TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science, Institute of Theoretical Computer Science, Knowledge-Based Systems Group

Reliance-Based Optimization of Existential Rule Reasoning

Alex Ivliev

A thesis presented for the degree of
Diplom Informatik

Supervisor: Prof. Dr. Markus Krötzsch
External Examiner: Prof. Dr. Sebastian Rudolph
Submitted on: December 3, 2021
Defended on: December 9, 2021

Abstract

Existential rules are a powerful formalism for describing implicit knowledge contained within a knowledge base. Extracting such knowledge can be achieved with the chase, which is a well-known algorithm for computing universal models. This is done by exhaustively calculating the consequences of each given rule. However, the order in which the rules are selected can have a significant impact on the run time of the procedure as well as the number of derived facts. It was discovered in recent work that core-stratified rule sets allow for a selection strategy that is guaranteed to produce so-called core models, which correspond to the smallest possible universal models if they are finite. The strategy is based on considering certain syntactic relationships between the rules called reliances. These indicate whether it is possible for a rule to produce facts used by another or if selecting a rule in the wrong order may introduce redundant facts into the result.

In this work, we utilize these reliances to devise rule application strategies that optimize the chase procedure based on the following criteria: First, we try to minimize the number of times rules are applied during the chase, aiming to improve run times. Second, we want to avoid applying rules in a way which introduces redundant facts. The goal here is to minimize the size of the resulting model, ideally producing a core. While it is always possible to derive a core model in core-stratified rule sets, we show situations where our approach is guaranteed to produce cores even if the rule set is not stratified. Moreover, we give a detailed description of the algorithms necessary for detecting a reliance relationship between two given rules as well as prove their correctness. We implement our approach into the rule reasoning engine VLog and evaluate its effectiveness on several knowledge bases used for benchmarking as well as some real-world data sets. We find a significant improvement in the run times for a small portion of the considered knowledge bases and are able to match VLog in the remaining ones.

Declaration of Authorship

I hereby declare that I have written this thesis independently and have listed all used sources and aids. I am submitting this thesis for the first time as part of an examination. I understand that attempted deceit will result in the failing grade „not sufficient“ (5.0).

(Personal information and signature omitted in online version.)

Date

Signature

Acknowledgements

First of all, I would like to thank my supervisor, Prof. Dr. Markus Krötzsch, for providing guidance and advice throughout this project and for always responding quickly to my questions or concerns.

In addition, I also want to thank Prof. Dr. Sebastian Rudolph for agreeing to serve as the external examiner for this work.

I would like to extend my sincere thanks to Lukas Gerlach for proofreading my thesis on short notice and providing very helpful comments and feedback.

I am also grateful to Eric Franke for helping me to stay focused during some of the more stressful periods of writing.

Lastly, I want to thank my family and friends for their encouragement and support during the time of my studies.

Contents

Chapter 1

Introduction

1.1 Existential Rules

Datalog extended with existential rules, also known as Datalog$^{\pm}$ or $\forall\exists$-rules [6, 3], is a powerful formalism that plays a huge role in the field of Knowledge Representation and Reasoning. It was originally described as tuple-generating dependencies where it represents integrity constraints [1], but is now used in logic programming, for data exchange, and also for capturing implicit knowledge in ontologies. Syntactically, existential rules are universally quantified implications, which may contain existentially quantified variables in the implication's consequent. The following example shows a set of existential rules. We usually omit universal quantifiers in the formulas.

Example 1.1.

$$\mathsf{leadingRole}(a, r, m) \to \mathsf{stars}(a, m) \qquad (\rho_1)$$

$$\mathsf{stars}(a, m) \wedge \mathsf{stars}(b, m) \to \mathsf{costar}(a, b, m) \qquad (\rho_2)$$

$$\mathsf{bigBudget}(m) \to \exists a.\, \mathsf{stars}(a, m) \wedge \mathsf{famous}(a) \qquad (\rho_3)$$

The above set of rules might describe implicit knowledge in an ontology containing information about various movies and actors. The first rule states that if an actor a plays a leading role r in a movie m, then this implies that a also stars in m. If two actors star in the same movie, then the second rule tells us that they are considered costars. The last rule asserts that any movie with a big budget has to have, perhaps to justify the huge cost of production, at least one famous actor staring in it. $\triangle$

A set of rules together with a database form a knowledge base. A fundamental reasoning task over such knowledge bases is to answer boolean conjunctive queries (BCQs).

