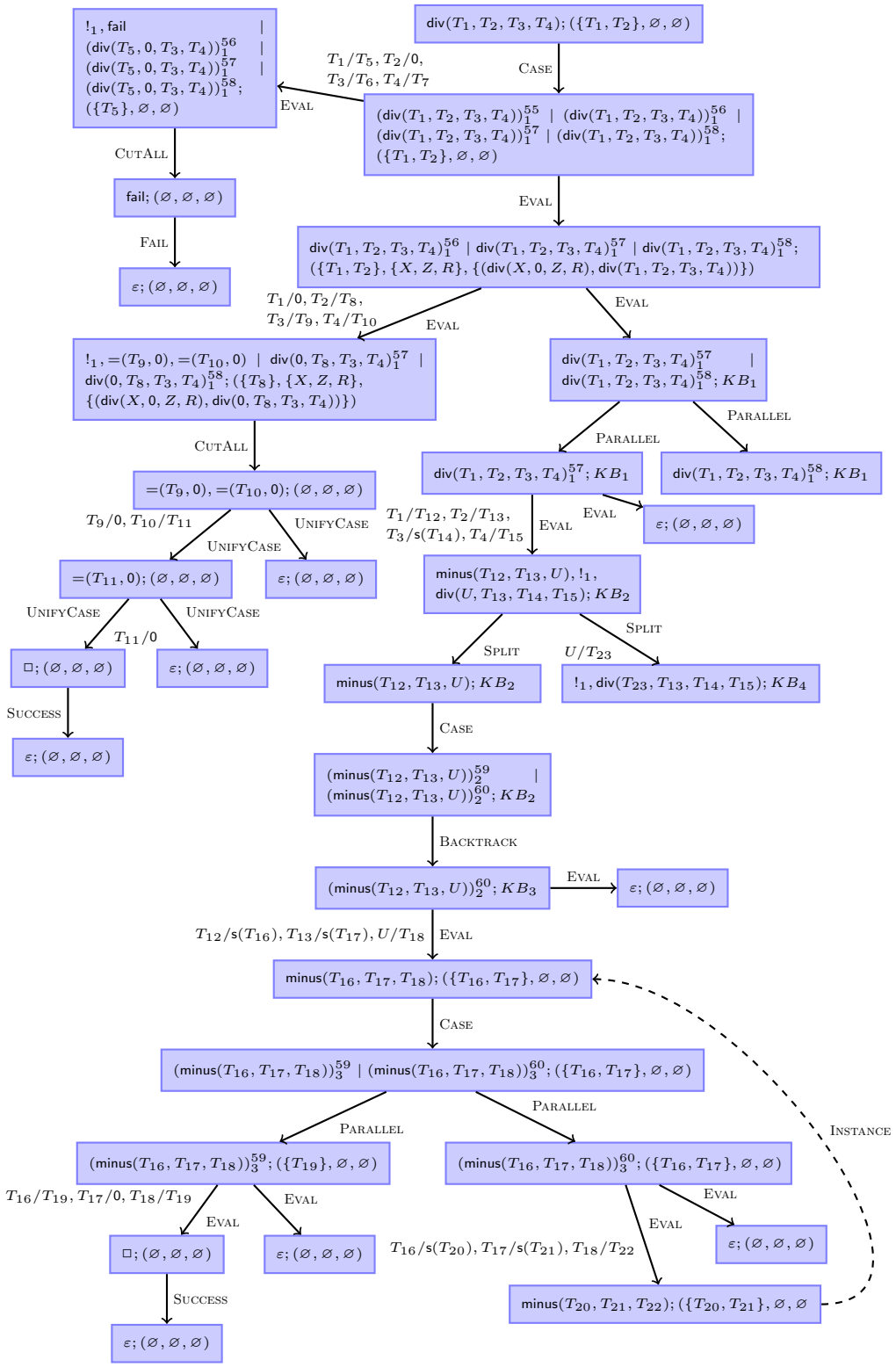
As the current state has only one state element, we apply the EVAL rule.



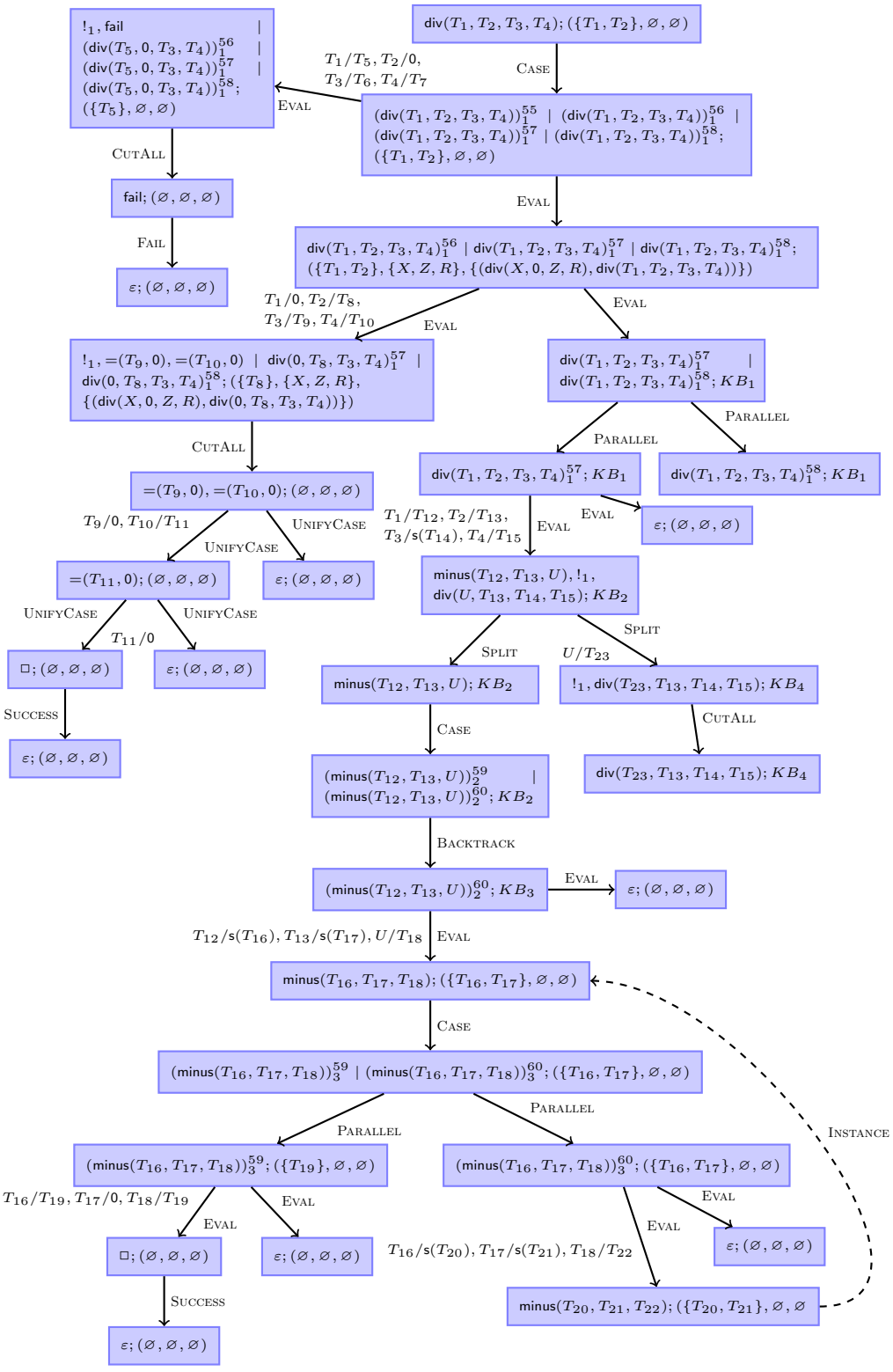
Still, the current state is not cuttable. The reason is that the first clause for minus does not unify with the first term in the current goal. Hence, we apply the SPLIT rule. The groundness analysis based on argument filtering used in AProVE finds out that the variable U will be instantiated by a ground term. As we only instantiate a free variable, we do not have to replace the remaining non-ground variables in the states by fresh abstract variables.



We continue in this manner for the evaluation of the state $\mathsf{minus}(T_{12}, T_{13}, U); KB_2$. However, we now skip the further abstract state-derivation for this left successor state of the SPLIT rule to see how we find an instance father for its right successor state $!_1, \mathsf{div}(T_{23}, T_{13}, T_{14}, T_{15}); KB_4$.



First, we have to apply the CutAll rule in instruction (i).



Now, due to the groundness analysis we know that the first two arguments of div are ground terms. Also, we have an equal number of different abstract variables in the current and the initial state. Thus, we can close the graph here for the recursive predicate div and, after further evaluation of the remaining Parallel branch, obtain the termination graph from Example 6.3.

6.3 Proving Termination and Correctness of the Standard Heuristic

To prove termination of this heuristic, we first need to look at the possibilities of applying abstract inference rules to abstract states, as the heuristic applies one rule in each step (unless it is aborted). To describe the successive application of abstract inference rules we introduce the notion of a *rule chain*.

Definition 6.18 (Rule Chain)**.** *A* rule chain *w.r.t. a* Prolog *program $\mathcal{P}$ is a (possibly infinite) sequence of abstract inference rules R_i and abstract states S_i of the form $S_0, R_1, S_1, R_2, S_2, \ldots$ such that we reach the state S_i from the state S_{i-1} by applying the inference rule R_i w.r.t. $\mathcal{P}$. Infinite rule chains may not contain the* Instance *rule, while finite rule chains may only contain the* Instance *rule as the last rule of the sequence.*

The first step in proving termination of the heuristic is to see that we would need to apply the Call, Case, Generalization or Repeat rule infinitely often to obtain an infinite rule chain.

Lemma 6.19 (Infinite Rule Chains Contain Infinitely Many *CCGR* Rules)**.** *Every infinite rule chain contains infinitely many rules from the set CCGR* $= \{$Call, Case, Generalization, Repeat$\}$.

Proof. We consider the possible applications of other abstract inference rules than the ones in *CCGR* before applying one of the rules from this set.

- After application of Instance we finish the rule chain and, thus, it cannot be infinite.
- As Call is not defined for terms having an infinite path with function symbols from *GoalJunctors* only, it is not possible to have such a term as a predication. Since the rules Conjunction, Disjunction, IfThen and IfThenElse, reduce the number of symbols from *GoalJunctors* in a state, they cannot be applied infinitely often before applying the Call rule which is from *CCGR*.
- Parallel can only be applied finitely often. The reason for this is that Parallel is only applicable to states with more than one state element and both children of Parallel have less state elements than their parent. The only rules where at least some children have more state elements than their parent not being from *CCGR* are Disjunction, IfThen, IfThenElse and Not where the first three rules are only finitely often applicable and introduce only a finite number of new state elements. For Not we must have a state of the form $\backslash+(t'), Q \mid S$. After application of Not we obtain a state $\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S$. Thus, the number of state elements

added by the NOT rule is always two, but as we cannot apply the NOT rule to a state starting with the first or third state element, the number of state elements cannot be raised infinitely often by the application of the NOT rule.

- EVAL and ONLYEVAL are only finitely often applicable, because they are only applicable to states starting with a labeled goal and reach states starting with an unlabeled goal or (in the case of EVAL) without the first state element of their parent respectively. So the number of labeled goals is reduced by one for every child of such a rule. The only rule introducing new labeled goals is CASE $\in$ *CCGR*.

- CUT and CUTALL can only be applied finitely often, since their application reduces the number of cuts in the state at least by one and the only rules not being from *CCGR* and introducing new cuts to their children are EVAL and ONLYEVAL which can only be applied finitely often and introduce a finite number of cuts.

- As SPLIT can only be applied to an unlabeled goal with more than one term in the list of terms and no further backtracking possibilities, it can only be applied finitely often, since it reduces the number of terms in the list of terms at least by one and the only rules not from *CCGR* which can introduce new unlabeled goals or add terms to existing ones are DISJUNCTION, EVAL and ONLYEVAL which can only be applied finitely often and introduce only one or two unlabeled goals per application.

- HALT, HALT1, THROW, VARIABLEERROR and UNDEFINEDERROR lead to the empty state for which no rule is applicable anymore. Thus, they cannot be part of an infinite rule chain.

- ATOMICFAIL, BACKTRACK, COMPOUNDFAIL, EQUALSFAIL, FAIL, FAILURE, NONVARFAIL, NOUNIFYFAIL, SUCCESS, UNEQUALSFAIL UNIFYFAIL and VARFAIL drop one state element and can, therefore, only be applied finitely often with the analogous argument as for PARALLEL.

- ATOMICSUCCESS, COMPOUNDSUCCESS, EQUALSSUCCESS, NEWLINE, NONVARSUCCESS, NOUNIFYSUCCESS, TRUE, UNEQUALSSUCCESS, UNIFYSUCCESS, VARSUCCESS, WRITE, WRITECANONICAL and WRITEQ reduce the number of terms in the first state element. The only rules introducing new terms and not being from *CCGR* are EVAL and ONLYEVAL which can only be applied finitely often.

- ATOMICCASE, COMPOUNDCASE, EQUALSCASE, NONVARCASE, NOUNIFYCASE, UNEQUALSCASE, UNIFYCASE and VARCASE have two successor states where one term in the first state element or one state element is reduced. Hence, they cannot be applied infinitely often before applying a rule from *CCGR* by the identical arguments as for the two cases before.

- For NOT we must have a state of the form $\backslash+(t'), Q \mid S$. After application of NOT we obtain a state $\mathsf{call}(t'), !_m, \mathsf{fail} \mid Q \mid ?_m \mid S$. Since we cannot apply the NOT rule to any term of the first state element or to the third state element of this resulting state and Q and S are not modified, we can apply the CUT rule only finitely often as we reduce the number of terms where NOT is applicable.

- For ONCE we must have a state of the form $\mathsf{once}(t'), Q \mid S$ and we reach the state $\mathsf{call}(,(t', !)), Q \mid S$ after application of ONCE. Thus, the number of terms where ONCE is applicable is reduced after every application of ONCE and we can apply ONCE only finitely often.

□

The next observation is that there are only finitely many different terms by which a state may start such that we can apply the CALL, CASE or REPEAT rule to it and do not apply the GENERALIZATION rule first.

Lemma 6.20 (Finite Set of Terms for CALL, CASE and REPEAT without Preceding Generalization)**.** *The set* $CCRTerms = \{t \in PrologTerms(\Sigma, \mathcal{V}) \mid t$ *is finite and there are at most* $k <$ FiniteGeneralizationDepth *positions* $pos_1 \lhd pos_2 \lhd \ldots \lhd pos_k \in Occ(t)$ *with* $\forall i \in \{1, \ldots, k\} : root(t|_{pos_i}) = f$ *for a single* $f \in \Sigma$ *or* t *is infinite and has at most* InfiniteGeneralizationDepth *different subterms*$\}$ *is finite modulo scope variants and separate variable renaming on* $\mathcal{N}$ *and* $\mathcal{A}$ *for all* FiniteGeneralizationDepth*,* InfiniteGeneralizationDepth $\in \mathbb{N} \setminus \{0, 1\}$.

Proof. Remember that a finite term is a finite tree. We consider a path from the root to a leaf in such a tree. By definition of *CCRTerms* such a path may not contain a certain function symbol more than FiniteGeneralizationDepth $- 1$ times. Thus, the length of the path is limited by the finite product of $(|\Sigma| - 1)$ and (FiniteGeneralizationDepth $- 1$) (since the infinitely many labeled cuts and variables do not have any arguments). As all function symbols in Σ have a finite arity the number of finite terms in *CCRTerms* is finite modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$.

As the infinite terms in *CCRTerms* may at most contain InfiniteGeneralizationDepth-1 different subterms and each subterm has a root symbol from Σ or is a labeled cut or is a variable from $\mathcal{N}$ or $\mathcal{A}$, their number modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$ is limited by $(|\Sigma|+3\cdot(\text{InfiniteGeneralizationDepth}-1))^{\text{InfiniteGeneralizationDepth}-1}$ and, therefore, finite. □

Finally, we show that the evaluation of non-recursive function symbols cannot lead to infinite rule chains alone.

Lemma 6.21 (Non-Recursive Symbols Finite Abstract State-Derivations)**.** *Let $S = t; KB$ be a state where $root(t)$ is not recursive w.r.t. a given Prolog program $\mathcal{P}$. If we do not use* Instance *or* Generalization*, all abstract state-derivations from $S; KB$ lead to a finite acyclic termination graph.*

Proof. We perform the proof by induction over the lexicographic combination of first the number k of function symbols used as root symbols for clause heads in $\mathcal{P}$, second the number of meta-predications t' w.r.t. t where the rules IfThen, IfThenElse, Not or Once are applicable to the state $t'; KB'$ for some knowledge base KB', third the number of predications t'' w.r.t. t where the rules Call, Conjunction or Disjunction are applicable to the state $t''; KB''$ for some knowledge base KB''. This relation is well-founded as t may not have infinitely many meta-predications.

First, we consider the case $root(t) \in \mathit{BuiltInPredicates}$. As $root(t)$ is not recursive, we know that $root(t) \neq \mathsf{repeat}/0$. All successor states of abstract inference rules applicable to S except for the rules Call, Conjunction, Disjunction, IfThen, IfThenElse, Not and Once lead to the empty state ε for S as they drop at least the first term. Thus, the lemma trivially holds in such cases.

We perform a case analysis over the remaining rules for built-in predicates.

- Call drops one $\mathsf{call}/1$ symbol from the root of t leading to the goal $t' \mid ?_m$ having one meta-predication less than t where Call is applicable and an equal number of meta-predications where the other remaining rules are applicable. Thus, we obtain a finite termination graph G for $t' \mid ?_m$ by the induction hypothesis. By inserting the abstract state-derivation from t to $t' \mid ?_m$ at the root of G, we obtain a finite termination graph for t.

- Conjunction drops one $,/2$ symbol from the root of t leading to the goal t_1, t_2 having one meta-predication less than t where Conjunction is applicable while leaving the number of meta-predications for the other rules unchanged. By the induction hypothesis we obtain a finite termination graph G for the state t_1, t_2. By inserting the abstract state-derivation from t to t_1, t_2 at the root of G, we obtain a finite termination graph for t.

- Disjunction drops one $;/2$ symbol from the root of t leading to the state $t_1 \mid t_2$ having one meta-predication less than t where Disjunction is applicable while leaving the number of meta-predications for the other rules unchanged. By the induction hypothesis we obtain a finite termination graph G for the state $t_1 \mid t_2$. By inserting the abstract state-derivation from t to $t_1 \mid t_2$ at the root of G, we obtain a finite termination graph for t.

- IfThen drops one $->/2$ symbol from the root of t leading to the state $\mathsf{call}(t_1), !_m, t_2 \mid ?_m$ for some scope m having one meta-predication less than t where IfThen is appli-

cable while leaving the number of meta-predications for the rules IfThenElse, Not and Once unchanged. By the induction hypothesis we obtain a finite termination graph G for the state $\mathsf{call}(t_1), !_m, t_2 \mid ?_m$. By inserting the abstract state-derivation from t to $\mathsf{call}(t_1), !_m, t_2 \mid ?_m$ at the root of G, we obtain a finite termination graph for t.

- IfThenElse drops one ;/2 symbol from the root of t and one ->/2 symbol from t's first argument leading to the state $\mathsf{call}(t_1), !_m, t_2 \mid t_3 \mid ?_m$ for some scope m having one meta-predication less than t where IfThenElse and IfThen are applicable while leaving the number of meta-predications for the rules Not and Once unchanged. By the induction hypothesis we obtain a finite termination graph G for the state $\mathsf{call}(t_1), !_m, t_2 \mid t_3 \mid ?_m$. By inserting the abstract state-derivation from t to $\mathsf{call}(t_1), !_m, t_2 \mid ?_m$ at the root of G, we obtain a finite termination graph for t.

- Not drops one \+/1 symbol from the root of t leading to the state $\mathsf{call}(t'), !_m, \mathsf{fail} \mid \Box \mid ?_m$ for some scope m having one meta-predication less than t where Not is applicable while leaving the number of meta-predications for the rules IfThen, IfThenElse and Once unchanged. We obtain a finite termination graph G for $\mathsf{call}(t'), !_m, \mathsf{fail} \mid \Box \mid ?_m$ by the induction hypothesis. By inserting the abstract state-derivation from t to $\mathsf{call}(t'), !_m, \mathsf{fail} \mid \Box \mid ?_m$ at the root of G, we obtain a finite termination graph for t.

- Once drops one $\mathsf{once}/1$ symbol from the root of t leading to the state $\mathsf{call}(t', !)$ having one meta-predication less than t where Once is applicable while leaving the number of meta-predications for the rules IfThen, IfThenElse and Not unchanged. We obtain a finite termination graph G for $\mathsf{call}(t', !)$ by the induction hypothesis. By inserting the abstract state-derivation from t to $\mathsf{call}(t', !)$ at the root of G, we obtain a finite termination graph for t.

Now let $\mathit{root}(t) \notin \mathit{BuiltInPredicates}$. For $k = 0$ we have an empty program and, hence, the state t has only one abstract state-derivation with UndefinedError to ε. Thus, the lemma trivially holds.

For $k > 0$ we can assume the lemma holds for every program using $k' < k$ function symbols. Consider the set $\mathit{Slice}(\mathcal{P}, t)$. If it is empty, the only abstract state-derivation leads with UndefinedError to the empty state. Otherwise we have $\mathit{Slice}(\mathcal{P}, t) = \{H_1 \leftarrow B_1, \ldots, H_n \leftarrow B_n\}$. As $\mathit{root}(t)$ is not recursive, we know that B_i does not have any meta-predication t' with $(\mathit{root}(t') = \mathit{root}(t) \vee \mathit{root}(t') = \mathsf{repeat}/0)$ and not infinitely many meta-predications for all $i \in \{1, \ldots, n\}$. Also, we know that B_i may not use any clause leading to such a meta-predication in a sequence of clause applications. Thus, the further evaluation of B_i w.r.t. $\mathcal{P}$ is equivalent to an evaluation w.r.t. $\mathcal{P}'$ where $\mathcal{P}'$ contains

all clauses from $\mathcal{P}$ except for the clauses having $root(t)$ as the root symbol of their heads. Hence, we can use the induction hypothesis to obtain the claim of the lemma. □

Now we are ready to prove termination of the standard heuristic.