Example 1.2. Given a knowledge base containing rules from Example 1.1 and a database $\mathcal{D} = \{\text{role}(\text{“Alice”}, \text{“Rick”}, \text{“Electric Sheep”}),\ \text{bigBudget}(\text{“Electric Sheep”})\}$, the following boolean conjunctive query encodes the question of whether Alice costars with a famous actor in any movie.

$$q = \exists b, m.\, \text{costar}(\text{“Alice”}, b, m) \wedge \text{famous}(b)$$

Although the database itself does not explicitly contain the required facts, we can use the above set of rules to deduce new information. From the first rule we know that since Alice played a major part in the movie “Electric Sheep” that she therefore stars in it as well. “Electric Sheep” is a big budget production, which means there has to be at least one famous actor staring in it as stated by the third rule. Consequently, Alice and said famous actor (who ironically is unknown to the database) both star in the same movie and are therefore costars by the second rule. Hence, the query q is entailed by the knowledge base consisting of the database $\mathcal{D}$ and the above set of rules. △

BCQ entailment is equivalent to determining whether the given formula is satisfied in every model of the knowledge base. In practice however, it suffices to only consider universal models [10]. These can be thought of as representative models, since they can be homomorphically mapped into any other model. Queries that are entailed by a universal model are therefore also entailed by every other model of the given knowledge base.

1.2 The Chase and Core Models

The chase refers to a class of algorithms that compute universal models for a given knowledge base [20, 10]. This is achieved through a bottom-up materialization of “missing” facts by exhaustively computing the consequences of every rule in the rule set. Conceptually, this mimics the reasoning we employed in Example 1.2. In this work, we mainly consider the restricted chase, which also known as the standard chase, though we also examine slight variations of it in the context of practical implementation.

The description of the standard chase does not restrict the order in which the consequences of a given rule have to be computed in. Different orders of materializing said consequences may lead to different (non-isomorphic) universal models. Amongst those, core models seem to be the most preferable. Intuitively, core models are universal models that contain the least amount of redundancy [16, 10]. Finite core models correspond to the smallest possible models and are therefore desirable in regard to space efficiency. With the core chase, there exists a variant of the chase algorithm that guarantees to produce core models for every input [10]. However, since it relies on the computation of homomorphisms within the whole augmented database, using this variant is computationally expensive compared to just using the standard chase.

That being said, there are cases where the comparatively cheap restricted chase produces core models without the need for any additional computation. Recent work shows that core-stratified rule sets allow for a rule application strategy that is guaranteed to produce a core [18]. But to the best of our knowledge, there exists no reasoning engine which takes this into account.

1.3 The Rule Engine VLog

VLog is a rule-based reasoning engine, which supports existential rules [23, 24, 9]. It implements a parallel version of the restricted chase where each derivation step calculates every consequence of a rule at once. VLog's unique selling point is the use of a column-based data structure for storing inferred facts. Columnar layouts allow for certain data-compression strategies and improve the performance of join algorithms employed during the derivations. However, the insertion of new facts into a columnar table is significantly more time-consuming than with traditional row-based layouts. VLog avoids frequent insertions by storing new facts in a separate table for each derivation. The entries of a predicate may therefore be split over several blocks of data. But this can lead to considerable overhead when joins need to be performed over multiple blocks. This problem is exacerbated when rules are applied frequently and hence new blocks are introduced more often. Rule application strategies that minimize the number of derivation steps therefore promise to increase performance.

1.4 Goal and Structure

The main goal of this thesis is to improve the restricted chase, particularly its implementation in VLog, by optimizing the order in which rules are applied in. We identify two dimensions along which to optimize:

1. Minimize the number of redundant derivations, producing core-models if possible
2. Minimize the number of derived predicate-blocks by applying rules as rarely as possible

Our strategy is based on analyzing certain syntactic relationships between rules, called reliances [18], which impose an ordering on the given rule set. We also implement our approach into VLog and evaluate its performance.

This work is structured as follows. Chapter 2 provides the basic definitions and notions used throughout the thesis. This includes the chase algorithm, the concept of reliances and core-stratification as well as some architectural aspects of VLog. We dedicate the entirety of Chapter 3 to the derivation of algorithms for detecting a reliance relationship between two rules. We further give detailed proofs that show their correctness. In Chapter 4, we use utilize the result of the previously given computation to formulate a rule application strategy that optimizes the restricted chase with respect to the goals mentioned above. We evaluate the effectiveness of the given strategies in Chapter 5. This is done by measuring the run time and number of derived facts on several knowledge bases used for benchmarking as well some real-world data sets. We put potential improvemens compared to the original strategy by VLog in relation the cost of calculating the reliance relations.

Chapter 2

Preliminaries

This chapter provides the background on fundamental concepts necessary for this work. We start by defining basic terms and notation used throughout the thesis in Section 2.1. We then introduce the chase in Section 2.2, including all of the major variants considered here. The following section focuses on core models, specifically on cases where they can be produced by the restricted chase without any additional computation. Crucially, this section also defines the syntactic relationship between rules on which we build our application strategies in Chapter 4. Lastly, Section 2.4 gives an overview over the rule execution engine VLog and how the order of the application of rules may influence its performance.

2.1 Basic Notation and Existential Rules

A *(directed) graph* is an ordered pair $G = \langle V, E \rangle$ consisting of a finite set of nodes V and a set of edges $E \subseteq V \times V$. If $\langle v, w \rangle \in E$, we refer to v as the *predecessor* of w and to w as the *successor* of v. For any node $v \in V$, we define $\text{pred}_G(v)$ and $\text{succ}_G(v)$ as the sets of all predecessors and successors of v in G respectively. A series of nodes $v_1, \ldots, v_n$ with $n \geq 2$ is a *(directed) path* from v_1 to v_n in G if $\langle v_i, v_{i+1} \rangle \in E$ for every $i \in [n-1]$. If $v_1 = v_n$ then the path is a *cycle* . A cycle is *proper* if it contains at least two distinct vertices. We say a node $w \in V$ is reachable from a node $v \in V$ in G, written $\text{reach}_G(v, w)$, if $v = w$ or if there exists a directed path from v to w in G. A graph G is *acyclic* if there is no cycle in G. A *topological sorting* of G is a series of pairwise distinct nodes $v_1, \ldots, v_n$ where $n = |V|$ such that $i < j$ implies that $\langle v_j, v_i \rangle \notin E$ for every $i, j \in [n]$. A graph G is *topologically sortable* if a topological sorting of G exists. This is the

case if and only if the graph does not contain any proper cycles. A set of nodes $V' \subseteq V$ is strongly connected if $\mathsf{reach}_G(v, w)$ for every $v, w \in V'$. We write $[v]_G$ for the strongly connected component containing v or simply $[v]$ if G is clear from context. It is sometimes useful to think of any binary relation $\prec \subseteq A \times A$ over some set A as a graph. In this case, we write $G(A, \prec) \coloneqq \langle A, \prec \rangle$ to refer to the graph corresponding to A and $\prec$. We define $A/_\prec$ to be the set of all strongly connected components in $G(A, \prec)$. We sometimes refer to a strongly connected component in $A/_\prec$ as a *$\prec$-group*. Based on that, we define $\hat{G}(A, \prec) \coloneqq G(A/_\prec, \hat{\prec})$ where $[a] \mathrel{\hat{\prec}} [b]$ iff $a' \prec b'$ for any $a' \in [a]$ and $b' \in [b]$. Note that $\hat{G}(A, \prec)$ cannot contain any proper cycles and is therefore always topologically sortable.

Let A and B be sets and $f \colon A \to B$ be a function. We call $\mathsf{dom}(f) \coloneqq A$ the *domain* and $\mathsf{im}(f) \coloneqq \{b \in B \mid \exists a \in A.\, f(a) = b\}$ the *image* of f. The *graph* of f is the set $\mathsf{graph}(f) \coloneqq \{a \mapsto b \mid a \in \mathsf{dom}(f),\, f(a) = b\}$. Given $b \in B$, we define $f^{-1}(b) \coloneqq \{a \in A \mid f(a) = b\}$. Let $X \subseteq A$ be a set. Then $f(X) \coloneqq \{f(x) \in B \mid x \in X\}$.