Lemma 6.22 (Termination of the Standard Heuristic)**.** *The standard heuristic from Definition 6.16 always terminates after finitely many steps.*

Proof. First, we see that every rule application is terminating as they produce only finitely many successor states and all tests and calculations used in the abstract inference rules are terminating. For the latter remember that rational terms have only finitely many different subterms such that it is sufficient to check for a loop of function symbols from *GoalJunctors* in the representation of rational terms to determine if t has only finitely many predication positions for a rational term t.

Second, we show that the algorithm performs only finitely many steps for each node. The first instruction consists of attempts to apply some abstract inference rules only. Hence, it clearly terminates. Instruction (*ii*) contains a test for the existence of a certain path in the partial termination graph, a test if a certain function symbol or clause is recursive, a test if enough rules from the set $\{\textsc{Call}, \textsc{Case}, \textsc{Repeat}\}$ have been applied so far and a test if the current state is cuttable. As every partial termination graph is finite, the test for the existence of a certain path clearly terminates. To determine if a function symbol or clause is recursive, one only has to detect loops in the finite Prolog program or rational terms having loops containing function symbols used for meta-programming. Consequently, this test also terminates. The comparison of a counter with a natural number obviously terminates. The check whether the current state is cuttable terminates as we only have to check finitely many terms and clauses. Now, if one of the tests fails, we continue with instruction (*iii*). Otherwise we first try to apply the Parallel rule if we have a state with more than one state elements where the first state element has no active cuts. This clearly terminates. Then we attempt to apply the Instance rule to an instance candidate for the current node if this node starts with an unlabeled goal. As the partial termination graph is finite and all conditions for instance candidates can be tested in finitely many steps, this instruction terminates, too. To see the latter, remember that we can determine the branching factor of a function symbol by a simple look-up in the finite program. Thus, its comparison with a natural number clearly terminates. If we did not apply the Instance rule, we perform a test for a generalization step. As we can perform this test for the finite representation of rational terms in finitely many steps, this test terminates as well. The generalization step also terminates as it consists of a node and set creation and replacements with fresh variables followed by a rule application. The replacements can be performed just like the test before and to find fresh variables it suffices to keep track of the finitely many variables used so far by using enumerated variables and using the next number to obtain a fresh variable. If we did not perform a generalization

step, we check if the current state contains more than one state element and possibly apply a parallel step. The check clearly terminates and the parallel step terminates as we have only finitely many partitions of the state elements. If we did not apply the parallel step, we test whether the current node consists of a single state element which is an unlabeled goal. Again, this test obviously terminates and the possibly following application of the SPLIT rule terminates as well. If we did not apply any abstract inference rule so far, we continue with instruction (*iii*) which only consists of further attempts to apply abstract inference rules. If we did not apply any rule up to this point, we abort the heuristic which trivially terminates.

Finally, we have to show that we do not create infinitely many new nodes while performing this algorithm. Assume we create infinitely many new nodes. To create new nodes, we have to apply some abstract inference rule to the current node. Thus, we have to construct an infinite rule chain χ as every rule only introduces a finite number of new nodes and we do not abort the heuristic. By Lemma 6.19 we know that χ contains infinitely many applications of rules from *CCGR*.

We continue by showing that we apply GENERALIZATION only finitely often. In order to apply GENERALIZATION we must have a current node n without any instance candidates for which INSTANCE is applicable and where $head(n)$ is recursive. After application of a generalization step, the resulting node does not satisfy the conditions of the test for the generalization step anymore. To see this, note that we replace every subterm satisfying the conditions of the test for a generalization step with a fresh abstract variable. By Lemma 6.20 we know that there are only finitely many different terms modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$ we might obtain by a generalization step. So after finitely many applications of the GENERALIZATION rule we obtain a state $S = t, Q \mid S_r; KB$ where there is another state $S' = t', Q' \mid S'_r; KB'$ in our graph and t and t' are equal modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$. Note that all terms in Q, S_r and KB are also generalized. If t, Q is cuttable w.r.t. $\mathcal{P}$ and KB, we will reach a cut in Q after finitely many steps and reduce the number of state elements in S or the number of terms in t, Q as all newly introduced state elements must be inserted before the corresponding question mark for the cut. If otherwise t, Q is not cuttable w.r.t. $\mathcal{P}$ and KB, we will separate t by repeatedly applying PARALLEL and SPLIT. This is possible as we can always apply PARALLEL to split at least the last state element from a state, we cannot apply any rule from *ShorteningRules* to S and S must still satisfy the conditions for the tests in instruction ii. But then w.l.o.g. we have done the same with S' and we obtain the states $t; KB$ and $t'; KB'$. $t'; KB'$ is an instance candidate for $t; KB$ as w.l.o.g. it is not an instance child itself (otherwise we consider its instance father) and we did not apply the GENERALIZATION rule to it since it does not satisfy the conditions of the tests for the generalization step. Thus, another rule was applied to $t'; KB'$ and, hence, the only outgoing edges from $t'; KB'$

are neither INSTANCE nor GENERALIZATION edges. The conditions $|\mathcal{V}(t)| \geq |\mathcal{V}(t')|$ and $|\mathcal{V}(t)| = |\mathcal{V}(t')| \implies |\mathcal{A}(t)| \geq |\mathcal{A}(t')|$ are trivially satisfied as t and t' are equal modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$. Thus, the μ we need for the application of INSTANCE is just the variable renaming between t and t'. W.l.o.g. we can also assume that KB and KB' are equal modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$. The reason for this is that there are only finitely many relevant knowledge bases modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$ which can be obtained during the execution of the standard heuristic. To see this, note that for some relevant knowledge base $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ w.r.t. a state S; $(\mathcal{G}, \mathcal{F}, \mathcal{U})$ we have $\mathcal{G} \subseteq \mathcal{V}(S)$, $\mathcal{F} \subseteq \mathcal{V}(S)$ and $\forall (s, s') \in \mathcal{U} : (\mathcal{V}(s) \cup \mathcal{V}(s')) \cap \mathcal{V}(S) \neq \varnothing \wedge s \sim s'$. As we generalize every non-relevant information implicitly, we know that KB and KB' are relevant knowledge bases w.r.t. t and t'. As $|\mathcal{V}(t)| = |\mathcal{V}(t')|$ is finite, there are only finitely many different sets $\mathcal{G}$ and $\mathcal{F}$ relevant for t and t'. As we also generalize the terms in $\mathcal{U}$, there are only finitely many different pairs of terms modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$ according to Lemma 6.20. Therefore, we can apply INSTANCE to $t; KB$ and $t'; KB'$. Analogously, we find instances for all the terms in Q. Thus, we can reduce the number of state elements after a finite number of generalization steps at least by one in finitely many abstract state-derivation steps and, hence, obtain a finite abstract state-derivation for generalized states using the standard heuristic. Thus, we apply the GENERALIZATION rule only finitely often.

By Lemma 6.21 and the fact that we use the GENERALIZATION rule only finitely often, we know that we can drop all terms having a non-recursive root symbol without introducing new terms or state elements instead in finitely many steps using the standard heuristic.

We are, thus, left to show that we obtain finite abstract state-derivations for terms with recursive root symbols to contradict our assumption. First, we know that the test for a minimal number of nodes from $\{Call(G), Case(G), RepeatG\}$ in instruction (ii) will be true after finitely many steps as χ must contain infinitely many rule applications from the set $\{\text{CALL}, \text{CASE}, \text{REPEAT}\}$. So we will try to find an instance for the current state after finitely many steps. If we do not apply the INSTANCE rule to a state, we continue by checking if the state has to be generalized. As shown above there are only finitely many different terms and knowledge bases we might have in our abstract state-derivation without generalizing them. Since we apply the GENERALIZATION rule only finitely often, we will reach states $S = t, Q \mid S_r; KB$ where there is another state $S' = t', Q' \mid S'_r; KB'$ in our graph and t and t' are equal modulo scope variants and separate variable renaming on $\mathcal{N}$ and $\mathcal{A}$ as in the case above. By the identical argument as before we obtain a finite abstract state-derivation for these states and, hence a finite abstract state-derivation for all states in the partial termination graph. This contradicts our assumption and, hence, proves the lemma. □

As the standard heuristic terminates, we can easily show its correctness which is stated by the following central theorem of this section.

Theorem 6.23 (Correctness of the Standard Heuristic)**.** *The standard heuristic from Definition 6.16 results in a termination graph or an abortion for every partial termination graph G and Prolog program $\mathcal{P}$.*

Proof. As we know by Lemma 6.22 that the standard heuristic terminates, it results in an abortion or a graph where all leaves are empty. In the first case the theorem clearly holds. For the second case we have to show that the resulting graph does not contain any cycles only consisting of INSTANCE and GENERALIZATION edges. As we start with a partial termination graph, this condition is satisfied in the beginning of the standard heuristic. Thus, we are left to show that we do not construct such cycles during the execution of the standard heuristic. As the only rule to construct cycles is INSTANCE and we apply this rule only to instance candidates of the current node in the execution of the standard heuristic, we know that every path from an instance candidate to the current node contains at least one node not from $\mathit{Instance}(G) \cup \mathit{Generalization}(G)$. In particular, the cycle we construct by the application of the INSTANCE rule contains such a node and, hence, cannot consist of INSTANCE and GENERALIZATION edges only. □

6.4 Summary

We introduced an always terminating heuristic which computes a termination graph for every Prolog program and query set as long as the construction of the termination graph does not reach a situation where our approach is stuck due to abstract meta-calls or transformations of terms with infinitely many predication positions. We also explained the parameters of this heuristic to allow for some heuristical tuning according to the set of examples which has to be analyzed. Moreover, we gave some example termination graphs for Prolog programs using several features we have handled in Chapter 3 and Chapter 4.

7 Transformation into Dependency Triple Problems

Once we have constructed a finite termination graph for a Prolog program w.r.t. a class of queries, we still cannot prove termination of this program w.r.t. the class of queries in general as our graph may contain cycles. Thus, we need a way to prove that the cycles in the termination graph cannot be traversed infinitely often from a starting query. While [Sch08] synthesizes a new Prolog program from a termination graph whose termination implies the well-foundedness of the edge relation w.r.t. the starting queries and, hence, termination of the original program w.r.t. the class of queries represented by the root state of the termination graph, we will synthesize DT problems with the same property which allow for a more powerful termination analysis. The reason for this is that DT problems allow for two different kinds of clauses which suits our needs to analyze intermediate goals which occur due to the splitting of goals in the termination graph differently compared to analyzing goals representing the traversal of cycles.

Structure of the Chapter

We start in Section 7.1 by defining the DT problem which is represented by a termination graph. This is illustrated with an example termination graph from the preceding chapter.

In Section 7.2 we continue by proving that termination of the represented DT problem implies termination of the original Prolog program w.r.t. the class of queries represented by the root node of the respective termination graph.

Afterwards we give a number of example transformations for Prolog programs used in this thesis in Section 7.3.

A summary of the contributions of this chapter is given in Section 7.4.

7.1 From Termination Graphs to DT Problems

In this section we show how to synthesize a DT problem from a termination graph built by the abstract rules from the preceding chapters where termination of the DT problem implies termination of the original Prolog program w.r.t. a set of queries for which the termination graph was constructed.

Our goal is to show termination of a Prolog program w.r.t. to a set of queries. This corresponds directly to the termination of the initial state of a termination graph which represents the set of queries. Since we have already proved that our abstract rules are sound, we are left to prove that the cycles in our graph cannot be traversed infinitely often, i.e. the INSTANCE edges cannot be used infinitely often.

Example 7.1. Consider again the termination graph for the Prolog program computing division with remainder from Example 6.3. There are two cycles, one corresponding to the recursive execution of the minus and one for the div predicate. We have to show that these cycles cannot be traversed infinitely often when starting with a query from the initial state. This will prove termination of the original Prolog program w.r.t. the set of queries represented by the initial state.

To this end, [Sch08] builds clauses for a new cut-free logic program where these clauses correspond to the traversal of a path through the termination graph from a state, which is an instance father of another state, to a state, which is an instance child of another state. Termination of this program implies that the cycles in the termination graph cannot be traversed infinitely often. Unfortunately, due to the SPLIT nodes in termination graphs, the clauses may contain intermediate goals representing the left successors of SPLIT nodes, which have to be successfully evaluated before one can evaluate the right successor of a SPLIT node. This evaluation is needed to obtain an answer substitution for the right successor. One problem with this approach is, that we have to build and analyze clauses for the evaluation of intermediate goals, even if these clauses for the answer substitutions do not necessarily have to terminate universally for universal termination of the cycles in the graph.

We try to improve the approach of [Sch08] by building DT problems instead of new logic programs, since we can distinguish between the simulation of the cycles and the evaluation of intermediate goals in DT problems. For the cycles we build DTs and for the intermediate goals we build clauses. A new problem arises with the use of DT problems in form of a call set which we have to specify. In this thesis, we restrict ourselves to use a call set which is general enough to contain all possible calls occurring in the evaluation of the starting queries. We leave a more detailed analysis of the call set to future work.

Example 7.2. Let us use this idea to prove termination of Example 4.22 by building a DT problem for the termination graph from Example 6.3 and prove termination of this represented DT problem instead. We still use the knowledge bases KB_1 to KB_4 defined in Example 6.3.

We use fresh predicates to represent each node with the different variables occurring in the respective node as arguments.

First, we consider the paths in the graph from the root to the beginning of a cycle, i.e., to an INSTANCE node or the successor of an INSTANCE node where we do not traverse such a node between the first and the last node. We look at the substitutions along the path whose composition corresponds to the answer substitution a concrete state-derivation would have along the path. Then we build a DT for the path where the intuition is, that we reach the node (represented by the respective fresh predicate) at the end of the path from the node at the beginning of the path if we apply the corresponding substitutions to the node at the beginning. For INSTANCE nodes, however, we consider their instance father instead where we apply the matching substitution for the respective instance child.

Thus, for the path from $\mathsf{div}(T_1, T_2, T_3, T_4); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\mathsf{div}(T_{23}, T_{13}, T_{14}, T_{15}; KB_4)$, we obtain the head $\mathsf{div}_1(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15})$ and the goal $\mathsf{div}_1(T_{23}, T_{13}, T_{14}, T_{15})$. For the path from $\mathsf{div}(T_1, T_2, T_3, T_4); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\mathsf{minus}(T_{16}, T_{17}, T_{18}); (\{T_{16}, T_{17}\}, \varnothing, \varnothing)$, we obtain the head $\mathsf{div}_1(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), \mathsf{s}(T_{14}), T_{15})$ and the goal $\mathsf{minus}_{22}(T_{16}, T_{17}, T_{18})$.

Now, these paths contain a SPLIT node where the second successor of that node is only reachable if the first successor is successfully evaluated. Hence, we have to add intermediate goals for those paths traversing a SPLIT node along its right successor which correspond to the evaluation of the left successor. We also have to apply the answer substitution to such intermediate goals, but as the nodes only contain terms where the preceding substitutions have already been applied to, it is sufficient to apply only the following substitutions from the respective SPLIT node.

So we obtain the intermediate goal $\mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23})$ for the first of the above paths. Note that we use a different predicate for this intermediate goal than for the goal from the second path since it represents a different node. Altogether, we obtain the two DTs $\mathsf{div}_1(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15}) \leftarrow \mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23}), \mathsf{div}_1(T_{23}, T_{13}, T_{14}, T_{15})$ and $\mathsf{div}_1(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), \mathsf{s}(T_{14}), T_{15}) \leftarrow \mathsf{minus}_{22}(T_{16}, T_{17}, T_{18})$.

Next, we also build DTs for the cycles themselves. Therefore, we consider those paths in the graph starting from a successor of an INSTANCE node and ending in an INSTANCE node or the successor of an INSTANCE node again. Since the root node is a successor of an INSTANCE node, we do not have to consider paths starting from this node again. For the other successor of an INSTANCE node, we have one more path from $\mathsf{minus}(T_{16}, T_{17}, T_{18}); (\{T_{16}, T_{17}\}, \varnothing, \varnothing)$ to $\mathsf{minus}(T_{20}, T_{21}, T_{22}); (\{T_{20}, T_{21}\}, \varnothing, \varnothing)$. So for this path we obtain one more DT $\mathsf{minus}_{22}(\mathsf{s}(T_{20}), \mathsf{s}(T_{21}), T_{22}) \leftarrow \mathsf{minus}_{22}(T_{20}, T_{21}, T_{22})$.

Concerning the evaluation for left successors of SPLIT nodes, we do not build DTs, but normal clauses for the DT problem. Still, the method how we construct the clauses is the same as for the DTs - with the only exception that we consider different paths and that we also may obtain facts. Therefore, we now consider paths starting in a left successor of a SPLIT node and ending in a SUCCESS node. The goal of the corresponding clause is then $\square$. However, in addition to the condition that we do not traverse INSTANCE nodes or their successors, such a path may not traverse another left successor of a SPLIT node

as we are only interested in completely successful evaluations. Thus, the right successor of a SPLIT node must be reached. Here, we have no such path.

Now, the evaluation for left successors of SPLIT nodes may also traverse cycles before it reaches a fact. Hence, we also have to consider paths starting in the left successor of a SPLIT node or the successor of an INSTANCE node and ending in an INSTANCE node, a successor of an INSTANCE node or a SUCCESS node. Still, the conditions for the traversal of other nodes must hold. Thus, we have paths from $\mathsf{div}(T_1, T_2, T_3, T_4); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\mathsf{div}(T_{23}, T_{13}, T_{14}, T_{15}; KB_4)$, from $\mathsf{div}(T_1, T_2, T_3, T_4); (\{T_1, T_2\}, \varnothing, \varnothing)$ to $\square; (\varnothing, \varnothing, \varnothing)$, from $\mathsf{minus}(T_{12}, T_{13}, U); KB_2$ to $\mathsf{minus}(T_{16}, T_{17}, T_{18}); (\{T_{16}, T_{17}\}, \varnothing, \varnothing)$, from $\mathsf{minus}(T_{16}, T_{17}, T_{18}); (\{T_{16}, T_{17}\}, \varnothing, \varnothing)$ to $\mathsf{minus}(T_{20}, T_{21}, T_{22}); (\{T_{20}, T_{21}\}, \varnothing, \varnothing)$ and from $\mathsf{minus}(T_{16}, T_{17}, T_{18}); (\{T_{16}, T_{17}\}, \varnothing, \varnothing)$ to $\square \mid \mathsf{minus}(T_{19}, \mathsf{0}, T_{19}); (\{T_{19}\}, \varnothing, \varnothing)$.

After building clauses for all these paths, we obtain the following DT problem $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ for the termination graph G from Example 6.3 where $\mathcal{D}_G$ contains the DTs

$$\begin{aligned}
\mathsf{div}_1(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23}), \mathsf{div}_1(T_{23}, T_{13}, T_{14}, T_{15}).\\
\mathsf{div}_1(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{22}(T_{16}, T_{17}, T_{18}).\\
\mathsf{minus}_{22}(\mathsf{s}(T_{20}), \mathsf{s}(T_{21}), T_{22}) &\leftarrow \mathsf{minus}_{22}(T_{20}, T_{21}, T_{22}).
\end{aligned}$$

and $\mathcal{P}_G$ consists of the following clauses.

$$\begin{aligned}
\mathsf{div}_{c1}(\mathsf{0}, T_8, \mathsf{0}, \mathsf{0}) &\leftarrow \square.\\
\mathsf{div}_{c1}(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23}), \mathsf{div}_{c1}(T_{23}, T_{13}, T_{14}, T_{15}).\\
\mathsf{minus}_{c19}(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), T_{18}) &\leftarrow \mathsf{minus}_{c22}(T_{16}, T_{17}, T_{18}).\\
\mathsf{minus}_{c22}(T_{19}, \mathsf{0}, T_{19}) &\leftarrow \square.\\
\mathsf{minus}_{c22}(\mathsf{s}(T_{20}), \mathsf{s}(T_{21}), T_{22}) &\leftarrow \mathsf{minus}_{c22}(T_{20}, T_{21}, T_{22}).
\end{aligned}$$

We assume that $\mathcal{C}_G$ contains all reachable goals for queries $\mathsf{div}_1(t_1, t_2, t_3, t_4)$ w.r.t. $\mathcal{D}_G \cup \mathcal{P}_G$ where t_1 and t_2 are ground terms.

This DT problem is easily shown to be terminating by our fully automated termination prover AProVE.

We now show how to obtain a DT problem from a termination graph in general. To this end, we first need the notions of triple and clause paths to characterize paths in the termination graph from which we generate the DTs and clauses respectively for the DT problem.

Definition 7.3 (Triple Path, Clause Path)**.** *A path* $\pi = n_1 \dots n_k$ *is a* triple path *w.r.t.* G *if, and only if,* $k > 1$ *and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G)) \cup \{\mathit{root}(G)\}$
- $n_k \in \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G))$
- *for all* $1 \leq j < k$, $n_j \notin \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G))$

A path $\pi = n_1 \dots n_k$ *is a* clause path *w.r.t.* G *if, and only if,* $k > 1$ *and the following conditions are satisfied:*

- $n_1 \in \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Split}(G))$
- $n_k \in \mathit{Success}(G) \cup \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G))$
- *for all* $1 \leq j < k$, $n_j \notin \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G))$
- *for all* $1 < j \leq k$, $n_{j-1} \in \mathit{Split}(G) \implies n_j = \mathit{Succ}(2, n_{j-1})$

The above definition of triple paths characterizes all non-trivial paths starting at the successor of an INSTANCE or GENERALIZATION node or at the root node of G and ending at an INSTANCE or GENERALIZATION node or their successors while not traversing other nodes of this kind. In other words, triple paths connect the root node with the cycles in the graph and the cycles themselves consist of triple paths again. Subsequent INSTANCE and GENERALIZATION edges will not yield additional paths, since they will be connected by their matching substitutions instead.

Clause paths, however, characterize all non-trivial paths starting at the successor of an INSTANCE or GENERALIZATION node or at the left successor of a SPLIT node and ending at a SUCCESS, INSTANCE or GENERALIZATION node or the successor of an INSTANCE or GENERALIZATION node while not traversing other INSTANCE or GENERALIZATION nodes, their successors or other left children of SPLIT nodes. Clause paths are used to simulate the evaluation of intermediate goals which occur due to SPLIT nodes. Whenever we traverse the right child of a SPLIT node, we need to know that and how the left child of that SPLIT node has been evaluated to use the corresponding answer substitution for the further evaluation. In other words, one can reach the end of a clause or triple path, if all left children of SPLIT nodes along the respective path can be successfully evaluated.