We construct first-order logic formulas over a vocabulary of countably infinite, mutually disjoint sets $\mathbf{V}$ of variables, $\mathbf{C}$ of *constants*, $\mathbf{N}$ of *labeled nulls* and $\mathbf{P}$ of *predicate names*. We further distinguish *universally quantified variables* $\mathbf{V}_\forall \subseteq \mathbf{V}$ and *existentially quantified variables* $\mathbf{V}_\exists \subseteq \mathbf{V}$ where $\mathbf{V}_\forall \cap \mathbf{V}_\exists = \emptyset$. Predicate names are assumed to have an *arity* $\mathsf{ar}(p) \geq 1$ for every $p \in \mathbf{P}$. A *term* is an element of the set $\mathbf{T} \coloneqq \mathbf{V} \cup \mathbf{C} \cup \mathbf{N}$. We use $\bar{t}$ to denote a list of terms $t_1, \ldots, t_{|\bar{t}|}$. An *atom* is an expression $p(\bar{t})$ where $p \in \mathbf{P}$ and $\bar{t}$ is a list of terms such that $|\bar{t}| = \mathsf{ar}(p)$. If $\mathcal{A}$ is a set of atoms, then $\mathsf{vars}(\mathcal{A})$ is the set of all variables contained in $\mathcal{A}$. We define $\mathsf{nulls}(\mathcal{A})$ and $\mathsf{terms}(\mathcal{A})$ analogously. Furthermore, $\mathsf{vars}_\forall(\mathcal{A})$ and $\mathsf{vars}_\exists(\mathcal{A})$ are the sets of all universal and all existential variables in $\mathcal{A}$ respectively. Atoms that do not contain any variables are called *facts*. An *interpretation* is a (possibly infinite) set $\mathcal{I}$ that only contains variable-free atoms. A finite interpretation without nulls is called a *database*.

We now define existential rules. We use the notation $\varphi[\bar{x}]$ for $\bar{x} \subseteq \mathbf{V}$ to mean that φ is a conjunction of atoms which exactly contains the variables in $\bar{x}$.

Definition 2.1. An *(existential) rule* ρ is a first-order logic formula

$$\rho = \forall \bar{x}, \bar{y}.\, \varphi[\bar{x}, \bar{y}] \to \exists \bar{z}.\psi[\bar{y}, \bar{z}]$$

where $\bar{x}, \bar{y} \subseteq \mathbf{V}_\forall$ and $\bar{z} \subseteq \mathbf{V}_\exists$ such that φ and ψ do not contain any nulls. We refer to φ as the *body* and to ψ as the *head* of the rule.

Rules that do not contain any existential variables are called *datalog rules*. In the following, we treat conjunctions of atoms as sets. A *knowledge base* $\mathcal{K} = \langle R, \mathcal{D} \rangle$ is an ordered pair consisting of rule set R and a database $\mathcal{D}$.

A *substitution* σ is any function $\sigma \colon \mathbf{S} \to \mathbf{T}$ where $\mathbf{S} \subseteq \mathbf{T}$. Given a term $t \in \mathbf{T}$, the result of *applying* σ to t, written $t\sigma$, is $\sigma(t)$ if $t \in \mathsf{dom}(\sigma)$ or t otherwise. A substitution σ can *applied* to an atom $p(\bar{t})$, written $p(\bar{t})\sigma$, by applying σ to each term in $\bar{t}$ and to a set of atoms $\mathcal{A}$, written $\mathcal{A}\sigma$, by applying σ to each atom in $\mathcal{A}$. If $\mathsf{dom}(\sigma) \subseteq \mathbf{V}$ and $\mathsf{im}(\sigma) \subseteq \mathbf{C} \cup \mathbf{N}$, then σ is called a *variable substitution*. Similarly, a *null substitution* is a substitution σ where $\mathsf{dom}(\sigma) \subseteq \mathbf{N}$ and $\mathsf{im}(\sigma) \subseteq \mathbf{C} \cup \mathbf{N}$. If $\mathsf{dom}(\sigma) = \mathbf{T}$, then σ is called a *total substitution*. For a variable substitution σ we write $\sigma_\forall \coloneqq \sigma_{|\mathbf{V}_\forall}$ and $\sigma_\exists \coloneqq \sigma_{|\mathbf{V}_\exists}$ as a shorthand for σ restricted to universal and existential variables respectively. If σ and τ are substitutions then $\sigma\tau$ is the substitution with the domain $\mathsf{dom}(\sigma\tau) = \mathsf{dom}(\sigma) \cup \mathsf{dom}(\tau)$, which is obtained by first applying σ and then τ. Clearly, we have that $\sigma = \sigma_\forall\sigma_\exists = \sigma_\exists\sigma_\forall$ for any variable substitution σ. We say that σ is a *unifier* of two atom sets $\mathcal{A}_1$ and $\mathcal{A}_2$ if $\mathcal{A}_1\sigma = \mathcal{A}_2\sigma$. If a unifier of $\mathcal{A}_1$ and $\mathcal{A}_2$ exists, then they are considered to be *unifiable*.