Example 7.4. Consider once more the termination graph G from Example 6.3 for which we synthesized a DT problem in Example 7.2. The paths for which we constructed DTs are in fact all triple paths in G while the paths for which we constructed clauses are all clause paths in G.

The basic idea for the synthesis of a DT problem is to construct a DT for each triple path and a clause for each clause path. The head of the DT or clause respectively is the renamed first state of the corresponding path where we apply the substitutions along the path. The last body term is the renamed last state of the respective path. The intermediate body terms are the renamed left children of SPLIT nodes along the path where we apply the substitutions between the respective SPLIT node and the last node of the path.

As in [Sch08], the renaming is useful, because we can use different predicate symbols for different nodes and, thus, obtain a more precise mapping from the evaluation in the graph to the evaluation for the DT problem. Additionally it simplifies the problem by reducing a whole state with arbitrary many state elements and terms to one atom.

Considering the substitutions we have to apply along a path, we must take care of backtracking, since this cancels the effects of the substitutions belonging to the backtracked state elements. To follow the state element reached at the end of the path, we use a skip value to jump over backtracked or cut state elements preceding the one we are interested in. We gather the substitutions backwards while starting with a skip value of zero as the last state of the path cannot start with a state element which has been backtracked already along the path. Whenever we traverse nodes which backtrack or cut state elements, we have to increase the skip value by the number of backtracked or cut state elements. For this purpose, the second successor state of PARALLEL can be interpreted as a massively backtracked state where we have to increase the skip value by the number of states in the first successor state of PARALLEL. Moreover, the cutting of state elements may only raise the skip value if we are not interested in the first state element as this remains in the successor state. On the other hand, whenever we traverse nodes where we introduce new state elements, we have to reduce the skip value accordingly. Thus, by gathering only those substitutions which are applied to a state element with a skip value of zero, we obtain the correct answer substitution. For those abstract rules which apply a substitution to the state elements following the first, we have to apply these substitutions also for skip values greater than zero, of course.

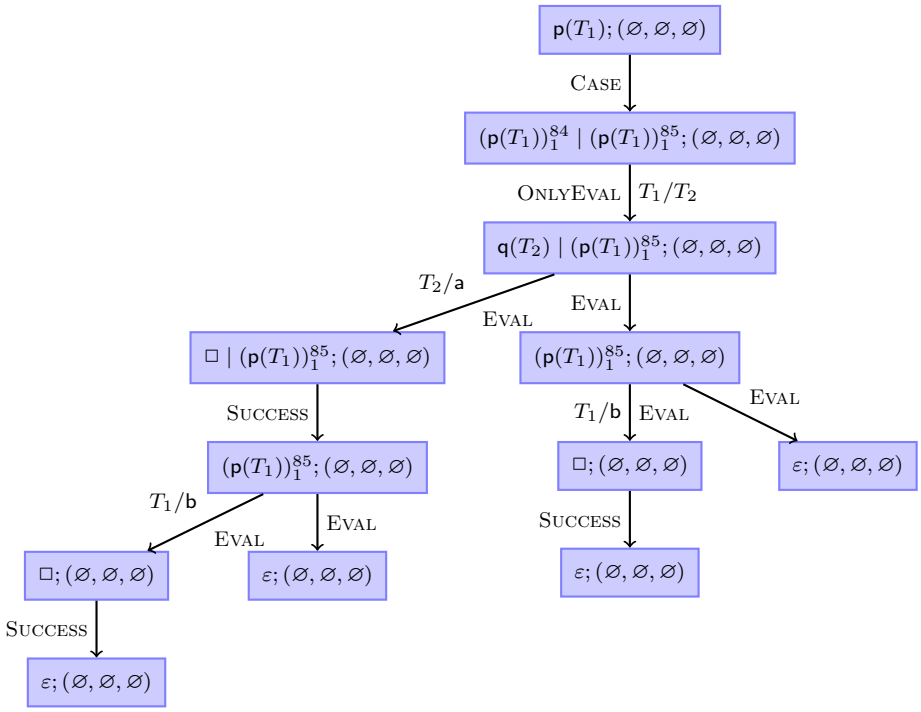
Example 7.5. Consider the following simple Prolog program $\mathcal{P}$

$$\mathsf{p}(X) \leftarrow \mathsf{q}(X). \tag{84}$$

$$\mathsf{p}(\mathsf{b}) \leftarrow \square. \tag{85}$$

$$\mathsf{q}(\mathsf{a}) \leftarrow \square. \tag{86}$$

and the query set $\mathcal{Q} = \{\mathsf{p}(t) \mid \mathsf{p}(t) \in \mathit{PrologTerms}(\Sigma, \mathcal{N})\}$. Using the standard heuristic with the same parameters as in Example 6.17, we obtain the following termination graph for $\mathcal{P}$ and $\mathcal{Q}$.



Now consider the path from the root node to the second last node on the leftmost path in this termination graph. If we successively apply all substitutions along this path the resulting substitution is $[T_1/\mathsf{a}]$. This is not what we want since the answer substitution for the complete path is $[T_1/\mathsf{b}]$. The problem is that we first reach a Success node for the answer substitution $[T_1/\mathsf{a}]$ before we reach the second Success node for the answer substitution $[T_1/\mathsf{b}]$. To detect the latter, we make use of the skip values. Starting at the Success node with a skip value of 0 we do not modify this value along the Eval edge for the left successor of the Eval node while gathering the substitution $[T_1/\mathsf{b}]$. Then we traverse a Success edge and increase the skip value by 1. Thus, for the next Eval edge for the left successor of the Eval node we only gather substitutions for ground terms as the skip value is greater than 0. So here, we only gather *id*. By the same reason we only gather *id* along the next edge for the OnlyEval node. Finally, we traverse a Case edge where we reduce the skip value to 0 again. Altogether, we obtain the intended substitution $[T_1/\mathsf{b}]$.

Note that we do not have to regard the skip value for Split nodes since such nodes must have only one state element and, thus, cannot be backtracked by reaching a non-empty state. Also note that the omitted trailing question marks are not problematic for skip values as no triple or clause path ends in a Failure or empty node.

Now, we define the represented DT problem for a termination graph.

Definition 7.6 (DT Problem from Termination Graph)**.** *The DT Problem $DTP(G)$ for a termination graph $G = (V, E)$ constructed from a Prolog program $\mathcal{P}$ and query set $\mathcal{Q}$ is defined as $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ with $\mathcal{D}_G = \bigcup_{\pi \text{ triple path w.r.t. } G} Triple(\pi)$, $\mathcal{P}_G = \bigcup_{\pi \text{ clause path w.r.t. } G} Clause(\pi)$ and $\mathcal{C}_G \supseteq Call(Start(G), \mathcal{D}_G \cup \mathcal{P}_G)$.*[16]

The set $Start(G)$ is defined as $\{p_{root(G)}(\mathcal{V}(root(G)))\sigma \mid \forall x \in \mathcal{V} : \mathcal{A}(x\sigma) = \varnothing \wedge (x \in \mathcal{G} \implies \mathcal{V}(x\sigma) = \varnothing)$ and $root(G) = S; (\mathcal{G}, \mathcal{F}, \mathcal{U})\}$.

For a path $\pi = n_1 \dots n_k$, we define $Clause(\pi) = Rename(n_1)\sigma_{\pi,0} \leftarrow I_\pi, Rename(n_k)$ and $Triple(\pi) = Rename_{Triple}(n_1)\sigma_{\pi,0} \leftarrow I_\pi, Rename_{Triple}(n_k)$. Here, the Rename function is defined as follows:

$$Rename(n) = \begin{cases} \square & \text{if } n \in Success(G) \\ Rename(Succ(1,n))\mu & \text{if } n \in Instance(G) \cup Generalization(G) \text{ where } \mu \text{ is the substitution associated with } n \\ p_n(\mathcal{V}(n)) & \text{otherwise, where } p_n \text{ is a fresh predicate symbol and } \mathcal{V}(S; KB) = \mathcal{V}(S). \end{cases}$$

Moreover, the $Rename_{Triple}$ function is likewise defined as follows.

$$Rename_{Triple}(n) = \begin{cases} \square & \text{if } n \in Success(G) \\ Rename(Succ(1,n))\mu & \text{if } n \in Instance(G) \cup Generalization(G) \text{ where } \mu \text{ is the substitution associated with } n \\ q_n(\mathcal{V}(n)) & \text{otherwise, where } q_n \text{ is a fresh predicate symbol and } \mathcal{V}(S; KB) = \mathcal{V}(S). \end{cases}$$

The predicate symbols p_n and q_n must be different from each other.

[16] This set can for example be over-approximated by defining a moding function $m_G(p, i) = \mathbf{in}$ if, and only if, for all triple and clause paths $\pi = (n_1; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \dots$ w.r.t. G, whenever $Rename(n_1) = p(x_1, \dots, x_n)$ then $x_i \in \mathcal{G}$.

Furthermore, $\sigma_{\pi,s}$ and I_π are defined as follows:

$$\sigma_{n_1\ldots n_j,s} = \begin{cases} id & \text{if } j = 1 \\ \sigma_{n_1\ldots n_{j-1},s} AnsSub(n_{j-1}) & \text{if } (n_{j-1} \in Split(G),\ n_j = Succ(2, n_{j-1}) \\ & \text{or } n_{j-1} \in EqualsCase(G), n_j = Succ(1, n_{j-i})) \\ & \text{or } (n_{j-1} \in Eval(G) \cup OnlyEval(G) \cup \\ & UnifyCase(G) \cup UnifySuccess(G), \\ & n_j = Succ(1, n_{j-1}), s = 0) \\ \sigma_{n_1\ldots n_{j-1},s} BackSub(n_{j-1}) & \text{if } n_{j-1} \in Eval(G) \cup OnlyEval(G) \cup \\ & UnifyCase(G) \cup UnifySuccess(G), \\ & n_j = Succ(1, n_{j-1}), s > 0 \\ \sigma_{n_1\ldots n_{j-1},s+1} BackSub(n_{j-1}) & \text{if } (n_{j-1} \in NoUnifyCase(G) \cup \\ & UnequalsCase(G), n_j = Succ(2, n_{j-1})) \\ & \text{or } n_{j-1} \in NoUnifyFail(G) \\ \sigma_{n_1\ldots n_{j-1},s}\sigma_{i-1} & \text{if } n_{j-1} \in VarCase(G) \text{ where } n_{j-1} \text{ has more} \\ & \text{than } i \text{ children}, n_j = Succ(i, n_{j-1}) \text{ and} \\ & \sigma_{i-1} \text{ is the substitution used for } n_j \\ \sigma_{n_1\ldots n_{j-1},s+change(n_{j-1},n_j)} & \text{if } (n_{j-1} \in BacktrackSecond(G), \\ & n_j = Succ(2, n_{j-1})) \\ & \text{or } (n_{j-1} \in Backtracking(G)) \\ & \text{or } (n_{j-1} \in VarCase(G) \text{ where } n_{j-1} \text{ has } k \\ & \text{children and } n_j = Succ(k, n_{j-1})) \\ & \text{or } (n_{j-1} \in Cut(G) \text{ and } s > 0) \\ \sigma_{n_1\ldots n_{j-1},reduce(n_{j-1},n_j,s)} & \text{if } n_{j-1} \in Introducing(G) \\ \sigma_{n_1\ldots n_{j-1},s} & \text{otherwise} \end{cases}$$

$$I_{n_j\ldots n_k} = \begin{cases} \square & \text{if } j = k \\ Rename(Succ(1, n_j))\sigma_{n_j\ldots n_k,0}, I_{n_{j+1}\ldots n_k} & \text{if } n_j \in Split(G), n_{j+1} = Succ(2, n_j) \\ I_{n_{j+1}\ldots n_k} & \text{otherwise} \end{cases}$$

Here, $AnsSub : V \to Subst(\Sigma, \mathcal{V})$ and $BackSub : V \to Subst(\Sigma, \mathcal{V})$ are defined by:

$$AnsSub(n) = \begin{cases} \sigma & \text{if } n \in EqualsCase(G) \text{ where } \sigma \text{ is the substitution} \\ & \text{used for } Succ(1,n) \\ \sigma' & \text{if } n \in Eval(G) \cup OnlyEval(G) \cup UnifyCase(G) \\ & \cup UnifySuccess(G) \text{ where } \sigma' \text{ is the substitution} \\ & \text{used for } Succ(1,n) \\ \mu & \text{if } n \in Split(G) \text{ where } \mu \text{ is the substitution} \\ & \text{used for } Succ(2,n) \\ id & \text{otherwise} \end{cases}$$

$$BackSub(n) = \begin{cases} \sigma|_{\mathcal{G}} & \text{if } n \in Eval(G) \cup NoUnifyFail(G) \cup OnlyEval(G) \\ & \cup UnifyCase(G) \cup UnifySuccess(G) \text{ where} \\ & \sigma|_{\mathcal{G}} \text{ is the substitution used for } Succ(1,n) \\ \sigma|_{\mathcal{G}} & \text{if } n \in NoUnifyCase(G) \text{ where } \sigma|_{\mathcal{G}} \text{ is the} \\ & \text{substitution used for } Succ(2,n) \\ \sigma & \text{if } n \in UnequalsCase(G) \text{ where } \sigma \text{ is the substitution} \\ & \text{used for } Succ(2,n) \\ id & \text{otherwise} \end{cases}$$

The functions $change : V \times V \to \mathbb{N}$ *and* $reduce : V \times V \times \mathbb{N} \to \mathbb{N}$ *are defined by:*

$$change(n_1, n_2) = \begin{cases} 1 & \text{if } (n_1 \in BacktrackSecond(G) \setminus \{\textsc{Parallel}\}, n_2 = Succ(2,n_1)) \text{ or} \\ & (n_1 \in Backtracking(G)) \text{ or } (n_1 \in VarCase(G) \text{ where} \\ & n_1 \text{ has } k \text{ children and } n_2 = Succ(k,n_1)) \text{ or} \\ & (n_1 \in Call(G) \cup Disjunction(G) \cup IfThen(G) \cup Repeat(G)) \\ 2 & \text{if } n_1 \in IfThenElse(G) \cup Not(G) \\ k & \text{if } (n_1 \in Parallel(G), n_2 = Succ(2,n_1), \\ & Succ(1,n_1) = S_1 \mid \ldots \mid S_k; KB \text{ where} \\ & S_i \in StateElements\ \forall i \in \{1,\ldots,k\}) \\ & \text{or } (n_1 \in Cut(G), n_1 = !_m, Q \mid S_1 \mid \ldots \mid S_k \mid ?_m \mid S; KB \\ & \text{where } S_i \in StateElements \setminus \{?_m\}\ \forall i \in \{1,\ldots,k\}) \\ & \text{or } (n_1 \in Case(G), n_1 = t, Q \mid S; KB \text{ and } |Slice(\mathcal{P},t)| = k) \\ 0 & \text{otherwise} \end{cases}$$

$$reduce(n_1, n_2, s) = \max(0, s - change(n_1, n_2))$$

Finally, the set Backtracking(G) is defined as the union of the sets

- *AtomicFail*(G)
- *Backtrack*(G)
- *CompoundFail*(G)
- *EqualsFail*(G)
- *Fail*(G)
- *Failure*(G)
- *NonvarFail*(G)
- *Success*(G)
- *UnequalsFail*(G)
- *UnifyFail*(G)
- *VarFail*(G)

while the set BacktrackSecond(G) is defined as the union of the sets

- *AtomicCase*(G)
- *CompoundCase*(G)
- *EqualsCase*(G)
- *Eval*(G)
- *NonvarCase*(G)
- *Parallel*(G)
- *UnifyCase*(G)

and the set Introducing(G) is defined as the union of the sets

- *Call*(G)
- *Case*(G)
- *Disjunction*(G)
- *IfThen*(G)
- *IfThenElse*(G)
- *Not*(G)
- *Repeat*(G)

Example 7.7. The DT problem with the triples

$$\begin{aligned}
\mathsf{div}_1(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23}), \mathsf{div}_1(T_{23}, T_{13}, T_{14}, T_{15}).\\
\mathsf{div}_1(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{22}(T_{16}, T_{17}, T_{18}).\\
\mathsf{minus}_{22}(\mathsf{s}(T_{20}), \mathsf{s}(T_{21}), T_{22}) &\leftarrow \mathsf{minus}_{22}(T_{20}, T_{21}, T_{22}).
\end{aligned}$$

and clauses

$$\begin{aligned}
\mathsf{div}_{c1}(\mathsf{0}, T_8, \mathsf{0}, \mathsf{0}) &\leftarrow \Box.\\
\mathsf{div}_{c1}(T_{12}, T_{13}, \mathsf{s}(T_{14}), T_{15}) &\leftarrow \mathsf{minus}_{c19}(T_{12}, T_{13}, T_{23}), \mathsf{div}_{c1}(T_{23}, T_{13}, T_{14}, T_{15}).\\
\mathsf{minus}_{c19}(\mathsf{s}(T_{16}), \mathsf{s}(T_{17}), T_{18}) &\leftarrow \mathsf{minus}_{c22}(T_{16}, T_{17}, T_{18}).\\
\mathsf{minus}_{c22}(T_{19}, \mathsf{0}, T_{19}) &\leftarrow \Box.\\
\mathsf{minus}_{c22}(\mathsf{s}(T_{20}), \mathsf{s}(T_{21}), T_{22}) &\leftarrow \mathsf{minus}_{c22}(T_{20}, T_{21}, T_{22}).
\end{aligned}$$

from Example 7.2 is in fact the represented DT problem for the termination graph from Example 6.3.

7.2 Proving the Correctness of the Transformation

The following lemmata show how the DTs and clauses of $DTP(G)$ can be used to simulate concrete state-derivations of concrete states described by the root state of the graph. Before we start to state the lemmata, we introduce the notions of a state prefix and extension respectively which will be used in the following proofs.

Definition 7.8 (State Prefix, State Extension)**.** *Let S be a state with $S = S_1 \mid \cdots \mid S_k$ where $\forall i \in \{1, \ldots, k\} : S_i \in StateElements$. Let S' be another state. S is a* state prefix *of S' iff there is a bijection $f : \mathbb{N} \to \mathbb{N}$ and $S' = S'_1 \mid \cdots \mid S'_k \mid S''$ for some state S'' where we have for all $i \in \{1, \ldots, k\}$:*

- $S_i \in \mathbb{N} \implies f(S_i) = S'_i$
- $S_i = \square \implies S_i = S'_i$
- $S_i = Q \implies S'_i = Q', Q''$ *for some list of terms Q'' where* $Q' = Q\xi$
- $S_i = (Q)^n_m \implies S'_i = (Q', Q'')^n_{f(m)}$ *for some list of terms Q'' where* $Q' = Q\xi$

Here, we define $\xi = [!_i/!_{f(i)} \forall i \in \mathbb{N}]$.

For two states S and S', S' is a state extension *of S iff S is a state prefix of S'.*

Example 7.9. Consider the state $S = t_1, t_2 \mid (t_3)^i_m$. The state t_1 is a state prefix of S while the state $t_1, t_2 \mid (t_3)^i_m \mid (t_4)^{i'}_{m'}$ is a state extension of S.

The notions of a state prefix and extension respectively are useful to describe the connection between a termination graph and the concrete state-derivations it represents. Due to the splitting of backtracking lists and goals with the rules Parallel and Split, the concrete state-derivation may contain states which are not represented by only one abstract state, but by several different abstract states instead. Still, we have to take this difference into account while we prove the correctness of our transformation.

Thus, for the simulation of concrete state-derivations by abstract state-derivations, we need to follow not only linear paths, but *tree paths* in a termination graph. This is also due to the splitting of goals by the SPLIT rule and to the splitting of backtracking lists we encounter at PARALLEL nodes. The following definition therefore gives us a structure for describing the way of a concrete state-derivation through a termination graph.

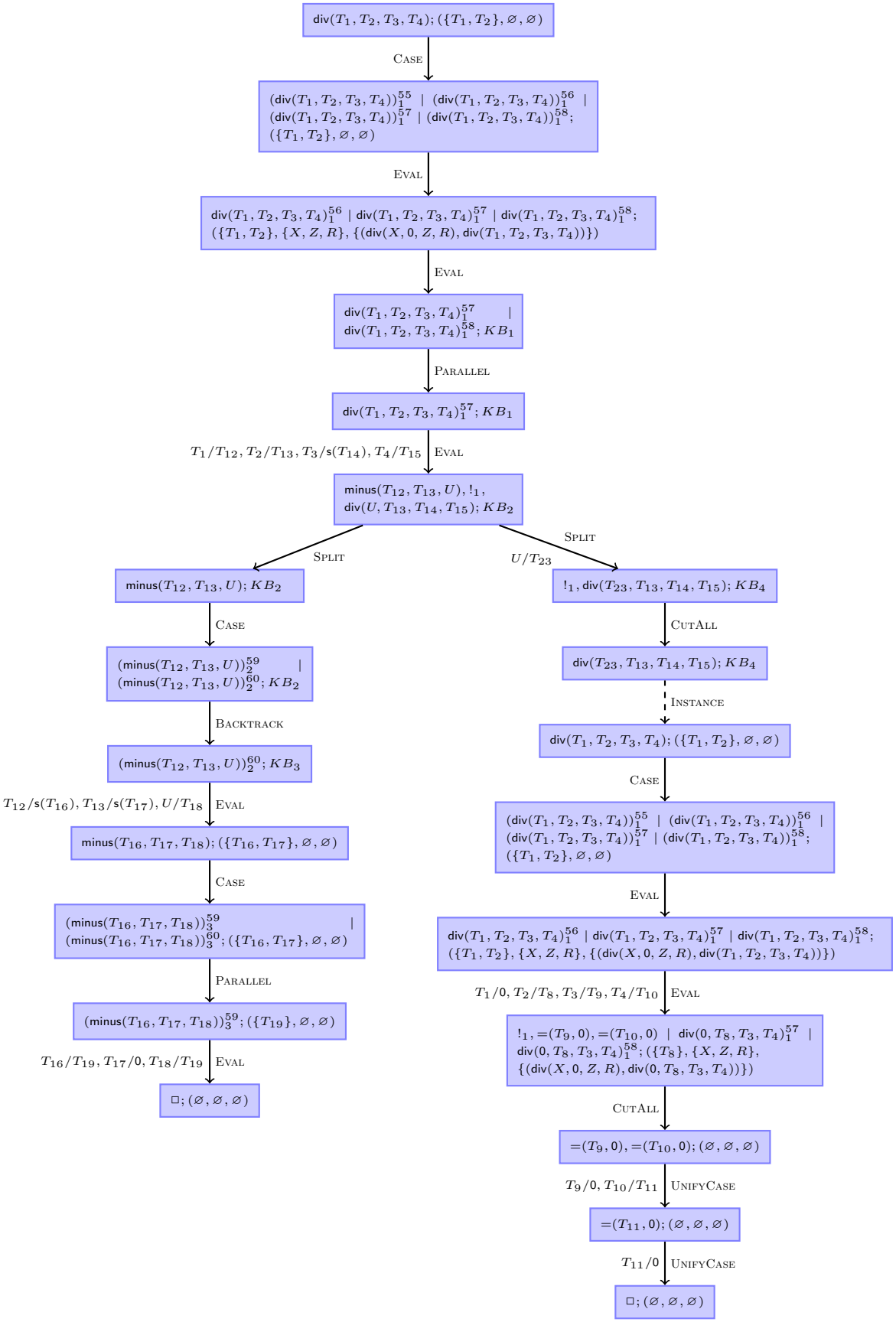
Definition 7.10 (Tree Path)**.** *For a termination graph $G = (V, E)$ we call a (possibly infinite) word $\pi = (n_0, v_0, p_0), (n_1, v_1, p_1), (n_2, v_2, p_2), \ldots$ over the set $\mathbb{N} \times V \times (\mathbb{N} \cup \{none\})$ a* tree path *w.r.t. G iff the following conditions are satisfied for all $i, j \in \mathbb{N}$:*

- $p_0 = none$,
- $n_i = n_j \implies i = j$,
- $p_i = none \implies i = 0$,
- $p_i \in \{n_0, n_1, n_2, \ldots\}$,
- $n_i = p_j \implies (v_i, v_j) \in E$ *and*
- $p_i < n_i$
- *there are indices $i_0, \ldots, i_{m_i} \in \{n_0, n_1, n_2, \ldots\}$ with $i_{m_i} = 0$, $i_0 = i$ and $p_{i_{r-1}} = n_{i_r}$ for all $r \in \{1, \ldots, m_i\}$.*

We call (n_i, v_i, p_i) a leaf of π iff there is no $(n_j, v_j, p_j) \in \pi$ with $p_j = n_i$. For (n_i, v_i, p_i) and (n_j, v_j, p_j) we call (n_i, v_i, p_i) an ancestor of (n_j, v_j, p_j) iff there are indices $i_0, \ldots, i_{m_i} \in \{n_0, n_1, n_2, \ldots\}$ with $i_{m_i} = i$, $i_0 = j$ and $p_{i_{r-1}} = n_{i_r}$ for all $r \in \{1, \ldots, m_i\}$.

To really follow a complete concrete state-derivation we would have to fork on PARALLEL nodes, but as we will be interested in the relevant parts of the concrete state-derivations for the reached states only, we may skip the failing branches due to backtracking. Thus, the only nodes where we have to fork our tree path are SPLIT nodes.

Example 7.11. Consider for the last time the termination graph from Example 6.3 and the concrete derivation from Example 5.9. The corresponding tree path is given as follows.



Lemma 7.12 (Success Tree for Concrete State-Derivations in Termination Graph)**.** *Let $S\gamma \in \mathcal{CON}(S; KB)$ with $S; KB = n \in G$ for a termination graph $G = (V, E)$ and there is a concrete state-derivation with l steps from $S\gamma$ to a state S''. Then there is a node $n' \in V$, a concretization γ' and a variable renaming ρ on $\mathcal{N}$ with $n' = S'; KB'$, $S'\gamma' \in \mathcal{CON}(S'; KB')$, $S'\gamma'\rho$ is a state prefix of S'' and there is a tree path $\pi = (0, v_0, p_0), \ldots, (k, v_k, p_k)$ w.r.t. G with the following properties:*

- $v_0 = n$
- *for all $i \in \{0, \ldots, k\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the concrete state-derivation reaches a scope variant of a state extension of $S_i\gamma_i\rho_i$ in $l_i \leq l$ steps where $v_i = S_i; KB_i$ and $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$*
- *for all leaves (i, v_i, p_i) of π with $i \neq k$ we have $v_i \in Success(G)$*
- *for all (i, v_i, p_i) with more than one successor in π, we have $v_i \in Split(G)$*
- *for all (i, v_i, p_i) with $v_i \in Split(G)$ and only one successor (j, v_j, i) in π, we have $v_j = Succ(1, v_i)$*
- $v_k = n'$

Proof. We perform the proof by induction over the lexicographic combination of first the length l of the concrete state-derivation and second the edge relation of G'. Here, G' is like G except that it only contains outgoing edges of Instance, Generalization, Parallel and Split nodes. Note that this induction relation is indeed well-founded as G' is an acyclic and finite graph. The reason is that when traversing nodes $(S; KB)$ in G' the number of terms in S cannot increase. Since this number is strictly decreased in Parallel and Split nodes any infinite path in G' must in the end only traverse Instance and Generalization nodes. This is in contradiction to the definition of termination graphs which disallows cycles consisting only of Instance and Generalization edges.

We first show that the lemma holds for nodes $S; KB$ where one of the abstract rules Instance, Generalization, Parallel or Split have been applied. Here, whenever we have to define the concretization γ' and the variable renaming ρ and if these are not specified then $\gamma' = \gamma$ and $\rho = id$.

- If we applied the Instance or Generalization rule to n, we have $Succ(1, n) = S'; KB'$ with $S = S'''\mu$ where S''' is a scope variant of S'. By Lemma 3.37 and Lemma 3.38 we know that there is a concretization γ'' such that $S'\gamma'' \in \mathcal{CON}(S'; KB')$ and $S\gamma = S'''\gamma''\mu|_{\mathcal{N}}$. As $\mu|_{\mathcal{N}}$ is a variable renaming and S''' is a scope variant of S' we conclude that the concrete state-derivation from $S\gamma$ to a state extension of S'' can be completely simulated by a corresponding concrete state-derivation from $S'\gamma''$ to a state extension of S'''' of length l where the only

difference is the application of $\mu|_{\mathcal{N}}$. To be more precise, if S_i is the i-th state in the concrete state-derivation from $S\gamma$ to a state extension of S'' then there also is an i-th state S'_i in the concrete state-derivation from $S'\gamma''$ to a state extension of S'''' and $S'_i\mu|_{\mathcal{N}} = S_i$. Hence, we can use the induction hypothesis for the latter concrete state-derivation to obtain a tree path π' with root $S'; KB'$. To obtain π from π' we first modify all variable renamings by additionally adding $\mu|_{\mathcal{N}}$ ($\rho_i = \rho'_i\mu|_{\mathcal{N}}$). Then we add the node $S; KB$ as new root and start the path with the edge from $S; KB$ to S', KB'.

- If we applied the PARALLEL rule to n, we reach two states $S_1; KB$ and $S_2; KB$ where $S = S_1 \mid S_2$. There are two cases depending on whether the concrete state-derivation reaches a state extension of $S_2\gamma$. If the concrete state-derivation reaches such a state, we use $Succ(2, n)$ instead of n and insert the path from n to $Succ(2, n)$ before the tree path we obtain by the induction hypothesis for $Succ(2, n)$. If the concrete state-derivation does not reach such a state, we know from the soundness proof of PARALLEL that a state prefix of S'' must be reachable from $S_1\gamma$ and as we clearly have that S_1 is a state prefix of S, we use $Succ(1, n)$ instead of n and insert the path from n to $Succ(1, n)$ before the tree path we obtain for $Succ(1, n)$ by the induction hypothesis.

- If we applied the SPLIT rule to n, we know that $S = t, Q$, $Succ(1, n) = t; KB$ and $Succ(2, n) = Q\mu; KB'$.

  If the concrete state-derivation reaches a state extension of $Q\gamma\mu'$ for some answer substitution μ', we know by Lemma 3.51 that there is a concretization γ' w.r.t. KB' such that $Q\gamma\mu' = Q\mu\gamma'$. Additionally, we know that the concrete state-derivation reaches a state extension of $\square$ from $t\gamma$. As this concrete state-derivation is shorter than the one of $(t, Q)\gamma$ we obtain a node $n'' \in Success(G)$ and a tree path π' for $Succ(1, n)$ by the induction hypothesis. Also, we obtain a node n''' and a tree path π'' for $Succ(2, n)$ by the induction hypothesis for the concrete state-derivation of $(Q\mu)\gamma'$ to S''. Using γ' and id for $Succ(2, n)$, we obtain the node $n' = n'''$ and the desired tree path π by using n as the root with π' as its left and π'' as its right subtree path.

  If the concrete state-derivation does not reach a state extension of $Q\gamma\mu'$ for any answer substitution μ', we know by the soundness proof of SPLIT that a state prefix of S'' must be reachable from $t\gamma$ within l steps. Hence, we can apply the induction hypothesis and add $(S; KB)$ as a new root with only one edge to $(t; KB)$.

For $l = 0$ we know that $S\gamma = S'' \in \mathcal{CON}(S; KB)$. Thus, for $\gamma_0 = \gamma$, $\rho_0 = id$ and $n' = n$ we obtain $\pi = (0, n, none)$ as the desired tree path. For $l > 0$, we can assume the lemma holds for concrete state-derivations of length at most $l - 1$.

We perform a case analysis over the first concrete inference rule applied in the concrete state-derivation where we can assume that the abstract inference rules INSTANCE, GENERALIZATION, PARALLEL and SPLIT were not applied to n.

- For CASE we have $S = t, Q \mid S_r$ where $root(t) \notin BuiltInPredicates$ and $Slice(\mathcal{P}, t) \neq \varnothing$ and the concrete state-derivation reaches the state $(t, Q)^{i_1}_j\gamma \mid \cdots \mid (t, Q)^{i_m}_j\gamma \mid S_r\gamma$. So the only applicable abstract inference rule for n is CASE.

  By applying the CASE rule to n, we reach the state $n'' = (t, Q)^{i_1}_j \mid \cdots \mid (t, Q)^{i_m}_j \mid S_r; KB$. By the induction hypothesis we obtain a node n''' and a tree path π' with the properties in Lemma 7.12 for n''. We obtain the desired node $n' = n'''$ and the tree path π by inserting the path from n to n'' before π' for n''.

- For SUCCESS we have $S = \square \mid S_r$ and the concrete state-derivation reaches the state $S_r\gamma$. So the only applicable abstract inference rule for n is SUCCESS.

  By applying the SUCCESS rule, we reach the state $S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π' using γ and id for $S_r; KB$.

- For FAILURE, CUT and CUTALL the proof is analogous to the case where the SUCCESS rule is the first rule in the concrete state-derivation.

- For VARIABLEERROR we have $S = \mathsf{call}(x), Q \mid S_r$ and the concrete state-derivation reaches the state ε. So the only applicable abstract inference rule for n is VARIABLEERROR.

  By applying the VARIABLEERROR rule, we reach the state $\varepsilon; KB$. As the concrete state-derivation has to end here, we obtain the desired node $\varepsilon; KB$ and the tree path $\pi = (0, S; KB, none), (1, \varepsilon; KB, 0)$ using id and id for $\varepsilon; KB$.

- For UNDEFINEDERROR, THROW, HALT and HALT1 the proof is analogous to the case where the VARIABLEERROR rule is the first rule in the concrete state-derivation.

- For EVAL we have $S = (t, Q)^i_j \mid S_r$ and the concrete state-derivation reaches the state $B'_i\sigma, Q\gamma\sigma \mid S_r\gamma$ as defined in the EVAL rule. From the soundness proof of BACKTRACK we know that the only applicable abstract inference rules for n are EVAL and ONLYEVAL.

  If we applied the EVAL rule we have $Succ(1, n) = B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ as defined in EVAL. From the soundness proof of EVAL we know that there is a concretization γ'' w.r.t. KB' with $B'_i\sigma'\gamma'', Q\sigma'\gamma'' \mid S_r\sigma|_{\mathcal{G}}\gamma'' = B'_i\sigma, Q\gamma\sigma \mid S_r\gamma$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $B'_i\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$. Thus, we obtain the desired node n' and the tree path

π by inserting the path from n to $B_i'\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ before π' using γ'' and id for $B_i'\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$.

If we applied the OnlyEval rule we have $Succ(1, n) = B_i'\sigma', Q\sigma' \mid S_r\sigma|_{\mathcal{G}}; KB'$ again and, hence, the same case as for Eval.

- For Backtrack we have $S = (t, Q)^i_j \mid S_r$ and the concrete state-derivation reaches the state $S_r\gamma$. From the soundness proof of OnlyEval we know that the only applicable abstract inference rules for n are Eval and Backtrack.

  If we applied the Eval rule we have $Succ(2, n) = S_r; KB'$ as defined in Eval where we know by the soundness proof of Eval that γ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB'$ before π' using γ and id for $S_r; KB'$.

  If we applied the Backtrack rule we have $Succ(1, n) = S_r; KB'$ and, hence, the same case for $Succ(1, n)$ here as for $Succ(2, n)$ in the case of Eval.

- For Call we have $S = \mathsf{call}(t'), Q \mid S_r$ and the concrete state-derivation reaches the state $t''\gamma, Q\gamma \mid ?_m \mid S_r\gamma$ for $t'' = \mathit{Transformed}(t', m)$. So the only applicable abstract inference rule for n is Call.

  By applying the Call rule, we reach the state $t'', Q \mid ?_m \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $t'', Q \mid ?_m \mid S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $t'', Q \mid ?_m \mid S_r; KB$ before π' using γ and id for $t'', Q \mid ?_m \mid S_r; KB$.

- For AtomicFail we have $S = \mathsf{atomic}(t'), Q \mid S_r$ where $t'\gamma$ is no constant and the concrete state-derivation reaches the state $S_r\gamma$. So the only applicable abstract inference rules for n are AtomicFail and AtomicCase.

  If we applied the AtomicCase rule, we have $Succ(2, n) = S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π' using γ and id for $S_r; KB$.

  If we applied the AtomicFail rule, we have $Succ(1, n) = S_r; KB$ and, hence, the same case for $Succ(1, n)$ here as for $Succ(2, n)$ in the case of AtomicCase.

- For AtomicSuccess we have $S = \mathsf{atomic}(t'), Q \mid S_r$ where $t'\gamma$ is a constant and the concrete state-derivation reaches the state $Q\gamma \mid S_r\gamma$. So the only applicable abstract inference rules for n are AtomicSuccess and AtomicCase.

If we applied the AtomicCase rule, we have $Succ(1, n) = Q \mid S_r; KB'$ as defined for AtomicCase where we know by the soundness proof of AtomicCase that γ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q \mid S_r; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q \mid S_r; KB'$ before π' using γ and id for $Q \mid S_r; KB'$.

If we applied the AtomicSuccess rule, we have $Succ(1, n) = Q \mid S_r; KB'$ again and, hence, the same case as for AtomicCase.

- For CompoundFail, EqualsFail and NonvarFail the proof is analogous to the case for AtomicFail.

- For CompoundSuccess, NonvarSuccess, NoUnifySuccess and UnequalsSuccess the proof is analogous to the case for AtomicSuccess.

- For Conjunction we have $S = ,(t_1, t_2), Q \mid S_r$ and the concrete state-derivation reaches the state $t_1, t_2, Q \mid S_r$. So the only applicable abstract inference rule for n is Conjunction.

  By applying the Conjunction rule, we have $Succ(1, n) = t_1, t_2, Q \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $t_1, t_2, Q \mid S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $t_1, t_2, Q \mid S_r; KB$ before π' using γ and id for $t_1, t_2, Q \mid S_r; KB$.

- For Disjunction, IfThen, IfThenElse, Not, Once and Repeat the proof is analogous to the case where the Conjunction rule is the first rule in the concrete state-derivation.

- For EqualsSuccess we have $S = ==(t_1, t_2), Q \mid S_r$ where $t_1\gamma = t_2\gamma$ and the concrete state-derivation reaches the state $Q\gamma \mid S_r\gamma$. So the only applicable abstract inference rules for n are EqualsSuccess and EqualsCase.

  If we applied the EqualsCase rule, we have $Succ(1, n) = Q\sigma \mid S_r\sigma; KB'$ as defined for EqualsCase where we know by the soundness proof of EqualsCase that $\gamma = \sigma\gamma$ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q\sigma \mid S_r\sigma; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q\sigma \mid S_r\sigma; KB'$ before π' using γ and id for $Q\sigma \mid S_r\sigma; KB'$.

  If we applied the EqualsSuccess rule, we have $t_1 = t_2$ and $Succ(1, n) = Q \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q \mid S_r; KB$. Thus, we obtain the desired node n'

and the tree path π by inserting the path from n to $Q \mid S_r; KB$ before π' using γ and id for $Q \mid S_r; KB$.

- For Fail we have $S = \mathsf{fail}, Q \mid S_r$. So the only applicable abstract inference rule for n is Fail.

  By applying the Fail rule, we have $Succ(1, n) = S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π' using γ and id for $S_r; KB$.

- For Newline we have $S = \mathsf{nl}, Q \mid S_r$. So the only applicable abstract inference rule for n is Newline.

  By applying the Newline rule, we have $Succ(1, n) = Q \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q \mid S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q \mid S_r; KB$ before π'.

- For NoUnifyFail we have $S = \backslash{=}(t_1, t_2), Q \mid S_r$ where $t_1\gamma \sim t_2\gamma$ and the concrete state-derivation reaches the state $S_r\gamma$. From the soundness proof of NoUnifySuccess we know that the only applicable abstract inference rules for n are NoUnifyCase and NoUnifyFail.

  If we applied the NoUnifyCase rule we have $Succ(2, n) = S_r\sigma|_{\mathcal{G}}; KB'$ as defined in NoUnifyCase. From the soundness proof of NoUnifyCase we know that $\gamma = \sigma_{\mathcal{G}}\gamma$ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r\sigma|_{\mathcal{G}}; KB'$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r\sigma|_{\mathcal{G}}; KB'$ before π' using γ and id for $S_r\sigma|_{\mathcal{G}}; KB'$.

  If we applied the NoUnifyFail rule we have $Succ(1, n) = S_r\gamma|_{\mathcal{G}}; KB'$ again and, hence the same case for $Succ(1, n)$ here as for $Succ(2, n)$ in the case of NoUnifyCase.

- For True the proof is analogous to the case for Newline.

- For UnequalsFail we have $S = \backslash{=}{=}(t_1, t_2), Q \mid S_r$ where $t_1\gamma = t_2\gamma$ and the concrete state-derivation reaches the state $S_r\gamma$. So the only applicable abstract inference rules for n are UnequalsFail and UnequalsCase.

  If we applied the UnequalsCase rule, we have $Succ(2, n) = S_r\sigma; KB'$ as defined for UnequalsCase where we know by the soundness proof of UnequalsCase that $\gamma = \sigma\gamma$ is a concretization w.r.t. KB'. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r\sigma; KB'$. Thus,

we obtain the desired node n' and the tree path π by inserting the path from n to $S_r\sigma; KB'$ before π' using γ and id for $S_r\sigma; KB'$.

If we applied the UnequalsFail rule, we have $t_1 = t_2$ and $Succ(1, n) = S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π' using γ and id for $S_r; KB$.

- For UnifyFail the proof is analogous to the case for Backtrack.
- For UnifySuccess the proof is analogous to the case for Eval.
- For VarFail we have $S = \mathsf{var}(t'), Q \mid S_r$ where $t'\gamma$ is no variable and the concrete state-derivation reaches the state $S_r\gamma$. So the only applicable abstract inference rules for n are VarFail and VarCase.

  If we applied the VarCase rule and n has j children, we have $Succ(j, n) = S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $S_r; KB$ before π' using γ and id for $S_r; KB$.

  If we applied the VarFail rule, we have $Succ(1, n) = S_r; KB$ and, hence, the same case for $Succ(1, n)$ here as for $Succ(j, n)$ in the case of VarCase.

- For VarSuccess we have $S = \mathsf{var}(t'), Q \mid S_r$ where $t'\gamma \in \mathcal{N}$ and the concrete state-derivation reaches the state $Q\gamma \mid S_r\gamma$. So the only applicable abstract inference rules for n are VarSuccess and VarCase.

  Since $t'\gamma \in \mathcal{N}$, we know that $t' \in \mathcal{A} \setminus \mathcal{G}$.

  If we applied the VarCase rule and n has j children, there are two cases depending on whether $t'\gamma \in \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB)$.

  If $t'\gamma \in \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB)$ there is an index j' with $1 < j' < j$ and $Succ(j', n) = Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB\sigma_{j'+1}$ where $\sigma_{j'+1} = [t'/t'\gamma]$. Thus we have $Q\sigma_{j'+1}\gamma \mid S_r\sigma_{j'+1}\gamma = Q\gamma \mid S_r\gamma$ and γ is a concretization w.r.t. $KB\sigma_{j'+1}$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB\sigma_{j'+1}$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q\sigma_{j'+1} \mid S_r\sigma_{j'+1}; KB\sigma_{j'+1}$ before π'.

  If $t'\gamma \notin \mathcal{N}(Q) \cup \mathcal{N}(S_r) \cup \mathcal{N}(KB)$, we have $Succ(1, n) = Q\sigma_0 \mid S_r\sigma_0; KB\sigma_0$ where $\sigma_0 = [t'/x]$ and $x \in \mathcal{N}_{fresh}$. By the soundness proof of VarCase we know that there is a concretization γ'' w.r.t. $KB\sigma_0$ and a variable renaming ρ' on $\mathcal{N}$ such that $\gamma\rho' = \gamma''$ and $Q\sigma_0\gamma\rho' \mid S_r\sigma_0\gamma\rho' = Q\sigma_0\gamma'' \mid S_r\sigma_0\gamma'' = Q\gamma'' \mid S_r\gamma''$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q\sigma_0 \mid S_r\sigma_0; KB\sigma_0$. Thus, using γ and ρ' for $Q\sigma_0 \mid S_r\sigma_0; KB\sigma_0$

we obtain the desired node n' and the tree path π by inserting the path from n to $Q\sigma_0 \mid S_r\sigma_0; KB\sigma_0$ before π'.

If we applied the VarSuccess rule, we have $Succ(1, n) = Q \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q \mid S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q \mid S_r; KB$ before π'.