Let $\mathcal{A}$ be a set of atoms and $\mathcal{I}$ an interpretation. A *homomorphism* from $\mathcal{A}$ to $\mathcal{I}$ is a variable substitution h such that $\mathcal{A}h \subseteq \mathcal{I}$. If a homomorphism from $\mathcal{A}$ to $\mathcal{I}$ exists, we write $\mathcal{I} \models \mathcal{A}$. A homomorphism h' *(existentially) extends* h if $h'(x) = h(x)$ for all $x \in \mathbf{V}_\forall$. This is equivalent to saying that that $h' = h'_\exists h_\forall = h_\forall h'_\exists$ for some variable substitution $h'_\exists \colon \mathbf{V}_\exists \to \mathbf{T}$. A rule $\rho = \varphi \to \psi$ is *satisfied* over an interpretation $\mathcal{I}$ if every homomorphism from φ to $\mathcal{I}$ can be extended to a homomorphism from ψ to $\mathcal{I}$.

Definition 2.2. Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base. An interpretation $\mathcal{I}$ is considered a *model* of $\mathcal{K}$ if

- $\mathcal{D} \subseteq \mathcal{I}$ and
- every rule $\rho \in R$ is satisfied over $\mathcal{I}$.

A model $\mathcal{I}$ of a knowledge base $\mathcal{K}$ is *universal* if for any (other) model $\mathcal{J}$ of $\mathcal{K}$ there exists a null substitution ν such that $\mathcal{I}\nu \subseteq \mathcal{J}$.

2.2 The Chase

The chase is a general term for various algorithms that compute universal models for a given knowledge base [20, 10]. Starting from the initial database, the chase iteratively derives new facts that are implied by the given rule set, which are then added to the current interpretation. This process is repeated until every rule is satisfied meaning that no new facts can be derived. There are, however, inputs for which such a procedure fails to terminate and detecting such cases turns out to be undecidable [4, 10, 14].

The details of each derivation step vary across different variants of the chase. In this work, we are mainly concerned with the restricted or standard chase and its practical implementation in the rule engine VLog, which is why we also introduce the 1-parallel and Datalog-First chase.

2.2.1 The Restricted Chase

In the restricted chase, each derivation step consists of finding a variable substitution that satisfies the body of a rule but not its head. We refer to such substitutions as unsatisfied matches.

Definition 2.3. Let $\rho = \varphi \to \psi$ be a rule and $\mathcal{I}$ be an interpretation. A variable substitution h is

- a *match* for ρ over $\mathcal{I}$ if h is a homomorphism from φ to $\mathcal{I}$ and
- a *satisfied match* if h is a match and there exists a homomorphism h' from ψ to $\mathcal{I}$ which extends h. Otherwise, h is an *unsatisfied match* for ρ over $\mathcal{I}$.

Given an interpretation $\mathcal{I}$ and a rule ρ, we say that ρ is *applicable* over $\mathcal{I}$ if an unsatisfied match for ρ over $\mathcal{I}$ exists. The result of *applying* or *satisfying* a rule ρ over $\mathcal{I}$ with the unsatisfied match h is an interpretation $\mathcal{J} = \mathcal{I} \cup \psi h'$ such that h' existentially extends h by assigning unique and unused nulls to each existential variable in ψ. Intuitively, each null represents an unknown entity that serves as a placeholder. Starting from the initial database and continuously applying rules results in a restricted chase sequence.