- For Write we have $S = \mathsf{write}(t'), Q \mid S_r$. So the only applicable abstract inference rule for n is Write.

  By applying the Write rule, we have $Succ(1, n) = Q \mid S_r; KB$. By the induction hypothesis we obtain a node n' and a tree path π' with the properties in Lemma 7.12 for $Q \mid S_r; KB$. Thus, we obtain the desired node n' and the tree path π by inserting the path from n to $Q \mid S_r; KB$ before π'.

- For WriteCanonical and WriteQ the proof is analogous to the case for Write.

□

Next, we show that we can use the same concretization and variable renaming as long as we do not traverse Instance or Generalization nodes.

Lemma 7.13 (Single Concretization and Variable Renaming)**.** *Given a path $\pi = n_1 \dots n_k$ with $n_j \notin Instance(G) \cup Generalization(G)$ for all $j \in \{1, \dots, k-1\}$ and a concrete state-derivation such that there are variable renamings $\rho_1, \dots, \rho_k$ and concretizations $\gamma_1, \dots, \gamma_k$ w.r.t. $KB_1, \dots, KB_k$ where $n_i = S_i; KB_i$ for all $i \in \{1, \dots, k\}$ and the concrete state-derivation goes from a state extension of $S_1\gamma_1\rho_1$ to a state extension of $S_k\gamma_k\rho_k$ by reaching state extensions of all $S_i\gamma_i\rho_i$, then there is a variable renaming ρ and a concretization γ w.r.t. all knowledge bases KB_i such that $S_i\gamma_i\rho_i = S_i\gamma\rho$.*

Proof. We perform the proof by induction over the length k of the path π.

For $k = 1$ we have $n_1 = n_k$ and only one variable renaming and concretization $\gamma_1\rho_1 = \gamma\rho$. Hence, the lemma trivially holds.

For $k > 1$ we can assume the lemma holds for paths of length at most $k - 1$. By inspection of all abstract inference rules other than Instance and Generalization we know that only fresh variables are introduced by these rules. We perform a case analysis over n_1 and n_2.

- If $n_1 \in Split(G)$ and $n_2 = Succ(2, n_1)$, i.e., we traverse the right child of a Split node, we have $n_1 = t, Q; KB$ and $n_2 = Q\mu; KB'$ as defined in the Split rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \dots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $Q\mu\gamma_2\rho_2 = Q\mu\gamma'\rho$. By Lemma 3.51 and the fact that the concrete state-derivation reaches a state extension of $Q\mu\gamma_2\rho_2$ from a state extension

of $(t, Q)\gamma_1\rho_1$ with some answer substitution μ', we obtain $\gamma_1\rho_1\mu' = \mu\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(KB)}$ and $\rho_1 = \rho_2$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\mu) \cup \mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j) \cup \mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\mu) \cup \mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \ldots, k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{Eval}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of an Eval node, we have $n_1 = (t, Q)^c_m \mid S; KB$ and $n_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the Eval rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \ldots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $B'_c\sigma'\gamma_2\rho_2, Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = B'_c\sigma'\gamma'\rho, Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By Lemma 3.28 and the fact that the concrete state-derivation reaches a state extension of $B'_c\sigma'\gamma_2\rho_2, Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(t, Q)^c_m\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(B'_c\sigma') \cup \mathcal{A}(Q\sigma') \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j) \cup \mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(B'_c\sigma') \cup \mathcal{A}(Q\sigma') \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \ldots, k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{OnlyEval}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse an OnlyEval node, we have $n_1 = (t, Q)^c_m \mid S; KB$ and $n_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the OnlyEval rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2, \ldots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $B'_c\sigma'\gamma_2\rho_2, Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = B'_c\sigma'\gamma'\rho, Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By Lemma 3.29 and the fact that the concrete state-derivation reaches a state extension of $B'_c\sigma'\gamma_2\rho_2, Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(t, Q)^c_m\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with

$\gamma_1|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)) \setminus (\mathcal{A}(B'_c\sigma')\cup\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j)\cup\mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(B'_c\sigma')\cup\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1,\ldots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\ldots,k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{UnifyCase}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of a UnifyCase node, we have $n_1 = =(t_1,t_2), Q \mid S; KB$ and $n_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifyCase rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2,\ldots,k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of UnifyCase and the fact that the concrete state-derivation reaches a state extension of $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(=(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j)\cup\mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1,\ldots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\ldots,k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{UnifySuccess}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse a UnifySuccess node, we have $n_1 = =(t_1,t_2), Q \mid S; KB$ and $n_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifySuccess rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2,\ldots,k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = Q\sigma'\gamma'\rho \mid S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of UnifySuccess and the fact that the concrete state-derivation reaches a state extension of $Q\sigma'\gamma_2\rho_2 \mid S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(=(t_1,t_2),Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain

$\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j)\cup\mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1,\ldots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\ldots,k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{NoUnifyCase}(G)$ and $n_2 = \mathit{Succ}(2, n_1)$, i.e., we traverse the right child of a NoUnifyCase node, we have $n_1 = \backslash{=}(t_1, t_2), Q \mid S; KB$ and $n_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NoUnifyCase rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2,\ldots,k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of NoUnifyCase and the fact that the concrete state-derivation reaches a state extension of $S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(\backslash{=}(t_1, t_2), Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced along π, we have for all abstract variables $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j)\cup\mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB))\setminus(\mathcal{A}(Q\sigma')\cup\mathcal{A}(S\sigma|_{\mathcal{G}})\cup\mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1,\ldots,k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2,\ldots,k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{NoUnifyFail}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse a NoUnifyFail node, we have $n_1 = \backslash{=}(t_1, t_2), Q \mid S; KB$ and $n_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NoUnifyFail rule. By the induction hypothesis we obtain a variable renaming ρ and a concretization γ' w.r.t. KB_j for all $j \in \{2,\ldots,k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho$. In particular, we have $S\sigma|_{\mathcal{G}}\gamma_2\rho_2 = S\sigma|_{\mathcal{G}}\gamma'\rho$. By the soundness proof of NoUnifyFail and the fact that the concrete state-derivation reaches a state extension of $S\sigma|_{\mathcal{G}}\gamma_2\rho_2$ from a state extension of $(\backslash{=}(t_1, t_2), Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with answer substitution σ'' and $\rho_1 = \rho_2$, we obtain $\gamma_1\rho_1\sigma'' = \sigma'\gamma_2\rho_2$ with $\gamma_1|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)} = \gamma_2|_{\mathcal{A}(t_1)\cup\mathcal{A}(t_2)\cup\mathcal{A}(Q)\cup\mathcal{A}(S)\cup\mathcal{A}(KB)}$. Since only fresh abstract variables are introduced

along π, we have for all abstract variables $T \in (\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma') \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ that $T \notin \mathcal{A}(S_j) \cup \mathcal{A}(KB_j)$. Hence, we can define the concretization γ by $T\gamma = T\gamma_1$ for $T \in (\mathcal{A}(t_1) \cup \mathcal{A}(t_2) \cup \mathcal{A}(Q) \cup \mathcal{A}(S) \cup \mathcal{A}(KB)) \setminus (\mathcal{A}(Q\sigma') \cup \mathcal{A}(S\sigma|_{\mathcal{G}}) \cup \mathcal{A}(KB'))$ and $T\gamma = T\gamma'$ otherwise. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho$ for all $j \in \{2, \ldots, k\}$. As γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- If $n_1 \in \mathit{VarCase}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the first child of a VarCase node where we introduce a fresh non-abstract variable, we have $n_1 = \mathsf{var}(a), Q \mid S; KB$ with $a \in \mathcal{A} \setminus \mathcal{G}$ and $n_2 = Q\sigma_0 \mid S\sigma_0; KB\sigma_0$ with $\sigma_0 = [a/x]$ and $x \in \mathcal{N}_{\mathit{fresh}}$ as defined in the VarCase rule. By the induction hypothesis we obtain a variable renaming ρ' and a concretization γ' w.r.t. KB_j for all $j \in \{2, \ldots, k\}$ such that $S_j\gamma_j\rho_j = S_j\gamma'\rho'$. In particular, we have $Q\sigma_0\gamma_2\rho_2 \mid S\sigma_0\gamma_2\rho_2 = Q\sigma_0\gamma'\rho' \mid S\sigma_0\gamma'\rho'$. By the soundness proof of VarCase and the fact that the concrete state-derivation reaches a state extension of $Q\sigma_0\gamma_2\rho_2 \mid S\sigma_0\gamma_2\rho_2$ from a state extension of $(\mathsf{var}(a), Q)\gamma_1\rho_1 \mid S\gamma_1\rho_1$ with an empty answer substitution, we obtain a variable renaming ρ such that $\gamma_1\rho_1\rho = \sigma_0\gamma_1\rho_2 = \sigma_0\gamma_2\rho_2$ and $a'\gamma_1 = a'\gamma_2$ for $a' \neq a$. We define γ by $a'\gamma = a'\gamma'$ for $a' \neq a$ and $a\gamma = a\gamma_1$. Since x is fresh and only fresh variables are introduced along π, we have for all non-abstract variables $x' \in (\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(KB)) \setminus (\mathcal{N}(Q\sigma_0) \cup \mathcal{N}(S\sigma_0) \cup \mathcal{N}(KB'))$ that $x' \notin \mathcal{N}(S_j) \cup \mathcal{N}(KB_j)$. Hence, we can define the variable renaming ρ by $x\rho = a\gamma_2$ and $x'\rho = x'\rho'$ for $x' \in (\mathcal{N}(Q) \cup \mathcal{N}(S) \cup \mathcal{N}(KB)) \setminus (\mathcal{N}(Q\sigma_0) \cup \mathcal{N}(S\sigma_0) \cup \mathcal{N}(KB'))$. Then we obviously have $S_i\gamma\rho = S_i\gamma_i\rho_i$ for all $i \in \{1, \ldots, k\}$ and $S_j\gamma\rho = S_j\gamma'\rho'$ for all $j \in \{2, \ldots, k\}$. Since γ is equally defined to γ' for all variables occurring in the knowledge bases KB_j, we clearly have that γ is a concretization w.r.t. KB_j. Moreover, as γ is equally defined to γ_1 for all variables occurring in KB_1, it is also a concretization w.r.t. KB_1.

- For all other cases we know that $\gamma_1\rho_1 = \gamma_2\rho_2$. Hence, the lemma follows by the induction hypothesis.

□

Furthermore, we show that our definition of skip values follows exactly the number of backtracked or cut state elements at the beginning of a state. For this we can already use the result of Lemma 7.13 and show this only for concrete state-derivations using the same concretization and variable renaming along a path. Moreover, we can assume that the path does not end in an empty state as we cannot have any triple or clause paths with such a path as a subpath.

Lemma 7.14 (Positive Skip Values Correspond to Backtracking or Cutting)**.** *Given a path $\pi = n_1 \dots n_k$ with $k > 1$, $n_j \notin Instance(G) \cup Generalization(G)$ for all $j \in \{1, \dots, k-1\}$, a concrete state-derivation such that there is a variable renaming ρ and a concretization γ w.r.t. $KB_1, \dots, KB_k$ where $n_i = S_i; KB_i$ for all $i \in \{1, \dots, k\}$ and the concrete state-derivation goes from a state extension of $S_1\gamma\rho$ to a state extension of $S_k\gamma\rho$ by reaching state extensions of all $S_i\gamma\rho$, and $S_k \neq \varepsilon$, then $\sigma_{\pi,0} = \sigma_{n_1 n_2, d}\sigma_{n_2 \dots n_k, 0}$ iff the concrete state-derivation backtracks or cuts the first d state elements of the state extension of $S_2\gamma\rho$ until it reaches the state extension of $S_k\gamma\rho$.*

Proof. We perform the proof by induction over the length k of π.

For $k = 2$ we have $\sigma_{\pi,0} = \sigma_{n_1 n_2, 0} = \sigma_{n_1 n_2, 0} id = \sigma_{n_1 n_2, 0}\sigma_{n_2, 0}$ and as the concrete state-derivation cannot backtrack or cut any state elements from the state extension of S_2 to the same state extension of S_2, the lemma holds. For $k > 2$ we can assume the lemma holds for paths of length at most $k - 1$.

By the induction hypothesis we obtain that $\sigma_{n_2 \dots n_k, 0} = \sigma_{n_2 n_3, d'}\sigma_{n_3 \dots n_k, 0}$ iff the concrete state-derivation backtracks or cuts d' state elements of $S_3\gamma\rho$ until it reaches the state extension of $S_k\gamma\rho$. By Definition 7.6 we also know that $\sigma_{\pi,0} = \sigma_{n_1 n_2, d}\sigma_{n_2 \dots n_k}$ for some $d \in \mathbb{N}$. We perform a case analysis over n_2 and n_3.

- If $n_2 \in NoUnifyCase(G) \cup UnequalsCase(G)$ and $n_3 = Succ(2, n_2)$, i.e., we traverse the right child of a NoUnifyCase or UnequalsCase node, we have $d = d' + 1$. Furthermore, the concrete state-derivation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in Parallel(G)$ and $n_3 = Succ(2, n_2)$, i.e., we traverse the right child of a Parallel node, we have $d = d' + j$ where $Succ(1, n_2)$ contains j state elements. Furthermore, as the concrete state-derivation reaches a state extension of $S_3\gamma\rho$ from a state extension of $S_2\gamma\rho$, it must backtrack or cut the first j state elements of of $S_2\gamma\rho$ from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in Cut(G)$ and $n_3 = Succ(1, n_2)$, i.e., we traverse a Cut node, there are two cases depending on whether $d' = 0$. If $d' = 0$, we have $d = 0$ and the concrete state-derivation does not backtrack or cut any state element before the first state element of the state extension of $S_3\gamma\rho$. Since this first state element of the state extension of $S_3\gamma\rho$ corresponds to the first state element of the state extension of

$S_2\gamma\rho$ the concrete state-derivation does not backtrack or cut any state element before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' > 0$, we have $d = d' + j$ where $n_2 = !_m, Q \mid S'_1 \mid \ldots \mid S'_j \mid ?_m \mid S'$ and $S'_i \in \mathit{StateElements} \setminus \{?_m\}\ \forall i \in \{1, \ldots, j\}$. Furthermore, the concrete state-derivation cuts exactly j state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. Since $d' > 0$ these state elements must belong to the first state elements of the state extension of $S_2\gamma\rho$ which are backtracked or cut during the concrete state-derivation. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{CutAll}(G)$ and $n_3 = \mathit{Succ}(1, n_2)$, i.e., we traverse a CutAll node, we must have $d' = 0$ since otherwise the concrete state-derivation would have backtracked or cut at least one state element of the state extension of $S_3\gamma\rho$. As S_3 contains only one state element, this is in contradiction to the condition $S_k \neq \varepsilon$. So we have $d = 0$ and the concrete state-derivation does not backtrack or cut any state element before the first state element of the state extension of $S_3\gamma\rho$. Since this first state element of the state extension of $S_3\gamma\rho$ corresponds to the first state element of the state extension of $S_2\gamma\rho$ the concrete state-derivation does not backtrack or cut any state element before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{BacktrackSecond}(G) \setminus \{\text{Parallel}\}$ and $n_3 = \mathit{Succ}(2, n_2)$, we have $d = d' + 1$. Furthermore, the concrete state-derivation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{Backtracking}(G)$ and $n_3 = \mathit{Succ}(1, n_2)$, we have $d = d' + 1$. Furthermore, the concrete state-derivation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' + 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{VarCase}(G)$ and $n_3 = \mathit{Succ}(j, n_2)$ where n_2 has j children, we have $d = d' + 1$. Furthermore, the concrete state-derivation backtracks exactly one state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension

of $S_3\gamma\rho$, it backtracks or cuts the first $d'+1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{Call}(G) \cup \mathit{Disjunction}(G) \cup \mathit{IfThen}(G) \cup \mathit{Repeat}(G)$ and $n_3 = \mathit{Succ}(1, n_2)$, i.e., we traverse a CALL, DISJUNCTION, IFTHEN or REPEAT node, we have $d = \max(0, d' - 1)$. There are two cases depending on whether $d' > 1$. If $d' > 1$, we have $d = d' - 1$. Furthermore, the concrete state-derivation introduces exactly one additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belongs to the first d' state elements of the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - 1 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq 1$, we have $d = 0$. Furthermore, the concrete state-derivation introduces exactly one additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{IfThenElse}(G) \cup \mathit{Not}(G)$ and $n_3 = \mathit{Succ}(1, n_2)$, i.e., we traverse an IFTHENELSE or NOT node, we have $d = \max(0, d' - 2)$. There are two cases depending on whether $d' > 2$. If $d' > 2$, we have $d = d' - 2$. Furthermore, the concrete state-derivation introduces exactly two additional state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belong to the first d' state elements of the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - 2 = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq 2$, we have $d = 0$. Furthermore, the concrete state-derivation introduces exactly two additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

- If $n_2 \in \mathit{Case}(G)$ and $n_3 = \mathit{Succ}(1, n_2)$, i.e., we traverse a CASE node, we have $d = \max(0, d' - j)$ where $S_2 = t, Q \mid S_r$ and $|\mathit{Slice}(\mathcal{P}, t)| = j$. There are two cases depending on whether $d' > j$. If $d' > j$, we have $d = d' - j$. Furthermore, the concrete state-derivation introduces exactly j additional state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$ which belong to the first

d' state elements of the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$, it backtracks or cuts the first $d' - j = d$ state elements of the state extension of $S_2\gamma\rho$ and the lemma holds. If otherwise $d' \leq j$, we have $d = 0$. Furthermore, the concrete state-derivation introduces exactly j additional state element from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. As the concrete state-derivation backtracks or cuts the first d' state elements of the state extension of $S_3\gamma\rho$ and, thus, backtracks or cuts at most as many state elements as introduced from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$, it backtracks or cuts no state elements before the first state element of the state extension of $S_2\gamma\rho$ and the lemma holds.

- For all other cases we have $d = d'$ and the concrete state-derivation neither backtracks or cuts nor introduces state elements from the state extension of $S_2\gamma\rho$ to the state extension of $S_3\gamma\rho$. Thus, the lemma holds.

□

Now we can show that the substitutions we use for the clause heads or intermediate goals of DTs or clauses read for triple and clause paths respectively correspond to the answer substitutions of the concrete state-derivations along the respective path.

Lemma 7.15 (Answer Substitutions are Instances of Path Substitutions)**.** *Given a path $\pi = n_1 \dots n_k$ with $n_j \notin Instance(G) \cup Generalization(G)$ for all $j \in \{1, \dots, k-1\}$ and a concrete state-derivation such that there is a variable renaming ρ and a concretization γ w.r.t. $KB_1, \dots, KB_k$ where $n_i = S_i; KB_i$ for all $i \in \{1, \dots, k\}$ and the concrete state-derivation goes from a state extension of $S_1\gamma\rho$ to a state extension of $S_k\gamma\rho$ with answer substitution δ by reaching state extensions of all $S_i\gamma\rho$, then $\sigma_{\pi,0}\gamma\rho = \gamma\rho\delta$ and $S_k\gamma\rho\delta = S_k\gamma\rho$.*

Proof. We perform the proof by induction over the length k of the path π.

For $k = 1$ we have $n_1 = n_k$ and the empty answer substitution $\delta = id = \sigma_{n_1,0}$. Hence, the lemma trivially holds.

For $k > 1$ we can assume the lemma holds for paths of length at most $k - 1$. We perform a case analysis over n_1 and n_2.