Definition 2.4. Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base and $(\mathcal{I}_k) = \mathcal{I}_0, \mathcal{I}_1, \ldots$ a series of interpretations. We call $(\mathcal{I}_k)$ a *restricted chase sequence* for $\mathcal{K}$ if the following properties hold:

- $\mathcal{I}_0 \coloneqq \mathcal{D}$
- Validity: $\mathcal{I}_k$ is obtained by applying a rule $\rho \in R$ over $\mathcal{I}_{k-1}$ for all $k > 0$.
- Fairness: If h is an unsatisfied match over $\mathcal{I}_k$ for some rule $\rho \in R$ and some $k \in \mathbb{N}$, then there exists $k' \in \mathbb{N}$ such that h is a satisfied match for ρ over $\mathcal{I}_{k'}$.

The *result* of a chase sequence $(\mathcal{I}_k)$ is a interpretation $\mathcal{I}_\infty \coloneqq \bigcup_{k\in\mathbb{N}} \mathcal{I}_k$. In this context, we refer to each entry $\mathcal{I}_k$ as a *chase step*.

During a restricted chase run, there might be one or more rules which continuously become applicable, thereby leading to an infinite sequence. Fairness ensures that each rule is satisfied at some point during the run, which guarantees that the computed interpretations approach a universal model. On the other hand, some rules cease to become applicable after a certain point in the chase. Since they cannot influence the procedure once this is the case, they can be removed from consideration.

Definition 2.5. Let R be a set of rules and $\mathcal{I}$ be an interpretation. A rule $\rho \in R$ is *inactive* over $\mathcal{I}$ and R if there does not exist a restricted chase sequence $\mathcal{I}_0, \mathcal{I}_1, \ldots$ for $\langle R, \mathcal{I}\rangle$ such that ρ is applicable over $\mathcal{I}_k$ for some $k \in \mathbb{N}$.

As shown in the next proposition, determining the exact moment after which a rule becomes inactive is undecidable. Nevertheless, there are still sufficient conditions that prove that a rule is inactive at a certain time in the chase run, though it might have been already the case earlier. To stress this fact, we refer to rules that are not proven to be inactive at some point as *potentially active*.

Proposition 2.1. *Let R be a set of rules and $\mathcal{I}$ be an interpretation. The problem of determining if ρ is active over $\mathcal{I}$ and R is undecidable.*

Proof. The proof is a reduction from BCQ answering, which is known to be undecidable [4]. Let $q = \exists \bar{x}.\, \varphi[\bar{x}]$ be a boolean conjunctive query. We can append to R the rule $\rho_q = \varphi[\bar{x}] \to \exists v.\, r(v)$ where r is some new unary predicate. Determining whether or not ρ_q is active over $\mathcal{D}$ is equivalent to determining whether or not q is entailed by $\mathcal{K}$. □

Throughout the thesis we will consider different algorithms that compute restricted chase sequences to which we will refer to as *chase algorithms*. In general, those will be non-deterministic procedures which allow for different (non-isomorphic) universal models. However, we do not require that every possible

restricted chase sequence should be a possible outcome of a chase algorithm. In fact, we oftentimes specifically exclude certain potential chase sequences with the intention of optimizing the actual computation. In the following, we present the basic restricted chase algorithm on top of which further improvements will be built.

Algorithm 2.1: restricted chase

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$
Output: Chase result for $\mathcal{K}$

Function `applyRule`(*Rule* $\rho = \varphi \to \psi$*, Interpretation* $\mathcal{I}$)**:**
    Find an unsatisfied match h for ρ over $\mathcal{I}$;
    $h' \coloneqq h_\forall \cup \{v \mapsto n_v \mid v \in \mathsf{vars}_\exists(\psi)\}$ where n_v is a unique null unused in $\mathcal{I}$;
    $\mathcal{I}' \coloneqq \mathcal{I} \cup \psi h'$;
    return $\mathcal{I}'$;
$\mathcal{I} \coloneqq \mathcal{D}$;
while R *contains a rule which is applicable over* $\mathcal{I}$ **do**
    $\rho \coloneqq \mathsf{select}(R)$;
    $\mathcal{I} \coloneqq \mathsf{applyRule}(\rho, \mathcal{I})$;
end
return $\mathcal{I}$

The procedure shown in Algorithm 2.1 mainly consists of a single while-loop that continuously selects a rule from the given rule set. In each step, a match of the selected rule over the current interpretation $\mathcal{I}$ is satisfied by calling the function `applyRule`. It extends said match with unique and unused nulls and adds the resulting facts into the current interpretation. Here, we assume that `select` is a function that picks a (applicable) rule in a fair manner. For practical purposes, we will also consider algorithms that may violate fairness in the non-terminating case and only require that every rule is satisfied over finite results. We refer to such procedures as *semi-fair* chase algorithms. Infinite sequences of interpretations that result from such semi-fair chase run are no longer guaranteed to approach a universal model. However, we deem this property to be of little value in practical applications and instead shift our focus to the finite case.