- If $n_1 \in Split(G)$ and $n_2 = Succ(2, n_1)$, i.e., we traverse the right child of a SPLIT node, we have $n_1 = t, Q; KB$ and $n_2 = Q\mu; KB'$ as defined in the SPLIT rule. By the induction hypothesis we obtain $\sigma_{n_2 \dots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $Q\mu\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution μ'

of the concrete state-derivation from a state extension of $(t, Q)\gamma\rho$ to a state extension of $Q\mu\gamma\rho$ we know by Lemma 3.51 that $\gamma\rho\mu' = \mu\gamma\rho$. Therefore, we have $\gamma\rho\delta = \gamma\rho\mu'\delta'' = \mu\gamma\rho\delta'' = \mu\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Furthermore, we know that μ is idempotent as all variables in the range of μ are fresh. As we applied μ to S_2 already and we know by inspection of the abstract inference rules other than INSTANCE and GENERALIZATION that only fresh variables are introduced along π, we obtain $S_k\mu = S_k$. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\mu'\delta'' = S_k\mu\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{Eval}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of an EVAL node, we have $n_1 = (t, Q)^c_m \mid S; KB$ and $n_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the EVAL rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $B'_c\sigma'\gamma\rho, Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution σ'' of the concrete state-derivation from a state extension of $(t, Q)^c_m\gamma\rho \mid S\gamma\rho$ to a state extension of $B'_c\sigma'\gamma\rho, Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by Lemma 3.28 that $\gamma\rho\sigma'' = \sigma'\gamma\rho$ and $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. Furthermore, we know that σ' is idempotent as the range of σ' contains only fresh variables. Now there are two cases depending on whether $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than INSTANCE and GENERALIZATION that only fresh variables are introduced along π, we obtain $S_k\sigma' = S_k$ by σ' being idempotent. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\sigma''\delta'' = S_k\sigma'\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$. In the second case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete concrete state-derivation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{OnlyEval}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse an ONLYEVAL node, we have $n_1 = (t, Q)^c_m \mid S; KB$ and $n_2 = B'_c\sigma', Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the ONLYEVAL rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $B'_c\sigma'\gamma\rho, Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution σ'' of the concrete state-derivation from a state extension of $(t, Q)^c_m\gamma\rho \mid S\gamma\rho$ to a state extension of $B'_c\sigma'\gamma\rho, Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by Lemma 3.29 that $\gamma\rho\sigma'' = \sigma'\gamma\rho$ and $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. Furthermore, we know that σ' is idempotent as the range of σ' contains only fresh variables. Now there are two cases depending on whether $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did not backtrack the

substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Instance and Generalization that only fresh variables are introduced along π, we obtain $S_k\sigma' = S_k$ by σ' being idempotent. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\sigma''\delta'' = S_k\sigma'\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$. In the second case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete concrete state-derivation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{UnifyCase}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of a UnifyCase node, we have $n_1 = =(t_1, t_2), Q \mid S; KB$ and $n_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifyCase rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution σ'' of the concrete state-derivation from a state extension of $(=(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by the soundness proof of UnifyCase that $\gamma\rho\sigma'' = \sigma'\gamma\rho$ and $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. Furthermore, we know that σ' is idempotent as the range of σ' contains only fresh variables. Now there are two cases depending on whether $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than Instance and Generalization that only fresh variables are introduced along π, we obtain $S_k\sigma' = S_k$ by σ' being idempotent. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\sigma''\delta'' = S_k\sigma'\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$. In the second case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete concrete state-derivation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{UnifySuccess}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse a UnifySuccess node, we have $n_1 = =(t_1, t_2), Q \mid S; KB$ and $n_2 = Q\sigma' \mid S\sigma|_{\mathcal{G}}; KB'$ as defined in the UnifySuccess rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. For the answer substitution σ'' of the concrete state-derivation from a state extension of $(=(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho$ we know by the soundness proof of UnifySuccess that $\gamma\rho\sigma'' = \sigma'\gamma\rho$ and $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. Furthermore,

we know that σ' is idempotent as the range of σ' contains only fresh variables. Now there are two cases depending on whether $\sigma_{\pi,0}$ starts with σ' or $\sigma|_{\mathcal{G}}$. In the first case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did not backtrack the substitution σ''. Hence, we obtain $\gamma\rho\delta = \gamma\rho\sigma''\delta'' = \sigma'\gamma\rho\delta'' = \sigma'\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Additionally, we already applied σ' to S_2. As we know by inspection of all abstract inference rules other than INSTANCE and GENERALIZATION that only fresh variables are introduced along π, we obtain $S_k\sigma' = S_k$ by σ' being idempotent. Hence, we have $S_k\gamma\rho\delta = S_k\gamma\rho\sigma''\delta'' = S_k\sigma'\gamma\rho\delta'' = S_k\gamma\rho\delta'' = S_k\gamma\rho$. In the second case we know by definition of $\sigma_{\pi,0}$ and Lemma 7.14 that the concrete state-derivation did backtrack the substitution σ'' and we have the same answer substitution δ'' for the complete concrete state-derivation. This amounts to $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{NoUnifyCase}(G)$ and $n_2 = \mathit{Succ}(2, n_1)$, i.e., we traverse the right child of a NOUNIFYCASE node, we have $n_1 = \backslash\!=\!(t_1, t_2), Q \mid S; KB$ and $n_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NOUNIFYCASE rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of NOUNIFYCASE that $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. As the answer substitution of the concrete state-derivation from a state extension of $(\backslash\!=\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{NoUnifyFail}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse a NOUNIFYFAIL node, we have $n_1 = \backslash\!=\!(t_1, t_2), Q \mid S; KB$ and $n_2 = S\sigma|_{\mathcal{G}}; KB'$ as defined in the NOUNIFYFAIL rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of NOUNIFYFAIL that $\gamma\rho = \sigma|_{\mathcal{G}}\gamma\rho$. As the answer substitution of the concrete state-derivation from a state extension of $(\backslash\!=\!(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma|_{\mathcal{G}}\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{\mathcal{G}}\gamma\rho\delta'' = \sigma_{\mathcal{G}}\sigma_{n_2\ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{EqualsCase}(G)$ and $n_2 = \mathit{Succ}(1, n_1)$, i.e., we traverse the left child of an EQUALSCASE node, we have $n_1 = ==\!(t_1, t_2), Q \mid S; KB$ and $n_2 = Q\sigma \mid S\sigma; KB'$ as defined in the EQUALSCASE rule. By the induction hypothesis we obtain $\sigma_{n_2\ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $Q\sigma\gamma\rho \mid S\sigma\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of EQUALSCASE that $\gamma\rho = \sigma\gamma\rho$.

As the answer substitution of the concrete state-derivation from a state extension of $(==(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $Q\sigma\gamma\rho \mid S\sigma\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{n_2 \ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- If $n_1 \in \mathit{UnequalsCase}(G)$ and $n_2 = \mathit{Succ}(2, n_1)$, i.e., we traverse the right child of an UnequalsCase node, we have $n_1 = \backslash==(t_1, t_2), Q \mid S; KB$ and $n_2 = S\sigma; KB'$ as defined in the UnequalsCase rule. By the induction hypothesis we obtain $\sigma_{n_2 \ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $S\sigma\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. We know by the soundness proof of UnequalsCase that $\gamma\rho = \sigma\gamma\rho$. As the answer substitution of the concrete state-derivation from a state extension of $(\backslash==(t_1, t_2), Q)\gamma\rho \mid S\gamma\rho$ to a state extension of $S\sigma\gamma\rho$ is empty, we know that $\delta = \delta''$ and, hence, $\gamma\rho\delta = \gamma\rho\delta'' = \sigma\gamma\rho\delta'' = \sigma\sigma_{n_2 \ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$. Moreover, we obtain $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

- For all other cases we know that the concrete state-derivation has the empty answer substitution from the state extension of $S_1\gamma\rho$ to the state extension of $S_2\gamma\rho$. By the induction hypothesis we obtain $\sigma_{n_2 \ldots n_k,0}\gamma\rho = \gamma\rho\delta''$ where δ'' is the answer substitution of the concrete state-derivation from a state extension of $S_2\gamma\rho$ to a state extension of $S_k\gamma\rho$ and $S_k\gamma\rho\delta'' = S_k\gamma\rho$. Then we have $\gamma\rho\delta = \gamma\rho\delta'' = \sigma_{n_2 \ldots n_k,0}\gamma\rho = \sigma_{\pi,0}\gamma\rho$ and $S_k\gamma\rho\delta = S_k\gamma\rho\delta'' = S_k\gamma\rho$.

□

Now, using the preceding results, we can simulate the evaluation of intermediate goals with the clauses from $\mathcal{P}_G$. We use the connection between concrete and abstract state-derivations by tree paths and the connection between the answer substitutions of successful evaluations and the substitutions we read off from a path in a termination graph.

Lemma 7.16 (Simulation of Intermediate Goals Using $\mathcal{P}_G$)**.** *Let* $S; KB \in \mathit{Succ}(1, \mathit{Instance}(G) \cup \mathit{Generalization}(G) \cup \mathit{Split}(G))$.

If there is a variable renaming ρ *on* $\mathcal{N}$ *and a concretization* γ *w.r.t.* KB *such that there is a concrete state-derivation from* $S\gamma\rho$ *to a state extension of* $\square$*, then we have* $\mathit{Rename}(S; KB)\gamma\rho \vdash^*_{\mathcal{P}_G} \square$.

Proof. From Lemma 7.12 we know that there is a tree path $\pi_{\mathit{tree}} = (0, v_0, \mathit{none}), (1, v_1, p_1), \ldots, (k, v_k, p_k)$ with the following properties:

- $v_0 = S; KB$
- for all $i \in \{0, \ldots, k\}$ there are concretizations γ_i and variable renamings ρ_i on $\mathcal{N}$ such that the concrete state-derivation reaches a scope variant of a state extension of $S_i\gamma_i\rho_i$ in $l_i \leq l$ steps where $v_i = S_i; KB_i$ and $S_i\gamma_i \in \mathcal{CON}(S_i; KB_i)$
- for all (i, v_i, p_i) with more than one successor in π_{tree}, we have $v_i \in \mathit{Split}(G)$
- for all (i, v_i, p_i) with $v_i \in \mathit{Split}(G)$ and only one successor (j, v_j, i) in π_{tree}, we have $v_j = \mathit{Succ}(1, v_i)$
- for all leaves (i, v_i, p_i) of π_{tree} we have $v_i \in \mathit{Success}(G)$

The last property follows from the fact that the concrete state-derivation reaches a state extension of $\square$.

We perform the proof by induction over the length k of the tree path π_{tree} which is at least 1.

For $k = 1$ we have $\pi_{tree} = (0, S; KB, \mathit{none})$ and $S; KB \in \mathit{Success}(G)$. Thus, we have $\mathit{Rename}(S; KB) = \square \vdash^*_{\mathcal{P}_G} \square$. For $k > 1$ we can assume the lemma holds for tree paths of length at most $k - 1$.

Consider the rightmost path π through the tree path π_{tree}. As the concrete state-derivation reaches a state extension of $\square$, π must be a sequence of clause paths w.r.t. G. If this sequence has the length 1, we obtain $\mathit{Rename}(S; KB)\sigma_{\pi,0} \leftarrow I_\pi \in \mathcal{P}_G$ and by Lemma 7.13 we can w.l.o.g. assume that all concretizations and variable renamings used in π are equal to γ and ρ. By Lemma 7.15 we know that $\mathit{Rename}(S; KB)\sigma_{\pi,0}$ unifies with $\mathit{Rename}(S; KB)\gamma\rho$ by $\gamma\rho\delta$ where δ is the answer substitution of the concrete state-derivation from $S\gamma\rho$ to a state extension of $\square$. Thus, we obtain $\mathit{Rename}(S; KB)\gamma\rho \vdash^*_{\mathcal{P}_G} I_\pi\gamma\rho\delta$. As we have for all intermediate goals in I_π some subtree paths in π such that the concrete state-derivation reaches a state extension of $\square$ from the respective intermediate goal in $I_\pi\gamma\rho$, we can use the induction hypothesis and obtain $I_\pi\gamma\rho \vdash^*_{\mathcal{P}_G} \square$. Now, since δ is the answer substitution for the complete concrete state-derivation, we obtain $I_\pi\gamma\rho\delta \vdash^*_{\mathcal{P}_G} \square$ by Lemma 7.15. Now let the length of the sequence be greater than 1. For the first clause path in the sequence to a node v_i we obtain $\mathit{Rename}(S; KB)\gamma\rho \vdash^*_{\mathcal{P}_G} \mathit{Rename}(v_i)\gamma\rho$ by the identical argument as for the sequence of length 1. The remaining tree path from (i, v_i, p_i) is shorter than k and, therefore, we obtain $\mathit{Rename}(v_i)\gamma\rho \vdash^*_{\mathcal{P}_G} \square$ by the induction hypothesis. $\square$

As we can simulate intermediate goals with $\mathcal{P}_G$, we can simulate the evaluation along triple paths with the DTs in $\mathit{DTP}(G)$.

Lemma 7.17 (Simulation of Concrete State-Derivations Using $DTP(G)$)**.** *Given a concrete state-derivation and a corresponding tree path π_{tree} w.r.t. G with the properties from Lemma 7.12 and a rightmost path $\pi = n_0 \dots n_k$ in π_{tree} where for $i < k$, $n_i \notin Instance(G) \cup Generalization(G)$, we have $Rename_{Triple}(n_0)\gamma_0\rho_0 \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_k)\gamma_k\rho_k$ where the γ_j and ρ_j are the concretizations and variable renamings respectively used in π_{tree}.*

Proof. We perform the proof by induction over k, which is the length of π minus one.

For $k = 0$ we have $n_0 = n_k$ and, therefore, $Triple(\pi) = Rename_{Triple}(n_0) \leftarrow Rename_{Triple}(n_0)$. Thus, obviously we have $Rename_{Triple}(n_0)\gamma_0\rho_0 \vdash_{Triple(\pi)} Rename_{Triple}(n_0)\gamma_0\rho_0$. For $k > 0$, we can assume the lemma holds for paths of length at most k. By Lemma 7.13 we can w.l.o.g. assume that $\gamma_0 = \dots = \gamma_k = \gamma$ and $\rho_0 = \dots = \rho_k = \rho$.

We now perform a case analysis based on n_k and n_{k-1}.

If $n_{k-1} \in Split(G)$ and $n_k = Succ(2, n_{k-1})$, i.e., we traverse the right child of a SPLIT node, we know by the induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \dots n_{k-1}) \circ \vdash^*_{\mathcal{P}_G}} Rename_{Triple}(n_{k-1})\gamma\rho$. From the Lemma 3.51 we know that $\gamma\rho\mu' = \mu\gamma\rho$ where $Rename(Succ(1, n_{k-1}))\gamma\rho \vdash^*_{\mathcal{P}_G} \square$ with answer substitution μ'. The latter follows from Lemma 7.16 and the fact that the concrete state-derivation must reach a state extension of $\square$ from $Succ(1, n_{k-1})\gamma\rho$. By Lemma 7.15, we also know that $Rename_{Triple}(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $Rename_{Triple}(n_k)$ completely. Furthermore, we obtain the same derivation for $Rename(Succ(1, n_{k-1}))\gamma\rho\delta$ as for $Rename(Succ(1, n_{k-1}))\gamma\rho$, but with the empty answer substitution by Lemma 7.15. Thus, we obtain:

$$
\begin{array}{rl}
 & Rename_{Triple}(n_0)\gamma\rho \\
\vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} & Rename(Succ(1, n_{k-1}))\gamma\rho\delta, Rename_{Triple}(n_k)\mu^{-1}\gamma\rho\mu' \\
\vdash^*_{\mathcal{P}_G} & Rename_{Triple}(n_k)\mu^{-1}\gamma\rho\mu' \\
= & Rename_{Triple}(n_k)\mu^{-1}\mu\gamma\rho \\
= & Rename_{Triple}(n_k)\gamma\rho
\end{array}
$$

If $n_{k-1} \in Eval(G)$ and $n_k = Succ(1, n_{k-1})$, i.e., we traverse the left child of an EVAL node, we know by the induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \dots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{k-1})\gamma\rho$. From Lemma 3.28 we know that $Q\gamma\rho\sigma'' = Q\sigma'\gamma\rho$ and $S\gamma\rho = S\sigma|_{\mathcal{G}}\gamma\rho$. By Lemma 7.15, we also know that $Rename_{Triple}(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $Rename_{Triple}(n_k)$ completely. Let

$n_{k-1} = (t, Q)^i_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & \mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \\ \vdash_{\mathit{Triple}(\pi)} \circ \vdash^*_{\mathcal{P}_G} & \mathit{Rename}_{\mathit{Triple}}(Q\gamma\rho\sigma'' \mid S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & \mathit{Rename}_{\mathit{Triple}}(Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})) \\ = & \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \mathit{OnlyEval}(G)$, we have the same case as for $n_{k-1} \in \mathit{Eval}(G)$ and $n_k = \mathit{Succ}(1, n_{k-1})$.

If $n_{k-1} \in \mathit{UnifyCase}(G)$ and $n_k = \mathit{Succ}(1, n_{k-1})$, i.e., we traverse the left child of an UnifyCase node, we know by the induction hypothesis that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \vdash_{\mathit{Triple}(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} \mathit{Rename}_{\mathit{Triple}}(n_{k-1})\gamma\rho$. From the proof of UnifyCase we know that $Q\gamma\rho\sigma'' = Q\sigma'\gamma\rho$ and $S\gamma\rho = S\sigma|_{\mathcal{G}}\gamma\rho$. By Lemma 7.15, we also know that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho$ unifies with the head of $\mathit{Triple}(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $\mathit{Rename}_{\mathit{Triple}}(n_k)$ completely. Let $n_{k-1} = (=(t_1, t_2), Q)^b_m \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & \mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \\ \vdash_{\mathit{Triple}(\pi)} \circ \vdash^*_{\mathcal{P}_G} & \mathit{Rename}_{\mathit{Triple}}(Q\gamma\rho\sigma'' \mid S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & \mathit{Rename}_{\mathit{Triple}}(Q\sigma'\gamma\rho \mid S\sigma|_{\mathcal{G}}\gamma\rho; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})) \\ = & \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \mathit{UnifySuccess}(G)$, we have the same case as for $n_{k-1} \in \mathit{UnifyCase}(G)$ and $n_k = \mathit{Succ}(1, n_{k-1})$.

If $n_{k-1} \in \mathit{EqualsCase}(G)$ and $n_k = \mathit{Succ}(1, n_{k-1})$, i.e., we traverse the left child of an EqualsCase node, we know by the induction hypothesis that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \vdash_{\mathit{Triple}(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} \mathit{Rename}_{\mathit{Triple}}(n_{k-1})\gamma\rho$. From the proof of EqualsCase we know that $\gamma\rho = \sigma\gamma\rho$. By Lemma 7.15, we also know that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho$ unifies with the head of $\mathit{Triple}(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $\mathit{Rename}_{\mathit{Triple}}(n_k)$ completely. Let $n_{k-1} = ==(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & \mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \\ \vdash_{\mathit{Triple}(\pi)} \circ \vdash^*_{\mathcal{P}_G} & \mathit{Rename}_{\mathit{Triple}}(Q\gamma\rho \mid S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & \mathit{Rename}_{\mathit{Triple}}(Q\sigma\gamma\rho \mid S\sigma\gamma\rho; (\mathcal{G}', \mathcal{F}, \mathcal{U}')) \\ = & \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \mathit{UnequalsCase}(G)$ and $n_k = \mathit{Succ}(2, n_{k-1})$, i.e., we traverse the right child of

an UnequalsCase node, we know by the induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{k-1})\gamma\rho$. From the proof of UnequalsCase we know that $\gamma\rho = \sigma\gamma\rho$. By Lemma 7.15, we also know that $Rename_{Triple}(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $Rename_{Triple}(n_k)$ completely. Let $n_{k-1} = \backslash{=}{=}(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & Rename_{Triple}(n_0)\gamma\rho \\ \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} & Rename_{Triple}(S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & Rename_{Triple}(S\sigma\gamma\rho; (\mathcal{G}', \mathcal{F}, \mathcal{U}')) \\ = & Rename_{Triple}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in NoUnifyCase(G)$ and $n_k = Succ(2, n_{k-1})$, i.e., we traverse the right child of a NoUnifyCase node, we know by the induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{k-1})\gamma\rho$. From the proof of NoUnifyCase we know that $S\gamma\rho = S\sigma|_{\mathcal{G}}\gamma\rho$. By Lemma 7.15, we also know that $Rename_{Triple}(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $Rename_{Triple}(n_k)$ completely. Let $n_{k-1} = \backslash{=}(t_1, t_2), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & Rename_{Triple}(n_0)\gamma\rho \\ \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} & Rename_{Triple}(S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & Rename_{Triple}(S\sigma|_{\mathcal{G}}\gamma\rho; (\mathcal{G}', \mathcal{F}', \mathcal{U}\sigma|_{\mathcal{G}})) \\ = & Rename_{Triple}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in NoUnifyFail(G)$, we have the same case as for $n_{k-1} \in NoUnifyCase(G)$ and $n_k = Succ(2, n_{k-1})$.

If $n_{k-1} \in VarCase(G)$, n_{k-1} has l children, $1 < i < l$ and $n_k = Succ(i, n_{k-1})$, i.e., we traverse a child of a VarCase node where we consider the case of an already existing variable, we know by the induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{k-1})\gamma\rho$. From the proof of VarCase we know that $\gamma\rho = \sigma_i\gamma\rho$. By Lemma 7.15, we also know that $Rename_{Triple}(n_0)\gamma\rho$ unifies with the head of $Triple(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $Rename_{Triple}(n_k)$ completely. Let $n_{k-1} = \mathsf{var}(a), Q \mid$

$S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & \mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \\ \vdash_{\mathit{Triple}(\pi)} \circ \vdash^*_{\mathcal{P}_G} & \mathit{Rename}_{\mathit{Triple}}(Q\gamma\rho \mid S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & \mathit{Rename}_{\mathit{Triple}}(Q\sigma_i\gamma\rho \mid S\sigma_i\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_i)) \\ = & \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \mathit{VarCase}(G)$ and $n_k = \mathit{Succ}(1, n_{k-1})$, i.e., we traverse a child of a VarCase node where we consider the case of a variable not already existing in the state, we know by the induction hypothesis that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \vdash_{\mathit{Triple}(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} \mathit{Rename}_{\mathit{Triple}}(n_{k-1})\gamma\rho$. From the proof of VarCase and Lemma 7.13 we know that $\sigma_0\gamma\rho = \gamma\rho$. By Lemma 7.15, we also know that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho$ unifies with the head of $\mathit{Triple}(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $\mathit{Rename}_{\mathit{Triple}}(n_k)$ completely. Let $n_{k-1} = \mathsf{var}(a), Q \mid S; (\mathcal{G}, \mathcal{F}, \mathcal{U})$. Then, we obtain:

$$\begin{array}{rl} & \mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \\ \vdash_{\mathit{Triple}(\pi)} \circ \vdash^*_{\mathcal{P}_G} & \mathit{Rename}_{\mathit{Triple}}(Q\gamma\rho \mid S\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U})) \\ = & \mathit{Rename}_{\mathit{Triple}}(Q\sigma_0\gamma\rho \mid S\sigma_0\gamma\rho; (\mathcal{G}, \mathcal{F}, \mathcal{U}\sigma_0)) \\ = & \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho \end{array}$$

If $n_{k-1} \in \mathit{BacktrackSecond}(G)$ and $n_k = \mathit{Succ}(2, n_{k-1})$ or $n_{k-1} \in \mathit{Backtracking}(G)$, i.e., we backtrack by removing the first element of the backtracking list, we know by Lemma 7.15 that $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho$ unifies with the head of $\mathit{Triple}(\pi)$ by $\gamma\rho\delta$ where δ is the answer substitution for the complete concrete state-derivation and that we do not have to apply δ to $\mathit{Rename}_{\mathit{Triple}}(n_k)$ completely. We perform a case analysis based on the existence of a node from the set $\mathit{Introducing}(G)$. If such a node does not exist in our path, there is also no Split node in our path. Thus, $I_\pi = \Box$. Furthermore, we have that $\sigma_{\pi,0}$ can only contain backtrack substitutions or substitutions introduced by the EqualsCase rule. To see this, remember that the only rules which reduce the skip value for $\sigma_{\pi,0}$ are from the set $\mathit{Introducing}(G)$. As we rise this value from n_k to n_{k-1} at least by one, we cannot use any other substitutions than backtrack substitutions, substitutions introduced by the EqualsCase rule or substitutions generalizing answer substitutions for Split nodes. Since we do not have the latter on the path π, we only have substitutions of the first two kinds. Both of these kinds of substitutions must correspond to the concretization γ as the concrete state-derivation reaches state extensions of all concretized states along π_{tree}. Thus, we obtain $\mathit{Rename}_{\mathit{Triple}}(n_0)\gamma\rho \vdash_{\mathit{Triple}(\pi)} \mathit{Rename}_{\mathit{Triple}}(n_k)\gamma\rho$. If there is such a node in our path, i.e., there is a j such that $n_j \in \mathit{Introducing}(G)$ and on the remaining path $\pi' = n_j \ldots n_k$ we have $n_i \notin \mathit{Introducing}(G)$ for all $i \in \{j, \ldots, k\}$, we know by the

induction hypothesis that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_j)} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_j)\gamma\rho$. For the remaining path π' we have $I_{\pi'} = \square$ and $\sigma_{\pi',0}$ consisting of backtrack substitutions and substitutions introduced by the EqualsCase rule only by the identical argument as before. Thus, we obtain $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_k)\gamma\rho$.

Finally, if n_{k-1} is in none of the above sets, we have $\sigma_\pi = \sigma_{n_1 \ldots n_{k-1}}$ and $I_\pi = I_{n_1 \ldots n_{k-1}}$. Again, from the induction hypothesis we know that $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(n_1 \ldots n_{k-1})} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{k-1})\gamma\rho$. From the definition of the abstract rules used, we know that $\mathcal{V}(s_k) \subseteq \mathcal{V}(s_{k-1})$ as they do not apply any substitutions and, thus, $Rename_{Triple}(n_0)\gamma\rho \vdash_{Triple(\pi)} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_k)\gamma\rho$. □

We now state the central theorem of this section where we prove that termination of the represented DT problem implies termination of the original Prolog program w.r.t. the specified set of queries represented by the root state of the termination graph for the Prolog program.

Theorem 7.18 (Correctness)**.** *If G is a termination graph for a Prolog program $\mathcal{P}$ such that $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ is terminating then all concretizations of $root(G)$ have only finite concrete state-derivations w.r.t. the rules of Definition 5.1.*

Proof. Assume $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ is terminating, but there is a concretization $S\gamma \in \mathcal{CON}(S; KB)$ from $root(G) = n_0 = S; KB$ that has an infinite concrete state-derivation. Then, according to Lemma 7.12 there is an infinite tree path π_{tree} where the rightmost path $\pi = n_0, n_1, n_2, \ldots$ in π_{tree} is an infinite sequence of triple paths $\pi_0, \pi_1, \pi_2, \ldots$ and there are indices $l_0, l_1, l_2, \ldots$ such that $\pi_m = n_{l_m}, \ldots$.

Now, according to Lemma 7.17 we know that for all $m \in \mathbb{N}$ there are concretizations γ_m and variable renamings ρ_m such that $Rename_{Triple}(n_{l_m})\gamma_{l_m}\rho_m \vdash_{Triple(\pi_m)} \circ \vdash^*_{\mathcal{P}_G} Rename_{Triple}(n_{l_{m+1}})\gamma_{l_{m+1}}\rho_{m+1}$.

Thus, $Rename_{Triple}(n_{l_0})\gamma_{l_0}\rho_0 \in \mathcal{C}_G$ starts an infinite $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$-chain
$(Rename_{Triple}(n_{l_0})\sigma_{\pi_0,\varnothing} \leftarrow I_{\pi_0}, Rename_{Triple}(n_{l_1}))$,
$(Rename_{Triple}(n_{l_1})\sigma_{\pi_1,\varnothing} \leftarrow I_{\pi_1}, Rename_{Triple}(n_{l_2}))$,
...
which contradicts our initial assumption and, hence, proves the theorem. □

Corollary 7.19 (Termination Analysis of Prolog)**.** *A Prolog program $\mathcal{P}$ is terminating w.r.t. a class of queries $\mathcal{Q}$ described by a symbol $p \in \Sigma$ and a moding function $m : \Sigma \times \mathbb{N} \to \{\boldsymbol{in}, \boldsymbol{out}\}$ if G is a termination graph for the initial node $S(p, m)$ and $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ is terminating.*

Still, as for the approach in [Sch08], the reverse direction of this corollary does not hold as our method is not termination-preserving. There are two reasons for that. First, we may have lost too much precision during the construction of the termination graph and obtain non-terminating cycles. Second, the represented DT problem may also lose

precision compared to the termination graph. We illustrate these issues with the following two examples.

Example 7.20. Consider again the first termination graph from Example 6.9 for the Prolog program $\mathcal{P}$ which is terminating w.r.t. the query set $\mathcal{Q}$ from the same example. We obtain the following DTs for this termination graph.

$$\begin{aligned} \mathsf{q}_1(T_3) &\leftarrow \mathsf{p}_{c4}(T_3), \mathsf{r}_5(T_3). \\ \mathsf{r}_5(T_6) &\leftarrow \mathsf{r}_5(T_6). \end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{r}_{c5}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) &\leftarrow \square. \\ \mathsf{r}_{c5}(T_6) &\leftarrow \mathsf{r}_{c5}(T_6). \\ \mathsf{p}_{c4}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) &\leftarrow \square. \end{aligned}$$

As we assume that the call set contains all possible goals for the renamed states, this DT problem is not terminating.

Note that we can obtain a terminating DT problem for $\mathcal{P}$ and $\mathcal{Q}$ by constructing an alternative termination graph like the second one from Example 6.9.

Especially the following example demonstrates that the analysis of existential termination for Prolog programs is hard for our approach despite the fact that our method is in principle capable of analyzing existential termination. In particular, the constructed termination graph in the example contains all relevant information, but we need a stronger transformation to use it.

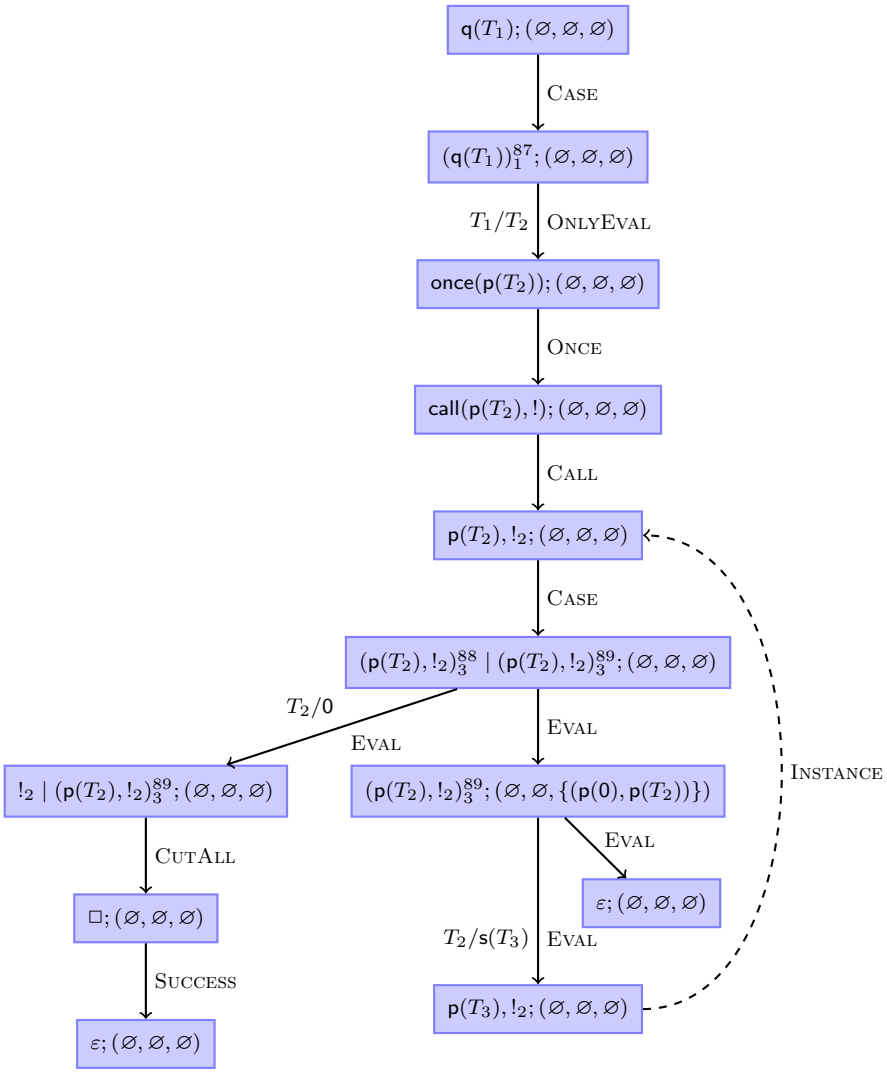
Example 7.21. Consider the following Prolog program $\mathcal{P}$

$$\mathsf{q}(X) \leftarrow \mathsf{once}(\mathsf{p}(X)). \tag{87}$$

$$\mathsf{p}(\mathsf{0}) \leftarrow \square. \tag{88}$$

$$\mathsf{p}(\mathsf{s}(X)) \leftarrow \mathsf{p}(X). \tag{89}$$

and query set $\mathcal{Q} = \{\mathsf{q}(t) \mid \mathsf{q}(t) \in \mathit{PrologTerms}(\Sigma, \mathcal{N})\}$. $\mathcal{P}$ is terminating w.r.t. $\mathcal{Q}$. Using the standard heuristic with the same parameters as in Example 6.3 except for *MinExSteps* where we use a value of 2, we obtain the following termination graph.



This termination graph represents the following DTs.

$$\begin{aligned} \mathsf{q}_1(T_2) &\leftarrow \mathsf{p}_5(T_2). \\ \mathsf{p}_5(\mathsf{s}(T_3)) &\leftarrow \mathsf{p}_5(T_3). \end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{p}_{c5}(0) &\leftarrow \Box. \\ \mathsf{p}_{c5}(\mathsf{s}(T_3)) &\leftarrow \mathsf{p}_{c5}(T_3). \end{aligned}$$

According to our assumptions concerning the call set, this DT problem is not terminating. The reason for the lost precision is that we lose the context of the trailing cut by our renaming which only considers the variables occurring in a state.

7.3 Example Transformations

We now give some examples for our transformation using Prolog programs from this thesis or the TPDB.

Example 7.22. Consider the termination graph from Example 3.39. We obtain the following DTs for this termination graph.

$$\begin{aligned} \mathsf{even}_1(T_2) &\leftarrow \mathsf{c}_5(T_2). \\ \mathsf{c}_5(\mathsf{s}(\mathsf{s}(T_4))) &\leftarrow \mathsf{c}_5(T_4). \end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{c}_{c5}(\mathsf{0}) &\leftarrow \Box. \\ \mathsf{c}_{c5}(\mathsf{s}(\mathsf{s}(T_4))) &\leftarrow \mathsf{c}_{c5}(T_4). \end{aligned}$$

Example 7.23. Consider the termination graph from Example 3.42. We obtain the following DTs for this termination graph.

$$\begin{aligned} \mathsf{q}_1(\mathsf{s}(\mathsf{s}(T_4))) &\leftarrow \mathsf{p}_{18}(T_4, \mathsf{s}(\mathsf{0})). \\ \mathsf{p}_{18}(\mathsf{s}(T_7), T_8) &\leftarrow \mathsf{p}_{18}(T_7, \mathsf{s}(T_8)). \end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{p}_{c18}(\mathsf{0}, T_6) &\leftarrow \Box. \\ \mathsf{p}_{c18}(\mathsf{s}(T_7), T_8) &\leftarrow \mathsf{p}_{c18}(T_7, \mathsf{s}(T_8)). \end{aligned}$$

Example 7.24. Consider the termination graph from Example 3.47. We obtain the following DT for this termination graph.

$$\mathsf{p}_1(\mathsf{s}(T_2)) \leftarrow \mathsf{p}_1(T_2).$$

Moreover, we obtain the following (equal) clause.

$$\mathsf{p}_{c1}(\mathsf{s}(T_2)) \leftarrow \mathsf{p}_{c1}(T_2).$$

Example 7.25. Consider the termination graph from Example 3.48. We obtain the following DT for this termination graph.

$$\mathsf{p}_1(\mathsf{s}(T_2)) \quad \leftarrow \quad \mathsf{p}_1(T_2).$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{p}_{c1}(T_3) &\leftarrow \Box. \\ \mathsf{p}_{c1}(\mathsf{s}(T_2)) &\leftarrow \mathsf{p}_{c1}(T_2). \end{aligned}$$

Example 7.26. Consider the termination graph from Example 3.52. We obtain the following DT for this termination graph.

$$\mathsf{p}_1(\mathsf{s}(T_2)) \quad \leftarrow \quad \mathsf{p}_1(T_2).$$

Moreover, we obtain the following clause.

$$\mathsf{p}_{c1}(\mathsf{s}(T_2)) \quad \leftarrow \quad \mathsf{p}_{c1}(T_2).$$

Note that we obtain this clause for the clause path to the SUCCESS node while we obtain no clause for the left successor of the SPLIT node since this node is an INSTANCE node itself.

Example 7.27. Consider the second termination graph from Example 6.9. We obtain no DTs for this termination graph and no clauses. Hence, the represented DT problem trivially terminates. For acyclic termination graphs G, we always obtain DT problems $(\mathcal{D}_G, \mathcal{C}_G, \mathcal{P}_G)$ where $\mathcal{D}_G$ is empty. Thus, we do not need to detect that a termination graph is acyclic to obtain a trivial termination proof.

Example 7.28. Consider the two termination graphs from Example 6.10. For the first graph, we obtain the following DT.

$$\mathsf{p}_1(T_{10}, T_{10}, \mathsf{a}) \quad \leftarrow \quad \mathsf{p}_1(T_{10}, T_{10}, \mathsf{a}).$$

Moreover, we obtain the following clauses.

$$\begin{aligned} \mathsf{p}_{c1}(T_4, T_4, \mathsf{a}) &\leftarrow \Box. \\ \mathsf{p}_{c1}(T_{10}, T_{10}, \mathsf{a}) &\leftarrow \mathsf{p}_{c1}(T_{10}, T_{10}, \mathsf{a}). \end{aligned}$$

As mentioned in Example 6.10, this DT problem is obviously not terminating. However, for the second termination graph we obtain no DTs again and can, thus, trivially prove termination of the original Prolog program.

Example 7.29. Consider the termination graph from Example 6.12. We obtain the following DTs for this termination graph.

$$\begin{aligned}
\mathsf{q}_1 &\leftarrow \mathsf{p}_4(\mathsf{s}(\mathsf{s}(0))).\\
\mathsf{p}_4(\mathsf{s}(\mathsf{s}(0))) &\leftarrow \mathsf{p}_8(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))).\\
\mathsf{p}_4(T_2) &\leftarrow \mathsf{p}_8(T_2).\\
\mathsf{p}_8(\mathsf{s}(T_3)) &\leftarrow \mathsf{p}_8(T_3).\\
\mathsf{p}_8(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) &\leftarrow \mathsf{p}_8(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))).
\end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned}
\mathsf{p}_{c4}(\mathsf{s}(\mathsf{s}(0))) &\leftarrow \mathsf{p}_{c8}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))).\\
\mathsf{p}_{c4}(T_2) &\leftarrow \mathsf{p}_{c8}(T_2).\\
\mathsf{p}_{c8}(0) &\leftarrow \square.\\
\mathsf{p}_{c8}(\mathsf{s}(T_3)) &\leftarrow \mathsf{p}_{c8}(T_3).\\
\mathsf{p}_{c8}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) &\leftarrow \mathsf{p}_{c8}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))).
\end{aligned}$$

As mentioned in Example 6.12, this DT problem is obviously not terminating according to our assumptions for the call set as we reach the last DT which is already not terminating alone.

Example 7.30. Consider the two termination graphs from Example 6.15. For the first graph, we obtain the following DTs.

$$\begin{aligned}
\mathsf{p}_1 &\leftarrow \mathsf{q}_4.\\
\mathsf{q}_4 &\leftarrow \mathsf{q}_4.
\end{aligned}$$

Moreover, we obtain the following clauses.

$$\begin{aligned}
\mathsf{q}_{c4} &\leftarrow \square.\\
\mathsf{q}_{c4} &\leftarrow \mathsf{q}_{c4}.
\end{aligned}$$

As mentioned in Example 6.15, this DT problem is obviously not terminating. However, for the second termination graph we obtain no DTs again and can, thus, trivially prove termination of the original Prolog program.

Example 7.31. To show that the transformation into DT problems is in fact more powerful than the pre-processing for Prolog programs, we consider the following Prolog program giesl97.pl from the TPDB

$$\begin{aligned} \mathsf{f}(\mathsf{0}, Y, \mathsf{0}) &\leftarrow \Box. \\ \mathsf{f}(\mathsf{s}(X), Y, Z) &\leftarrow \mathsf{f}(X, Y, U), \mathsf{f}(U, Y, Z). \end{aligned}$$

and the set of queries $\mathcal{Q} = \{\mathsf{f}(t_1, t_2, t_3) \mid t_1 \text{ is ground}\}$. This program terminates w.r.t. $\mathcal{Q}$.

Although the program is very small, the termination graph we construct for it using the standard heuristic with the same parameters as in Example 7.21 is already quite complex. Thus, we omit the graph for this example.

Using the pre-processing from [Sch08], we obtain the following Prolog program $\mathcal{P}$

$$\begin{aligned} \mathsf{f}_1(\mathsf{0}, T_5, \mathsf{0}) &\leftarrow \Box. \\ \mathsf{f}_1(\mathsf{s}(T_9), T_{12}, T_{13}) &\leftarrow \mathsf{p}_7(T_9, T_{12}, X_{13}, T_{13}). \\ \mathsf{p}_7(\mathsf{0}, T_{27}, \mathsf{0}, \mathsf{0}) &\leftarrow \Box. \\ \mathsf{p}_7(\mathsf{s}(T_{32}), T_{34}, X_{47}, T_{35}) &\leftarrow \mathsf{f}_{23}(T_{32}, T_{34}, X_{46}). \\ \mathsf{p}_7(\mathsf{s}(T_{32}), T_{39}, X_{47}, T_{40}) &\leftarrow \mathsf{f}_{23}(T_{32}, T_{39}, T_{38}), \mathsf{p}_7(T_{38}, T_{39}, X_{47}, T_{40}). \\ \mathsf{f}_{23}(\mathsf{0}, T_{47}, \mathsf{0}) &\leftarrow \Box. \\ \mathsf{f}_{23}(\mathsf{s}(T_{52}), T_{54}, X_{74}) &\leftarrow \mathsf{f}_{23}(T_{52}, T_{54}, X_{73}). \\ \mathsf{f}_{23}(\mathsf{s}(T_{52}), T_{58}, X_{74}) &\leftarrow \mathsf{f}_{23}(T_{52}, T_{58}, T_{57}), \mathsf{f}_{23}(T_{57}, T_{58}, X_{74}). \end{aligned}$$

and the set of queries $\mathcal{Q}' = \{\mathsf{f}_1(t_1, t_2, t_3) \mid t_1 \text{ is ground}\}$. $\mathcal{P}$ is not terminating w.r.t. $\mathcal{Q}'$. To see this, consider the query $\mathsf{f}_1(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))), Y, Z)$. This query can have the following derivation: $\mathsf{f}_1(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0})))), Y, Z) \vdash \mathsf{p}_7(\mathsf{s}(\mathsf{s}(\mathsf{0}))), Y, X_{13}, Z) \vdash \mathsf{f}_{23}(\mathsf{s}(\mathsf{s}(\mathsf{0})), Y, X_{46}) \vdash \mathsf{f}_{23}(\mathsf{s}(\mathsf{0}), Y, T_{57}), \mathsf{f}_{23}(T_{57}, Y, X_{46}) \vdash \mathsf{f}_{23}(\mathsf{0}, Y, X_{73}), \mathsf{f}_{23}(T_{57}, Y, X_{46}) \vdash \mathsf{f}_{23}(T_{57}, Y, X_{46})$. Now, this last query does not terminate universally.

However, using our transformation into DT problems, we obtain the DTs

$$\begin{aligned} \mathsf{f}_1(\mathsf{s}(T_9), T_{12}, T_{13}) &\leftarrow \mathsf{p}_7(T_9, T_{12}, X_{13}, T_{13}). \\ \mathsf{p}_7(\mathsf{s}(T_{32}), T_{34}, X_{47}, T_{35}) &\leftarrow \mathsf{f}_{23}(T_{32}, T_{34}, X_{46}). \\ \mathsf{p}_7(\mathsf{s}(T_{32}), T_{39}, X_{47}, T_{40}) &\leftarrow \mathsf{f}_{c23}(T_{32}, T_{39}, T_{38}), \mathsf{p}_7(T_{38}, T_{39}, X_{47}, T_{40}). \\ \mathsf{f}_{23}(\mathsf{s}(T_{52}), T_{54}, X_{74}) &\leftarrow \mathsf{f}_{23}(T_{52}, T_{54}, X_{73}). \\ \mathsf{f}_{23}(\mathsf{s}(T_{52}), T_{58}, X_{74}) &\leftarrow \mathsf{f}_{c23}(T_{52}, T_{58}, T_{57}), \mathsf{f}_{23}(T_{57}, T_{58}, X_{74}). \end{aligned}$$

and the following clauses.

$$\begin{aligned}
\mathsf{q}_{c7}(\mathsf{0}, T_{27}, \mathsf{0}, \mathsf{0}) &\leftarrow \Box. \\
\mathsf{q}_{c7}(\mathsf{s}(T_{32}), T_{39}, X_{47}, T_{40}) &\leftarrow \mathsf{f}_{c23}(T_{32}, T_{39}, T_{38}), \mathsf{q}_{c7}(T_{38}, T_{39}, X_{47}, T_{40}). \\
\mathsf{f}_{c23}(\mathsf{0}, T_{47}, \mathsf{0}) &\leftarrow \Box. \\
\mathsf{f}_{c23}(\mathsf{s}(T_{52}), T_{58}, X_{74}) &\leftarrow \mathsf{f}_{c23}(T_{52}, T_{58}, T_{57}), \mathsf{f}_{c23}(T_{57}, T_{58}, X_{74}).
\end{aligned}$$

This DT problem is terminating according to our assumptions regarding the call set and our fully automated termination prover AProVE manages to prove this.

Example 7.32. Finally, we demonstrate the complete transformation for the Prolog program computing addition of natural numbers from Example 4.11.