We show a possible run of the restricted chase on the rule set from the introductory example. Here, we purposefully choose an inefficient ordering for applying rules with the intention to demonstrate potential optimizations.

Example 2.1. Assume the rule set from the introductory Example 1.1 and the database

$$\mathcal{D} = \{\text{leadingRole}(\text{“Alice”}, \text{“Rick”}, \text{“Electric Sheep”}),\\ \text{bigBudget}(\text{“Electric Sheep”}), \text{famous}(\text{“Alice”})\}.$$

The following sequence of chase steps constitutes a possible restricted chase run. Starting with $\mathcal{I}_0 \coloneqq D$, we get the following sequence of interpretations:

$$\mathcal{I}_1 \coloneqq \mathcal{I}_0 \cup \{\text{stars}(n, \text{“Electric Sheep”}, \text{famous}(n))\} \quad (\rho_3)$$
$$\mathcal{I}_2 \coloneqq \mathcal{I}_1 \cup \{\text{costar}(n, n, \text{“Electric Sheep”}))\} \quad (\rho_2)$$
$$\mathcal{I}_3 \coloneqq \mathcal{I}_2 \cup \{\text{stars}(\text{“Alice”}, \text{“Electric Sheep”})\} \quad (\rho_1)$$
$$\mathcal{I}_4 \coloneqq \mathcal{I}_3 \cup \{\text{costar}(\text{“Alice”}, \text{“Alice”}, \text{“Electric Sheep”})\} \quad (\rho_2)$$
$$\mathcal{I}_5 \coloneqq \mathcal{I}_4 \cup \{\text{costar}(\text{“Alice”}, n, \text{“Electric Sheep”})\} \quad (\rho_2)$$
$$\mathcal{I}_6 \coloneqq \mathcal{I}_5 \cup \{\text{costar}(n, \text{“Alice”}, \text{“Electric Sheep”})\} \quad (\rho_2)$$

From this point onward, every rule is satisfied and the algorithm terminates. The null n introduced in the beginning of the chase run serves as a placeholder for any famous actor who stars in “Electric Sheep”. However, we point out that since Alice is famous themselves and plays a leading role in said move, the placeholder n could also refer to them as well. This indicates a redundancy in the chase result which, as we will see, was introduced due to the order we applied the rules in. △

The main computational task performed by the restricted chase algorithm shown in this section is finding unsatisfied matches for every selected rule. This problem has been shown to be Σ_2^{P}-complete w.r.t. the size of the rule [14]. Basically, it involves first finding a homomorphism from the body of the rule to the interpretation which is a NP-complete problem. Then, another NP-oracle is needed to verify that the given match is unsatisfied.

2.2.2 1-Parallel Chase

In the restricted chase, individual matches are satisfied one at the time. Benchmarks have shown that applying rules in batches, i.e. satisfying every match of a given rule at once, may lead to better overall performance [5]. This leads to a chase variant known as the 1-parallel chase, which is the variant implemented in VLog.

Let $\mathcal{I}$ be an interpretation and $\rho = \varphi \to \psi$ a rule. We define

$$\mathfrak{A}(\mathcal{I}, \rho) = \{h \colon \mathsf{vars}(\varphi) \to \mathbf{T} \mid h \text{ is applicable for} \rho \text{ over } \mathcal{I}\}$$

as the set which contains every (relevant) unsatisfied match for ρ over the interpretation $\mathcal{I}$.

Definition 2.6. Let $(\mathcal{I}_k) = \mathcal{D}, \mathcal{I}_1, \mathcal{I}_2, \ldots$ be a fair sequence of chase steps for a knowledge base $\mathcal{K} = \langle R, \mathcal{D} \rangle$. The sequence $(\mathcal{I}_k)$ is a *1-parallel* chase sequence if

- for every $k > 0$ we have $\mathcal{I}_k = \bigcup_{h \in \mathfrak{A}(\mathcal{I}_{k-1}, \rho)} \mathcal{I}^h$ such that $\mathcal{I}^h$ was obtained by applying $\rho \in R$ over $\mathcal{I}_{k-1}$ for the match h.