Using the standard heuristic with the same parameters as for Example 6.3, we obtain the termination graph depicted on the next page. In this graph, we use the following knowledge bases.

$$\begin{aligned}
KB_1 &= (\{T_8\}, \{P, X\}, \{(\mathsf{add}(X, \mathsf{0}, X), \mathsf{add}(T_1, T_8, T_3)) \\
KB_2 &= (\{T_8\}, \{P, X\}, \{(\mathsf{add}(X, \mathsf{0}, X), \mathsf{add}(T_1, T_8, T_3)), (\mathsf{isZero}(\mathsf{0}), \mathsf{isZero}(T_8))\}) \\
KB_3 &= (\{T_8\}, \{P, X\}, \{(\mathsf{add}(X, \mathsf{0}, X), \mathsf{add}(T_1, T_8, T_3)), (\mathsf{isZero}(\mathsf{0}), \mathsf{isZero}(T_8)), \\
&\quad\ (\mathsf{p}(\mathsf{0}, \mathsf{0}), \mathsf{p}(T_8, P))\})
\end{aligned}$$

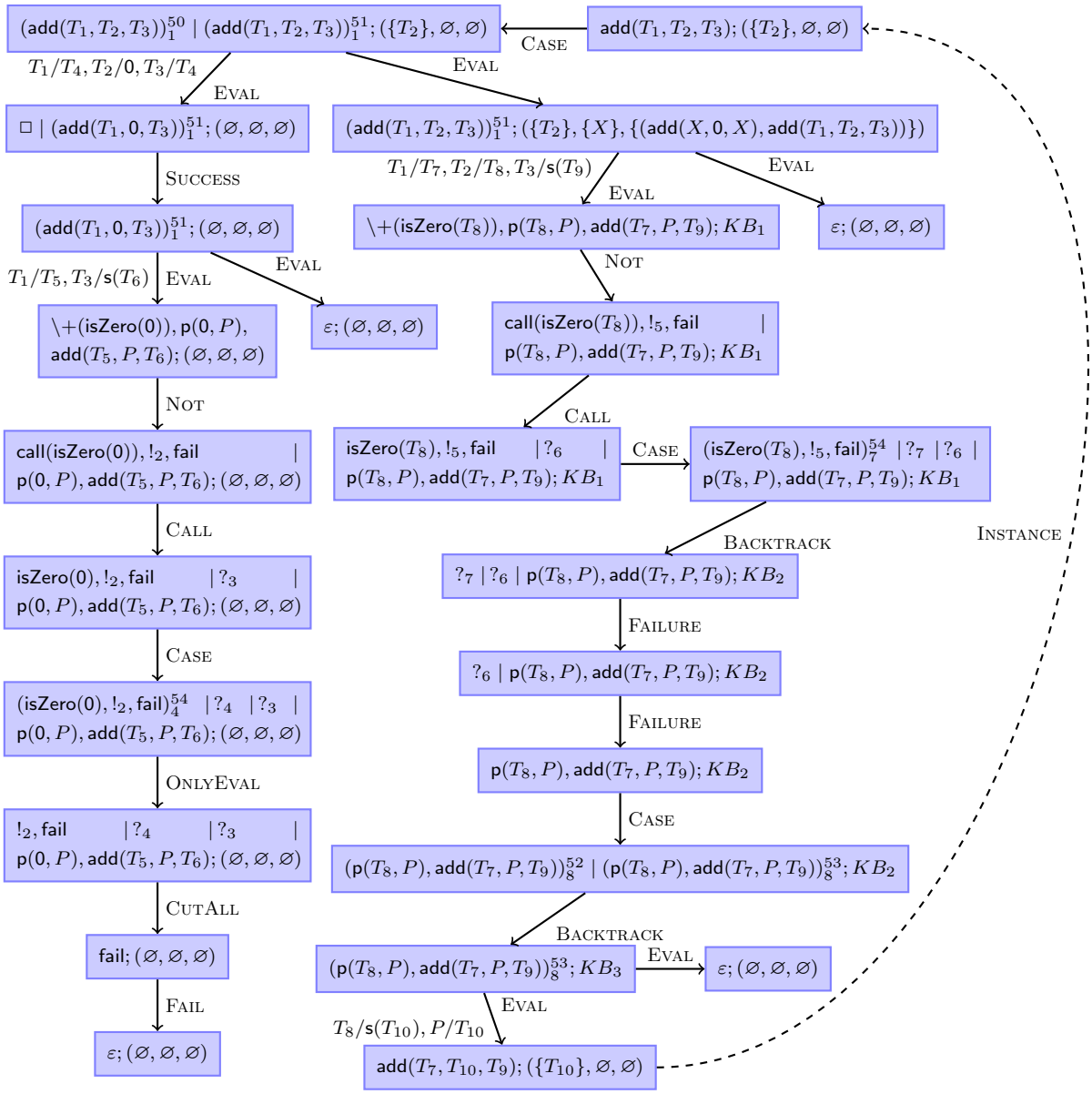
This termination graph represents the following DT.

$$\mathsf{add}_1(T_7, \mathsf{s}(T_{10}), \mathsf{s}(T_9)) \leftarrow \mathsf{add}_1(T_7, T_{10}, T_9).$$

Moreover, the following clauses are represented as well.

$$\begin{aligned}
\mathsf{add}_{c1}(T_4, \mathsf{0}, T_4) &\leftarrow \Box. \\
\mathsf{add}_{c1}(T_7, \mathsf{s}(T_{10}), \mathsf{s}(T_9)) &\leftarrow \mathsf{add}_{c1}(T_7, T_{10}, T_9).
\end{aligned}$$

Using the same assumption for the call set as in Example 7.2, AProVE can easily prove termination of this DT problem.



7.4 Summary

We defined the DT problem which is represented by a termination graph for a Prolog program and query set and proved that termination of the DT problem implies termination of the original Prolog program w.r.t. the specified query set. Hence, we can apply any technique for termination analysis of DT problems to prove termination of Prolog programs. Furthermore, we gave the DT problems which are represented by a number of example termination graphs for Prolog programs from this thesis.

8 Conclusion and Empirical Results

In Chapter 3 we extended the approach from [Sch08] to fully handle the use of cuts and meta-programming (C1) according to the ISO standard for Prolog [DEC96] in the presence of rational terms and unification without occurs-check (C3). We also introduced additional inference rules to handle errors due to uninstantiated variable or undefined predicate calls (C2). Additionally, we closed a gap in the automation of the approach by introducing a new backtracking criterion (C4) while we also improved the precision and speed of this approach by extending the inference rules for evaluation, instantiation and splitting of goals (C5).

We further extended the method to altogether handle 26 built-in predicates (C6) by introducing 33 additional concrete and 41 additional sound abstract inference rules in Chapter 4. Furthermore, we discussed the problems occurring by trying to handle the remaining built-in predicates defined in the ISO standard and gave ideas how to solve these problems in extensions of our method for future work. The results of Chapter 3 were also adapted to the extended rule set where necessary.

We proved in Chapter 5 that the concrete inference rules from our approach can be used to simulate the operational semantics of Prolog according to the ISO standard (C7). Thus, the analysis of termination graphs yields valid results for Prolog programs.

For the deterministic construction of termination graphs for a given Prolog program and query set we introduced an always terminating standard heuristic (C8) which controls the application of graph closing rules in contrast to evaluating rules in Chapter 6.

Afterwards we showed how to obtain a DT problem from a termination graph where termination of the DT problem implies termination of the original Prolog program w.r.t. the query set for which the termination graph was constructed (C9) in Chapter 7. Thus, we can use every technique for termination analysis of DT problems to analyze termination of Prolog programs. Therefore, we have turned the pre-processing for Prolog programs from [Sch08] into a transformation from Prolog programs to DT problems which further increases the precision of this approach.

Throughout the thesis we used examples to illustrate definitions and problems where we have submitted most of these examples to the TPDB and where already 76 examples have been accepted and used for the international Termination Competition 2009 (C11).

Combining the Contributions

Starting with a Prolog program and query set to analyze, Chapter 5 connects such Prolog programs and queries with the concrete inference rules and, consequently, abstract inference rules and termination graphs used in our approach. While Chapter 3 and Chapter 4 allow for an analysis of Prolog programs using more features offered by the ISO standard or real Prolog implementations, Chapter 6 yields an algorithm to deterministically apply the abstract inference rules in order to obtain a termination graph for the Prolog program and query set. According to Chapter 7, this termination graph represents a DT problem whose termination implies termination of the original Prolog program w.r.t. the specified query set. Afterwards, every technique for termination analysis of DT problems can be used to analyze the resulting DT problem. Thus, we have made all such techniques for termination analysis of DT problems available for termination analysis of real Prolog programs as opposed to standard logic programs. Since DT problems can be transformed into dependency pair problems [Sch08, SGN09], all techniques for termination analysis of dependency pair problems (e.g., [AG00, GTSF03, GTS05, TGS04, HM05, GTSF06]) and, therefore, term rewriting (e.g., [Der87, Zan95, AG00, GTSF06, EWZ06, Sch08, FGP+09, SKW+09]) can also be applied to analyze Prolog programs in combination with our transformation.

Empirical Evaluation

The following contributions of this thesis have already been implemented in the fully automated termination prover AProVE and used for the international Termination Competition 2009 [MZ07] (C10).

- We used the more precise and fully automatable abstract evaluation rules Backtrack, Eval and OnlyEval from this thesis.
- We used the pre-processing from [Sch08] to show termination of Prolog programs using cuts and negation-as-failure. The correctness of this method relies on our proof that concrete state-derivations can be used to simulate the operational semantics of Prolog programs.
- To construct termination graphs automatically we used a simplified version of the standard heuristic presented in this thesis. This heuristic already makes use of the distinction between the Instance and Generalization rule.

During the international Termination Competition 2009, AProVE was the only submitted tool capable of analyzing logic programs with cuts, i.e., Prolog programs not using other built-in predicates than !/0 and only using a simplified form of meta-programming without the transformation into applications of the built-in predicate call/1. Since not all examples from the TPDB [TPD09] were used in the international Termination Competition 2009,

we also ran AProVE on all 104 Prolog programs from the TPDB category *LP CUT*. These Prolog programs make use of the cut and, to some extent, of meta-programming as used for negation-as-failure. From these 104 examples there are 10 known to be non-terminating. AProVE managed to prove termination of 78 of the remaining 94 potentially terminating examples. To see the progress of AProVE on Prolog programs using cuts, we additionally ran a version of AProVE on the same example set where both versions are identical except that the second version does not use the pre-processing. Both versions of AProVE were run on a 2.67 GHz Intel Core i7 and, as in the international Termination Competition, we used a time-out of 60 seconds for each example.

For both versions we give the number of examples which could be proved terminating (denoted "Successes"), the number of examples where termination could not be shown ("Failures"), the number of examples for which the timeout of 60 seconds was reached ("Timeouts"), and the total running time ("Total") in seconds. Here, AProVE cut is the version of AProVE using the pre-processing and AProVE 09 is the version where the pre-processing is deactivated.

	AProVE cut	AProVE 09
Successes	**78**	10
Failures	**22**	89
Timeouts	**4**	5
Total	**583.7**	1069.7

The table shows the significant advance in the termination analysis of Prolog programs using cuts. All details of this empirical evaluation can also be seen online and one can run AProVE on arbitrary examples via a web interface [SGS+10].

Nevertheless, except for the features concerning unification without occurs-check, all theoretical contributions of this thesis have been implemented in AProVE, too. By the transformation into DT problems, we can show termination of four additional examples while reducing the average time needed to find termination proofs by half. A very strong increase of power can, however, be seen on the examples from the category *LP LP*, which contains only definite logic programs. Considering the parameters of the heuristic, we have tested AProVE on the complete example set with several combinations of parameter values. The best results yielding a superset of proved examples compared to all other combinations of parameters on this example set was achieved using the parameters $MinExSteps = 2$, $MaxBranchingFactor = 3$, $FiniteGeneralizationDepth = 7$ and $FiniteGeneralizationPosition = 2$. The following table shows the empirical results for AProVE using the new transformation (AProVE DT) compared to the version AProVE cut and a version of AProVE, using a direct transformation into term rewriting and the pre-processing (AProVE parallel cut) or the transformation into DT problems (AProVE parallel DT) in parallel on the category LP LP containing 298 examples.

	AProVE cut	AProVE DT	AProVE parallel cut	AProVE parallel DT
Successes	181	233	234	**235**
Failures	107	57	50	**52**
Timeouts	10	8	14	**11**
Total	2395.5	1835.8	1497.7	**1667.2**

For the category LP CUT we obtain the following results.

	AProVE parallel cut	AProVE DT	AProVE parallel DT
Successes	78	82	**82**
Failures	22	20	**19**
Timeouts	4	2	**3**
Total	892.5	475.8	**591.8**

The parallel execution of the direct transformation into term rewriting and the new transformation into DT problems yields a real superset of proved examples compared to all other versions of AProVE.

Future Work

While the contributions of this thesis significantly improve the state of the art in termination analysis of real Prolog programs, there are still some problems left to solve.

The precision of the abstract inference rules from Chapter 3 and Chapter 4 can be further improved by for example adding knowledge about inequality of terms (as opposed to non-unifiability) or abstract variables which do not represent non-abstract variables themselves (as opposed to ground terms which do not contain variables at all) to the knowledge bases or refining the conditions and approximations used in the abstract inference rules. Especially a more detailed shape analysis could strongly improve the precision of our approach. Also, a more sophisticated analysis of the call set for the DT problems represented by termination graphs might yield a more precise method. Further improvements of both precision and speed might be achieved by extending the standard heuristic or finding alternative or more specialized heuristics and methods to automatically decide which heuristic and which parameters to choose for a certain pair of programs and inputs.

Furthermore, there are still some features of real Prolog programs which are not handled in this thesis. The most important features which should be analyzed are the arithmetical features of Prolog. While there are already existing approaches capable of analyzing numerical features [SD04], another promising approach would be to adapt the results for integer term rewriting [FGP$^+$09] to logic programming or, even better, the dependency triple framework. However, as discussed in Chapter 4, the heuristic for the construction of termination graphs must also be adapted to generalize numbers in order to be still terminating as soon as we consider numbers in our approach. Anyway, the extension

of this framework to analyze programs over an infinite or growing signature would also be important to better handle, e.g., the built-in predicates for type testing. Another point would be to gather knowledge about numerical properties of abstract variables in the knowledge bases. Further important features like error handling or input and output are also open topics. In addition to that, real Prolog implementations usually offer more features than defined in the ISO standard. Definite clause grammars and constraint logic programming are examples for features which are widely supported by real Prolog implementations while they do not belong to the ISO standard. It is practically relevant to handle such features as many real Prolog applications make use of them. For instance, the approaches from [MR03] and [MB05] are capable of analyzing constraint logic programs. However, by handling more features of Prolog some of the assumptions we make in this thesis will not hold anymore, such that some of the features already considered in this thesis have to be re-considered.

Bibliography

[AG00] T. Arts and J. Giesl. Termination of Term Rewriting using Dependency Pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[Apt97] K. R. Apt. *From Logic Programming to Prolog*. Prentice Hall, London, 1997.

[BCG+07] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis of Logic Programs through Combination of Type-Based Norms. *ACM Transactions on Programming Languages and Systems*, 29(2):Article 10, 2007.

[BKN07] L. Bulwahn, A. Krauss, and T. Nipkow. Finding Lexicographic Orders for Termination Proofs in Isabelle/HOL. In *TPHOLs '07*, volume 4732 of *LNCS*, pages 38–53, 2007.

[BM79] R.S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, 1979.

[Cla78] K Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*. Perseus Publishing, 1978. Available online at `http://www.doc.ic.ac.uk/~klc/NegAsFailure.pdf`.

[CLS05] M. Codish, V. Lagoon, and P. J. Stuckey. Testing for Termination with Monotonicity Constraints. In *ICLP '05*, volume 3668 of *LNCS*, pages 326–340, 2005.

[CLSS06] M. Codish, V. Lagoon, P. Schachte, and P. J. Stuckey. Size-Change Termination Analysis in k-Bits. In *ESOP '06*, volume 3924 of *LNCS*, pages 230–245, 2006.

[Col82] A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. Tärnlund, editors, *Logic Programming*. Academic Press, Oxford, 1982.

[CPR05] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *SAS '05*, volume 3672 of *LNCS*, pages 87–101, 2005.

[DD94] D. De Schreye and S. Decorte. Termination of Logic Programs: The Never-Ending Story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[DEC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer, Berlin, Heidelberg, New York, 1996.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1–2):69–116, 1987.

[DS02] D. De Schreye and A. Serebrenik. Acceptability with General Orderings. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond. Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *LNCS*, pages 187–210. Springer-Verlag, 2002.

[DW90] S. K. Debray and D. S. Warren. Towards banishing the cut from Prolog. *IEEE Trans. Software Eng.*, 16(3):335–349, 1990.

[EWZ06] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. In *IJCAR '06*, volume 4130 of *LNAI*, pages 574–588, 2006.

[FGP+09] C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving Termination of Integer Term Rewriting. In *RTA '09*, volume 5595 of *LNCS*, pages 32–47, 2009.

[GST06] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the DP Framework. In *IJCAR '06*, volume 4130 of *LNAI*, pages 281–286, 2006.

[GTS05] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *LPAR '04*, volume 3452 of *LNAI*, pages 301–331, 2005.

[GTSF03] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Improving Dependency Pairs. In *LPAR '03*, volume 2850 of *LNAI*, pages 165–179, 2003.

[GTSF06] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

[GZ03] J. Giesl and H. Zantema. Liveness in Rewriting. In *RTA '03*, volume 2706 of *LNCS*, pages 321–336, 2003.

[HM05] N. Hirokawa and A. Middeldorp. Automating the Dependency Pair Method. *Information and Computation*, 199(1,2):172–199, 2005.

[Hue76] G. Huet. *Résolution d'Equations dans les Langages d'Ordre 1, 2, . . . , ω*. PhD thesis, Université Paris VII, France, 1976.

[KB70] D. Knuth and P. Bendix. Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra*, pages 263–297, 1970.

[LMS03] V. Lagoon, F. Mesnard, and P. J. Stuckey. Termination Analysis with Types Is More Accurate. In *ICLP '03*, volume 2916 of *LNCS*, pages 254–268, 2003.

[LSS97] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A System for Checking Termination of Queries to Logic Programs. In *CAV '97*, volume 1254 of *LNCS*, pages 444–447, 1997.

[Mar96] M. Marchiori. Proving Existential Termination of Normal Logic Programs. In *AMAST '96*, volume 1101 of *LNCS*, pages 375–390, 1996.

[MB05] F. Mesnard and R. Bagnara. cTI: A Constraint-Based Termination Inference Tool for ISO-Prolog. *Theory and Practice of Logic Programming*, 5(1, 2):243–257, 2005.

[MR03] F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.

[MS07] F. Mesnard and A. Serebrenik. Recurrence with Affine Level Mappings is P-Time Decidable for CLP(R). *Theory and Practice of Logic Programming*, 8(1):111–119, 2007.

[MV06] P. Manolios and D. Vroon. Termination Analysis with Calling Context Graphs. In *CAV '06*, volume 4144 of *LNCS*, pages 401–414, 2006.

[MZ07] C. Marché and H. Zantema. The Termination Competition. In *RTA '07*, volume 4533 of *LNCS*, pages 303–313, 2007. See also `http://www.lri.fr/~marche/termination-competition/`.

[ND05] M. T. Nguyen and D. De Schreye. Polynomial Interpretations as a Basis for Termination Analysis of Logic Programs. In *ICLP '05*, volume 3668 of *LNCS*, pages 311–325, 2005.

[ND07] M. T. Nguyen and D. De Schreye. Polytool: Proving Termination Automatically Based on Polynomial Interpretations. In *LOPSTR '06*, volume 4407 of *LNCS*, pages 210–218, 2007.

[NGSD08] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *LOPSTR '07*, volume 4915 of *LNCS*, pages 8–22, 2008.

[OCM00] E. Ohlebusch, C. Claves, and C. Marché. TALP: A Tool for the Termination Analysis of Logic Programs. In *RTA '00*, volume 1833 of *LNCS*, pages 270–273, 2000.

[Sch08] P. Schneider-Kamp. *Static Termination Analysis for Prolog using Term Rewriting and SAT Solving*. PhD thesis, Aachen University of Technology, Germany, 2008.

[SD03] A. Serebrenik and D. De Schreye. Hasta-La-Vista: Termination Analyser for Logic Programs. In *WLPE '03*, pages 60–74. K. U. Leuven, 2003.

[SD04] A. Serebrenik and D. De Schreye. Inference of Termination Conditions for Numerical Loops in Prolog. *Theory and Practice of Logic Programming*, 4(5&6):719–751, 2004.

[SD05] A. Serebrenik and D. De Schreye. On Termination of Meta-Programs. *Theory and Practice of Logic Programming*, 5(3):355–390, 2005.

[SGN09] P. Schneider-Kamp, J. Giesl, and M. T. Nguyen. The Dependency Triple Framework for Termination of Logic Programs. In *LOPSTR '09*, Coimbra, Portugal, 2009. To appear.

[SGS+10] P. Schneider-Kamp, J. Giesl, A. Serebrenik, T. Ströder, and R. Thiemann. Automated termination analysis for logic programs with cut - experiments and proofs, 2010. `http://aprove.informatik.rwth-aachen.de/eval/cut/`.

[SKW+09] H. Sato, M. Kurihara, S. Winkler, , and A. Middeldorp. Constraint-based multi-completion procedures for term rewriting systems. *IEICE Transactions on Information and Systems*, E92-D(2):220 – 234, 2009.

[Sma04] J.-G. Smaus. Termination of Logic Programs Using Various Dynamic Selection Rules. In *ICLP '04*, volume 3132 of *LNCS*, pages 43–57, 2004.

[SP09] T. Ströder and M. Pagnucco. Realizing Deterministic Behaviour from Multiple Non-Deterministic Behaviours. In *IJCAI '09*, pages 936–941, 2009.

[TGC02] C. Taboch, S. Genaim, and M. Codish. TerminWeb: Semantic Based Termination Analyser for Logic Programs, 2002. `http://www.cs.bgu.ac.il/~mcodish/TerminWeb`.

[TGS04] R. Thiemann, J. Giesl, and P. Schneider-Kamp. Improved Modular Termination Proofs Using Dependency Pairs. In *IJCAR '04*, volume 3097 of *LNAI*, pages 75–90, 2004.

[TPD09] The Termination Problem Data Base 7.0 (December 11, 2009), 2009. Available at `http://termcomp.uibk.ac.at/status/downloads/`.

[Tur36] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Available online at `http://www.abelard.org/turpap2/tp2-ie.asp`.

[WSW06] I. Wehrman, A. Stump, and E. Westbrook. Slothrop : Knuth-Bendix Completion with a Modern Termination Checker. In *RTA '06*, volume 4098 of *LNCS*, pages 287–296, 2006.

[Zan95] H. Zantema. Termination of Term Rewriting by Semantic Labelling. *Fundamenta Informaticae*, 24(1–2):89–105, 1995.

Index