We obtain a 1-parallel chase algorithm by replacing the call to `applyRule` in Algorithm 2.1 with a call to the following function.

Function `apply-1-parallel(`*Rule* $\rho = \varphi \to \psi$, *Interpretation* $\mathcal{I}$`)`**:**
  $\Delta \coloneqq \emptyset$;
  foreach *unsatisfied match* $h \colon$ *vars*$(\varphi) \to \mathbf{T}$ *for* ρ *over* $\mathcal{I}$ **do**
    $h' \coloneqq h \cup \{v \mapsto n_v \mid v \in \mathsf{vars}_\exists(\psi)\}$ where n_v is a null unused in $\mathcal{I} \cup \Delta$;
    $\Delta \coloneqq \Delta \cup \psi h'$;
  end
  $\mathcal{I}' \coloneqq \mathcal{I} \cup \Delta$;
  return $\mathcal{I}'$;

For theoretical purposes, we will treat the 1-parallel chase as a restricted chase algorithm that always satisfies every match of a rule before moving on to the next. This view is technically incorrect, since applying the same rule in separate steps may cause matches to become satisfied. But it will allow us to use certain definitions already provided in the literature and simplify theoretical arguments.

2.2.3 Datalog-First Chase

The variants of the chase considered thus far do not impose any restriction on the order that rules are applied in as long as the chosen order is fair. This allows for chase sequences of varying length. In particular, the chosen order of rule application might influence whether a chase sequence is finite or infinite as seen in the next example.

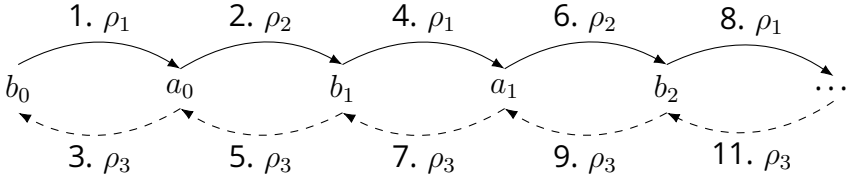


Figure 2.1: Infinite chase sequence for the knowledge base shown in Example 2.2. Arrows represent the writtenBy and authorOf relationship. A solid line implies the introdoction of a new null.

Example 2.2. Consider the following set of rules.

$$\mathsf{book}(x) \to \exists v.\, \mathsf{writtenBy}(x, v) \wedge \mathsf{author}(v) \qquad (\rho_1)$$

$$\mathsf{author}(x) \to \exists w.\, \mathsf{authorOf}(x, w) \wedge \mathsf{book}(w) \qquad (\rho_2)$$

$$\mathsf{authorOf}(x, y) \to \mathsf{writtenBy}(y, x) \qquad (\rho_3)$$

$$\mathsf{writtenBy}(x, y) \to \mathsf{authorOf}(y, x) \qquad (\rho_4)$$

The first two rules encode that every book must have an author and similarly, that every author must have written a book. The last two are datalog rules which simply state that authorOf is the inverse relation of writtenBy and vice versa. Starting with $\mathcal{D} = \{\mathsf{book}(b_0)\}$ it is possible to obtain an infinite restricted chase sequence by alternating between applying ρ_1 and ρ_2 while occasionally satisfying ρ_3 and ρ_4 on older matches to guarantee fairness. The resulting sequence is visualized in Figure 2.1 However, prioritizing the datalog rules ρ_3 and ρ_4 – applying them as soon as an unsatisfied match is available – ensures that the resulting sequence will be finite. In our case, this results in the following finite interpretation $\mathcal{I} = \mathcal{D} \cup \{\mathsf{writtenBy}(b_0, a_0), \mathsf{author}(a_0), \mathsf{authorOf}(a_0, b_0)\}$. △

The previous example motivates a strategy for selecting rules that prefers the application of datalog rules over others [7]. Since a datalog rule $\rho = \varphi \to \psi$ does not contain any existential variables, an unsatisfied match h can only be satisfied by introducing ψh into the chase result. Therefore, delaying the application of a datalog rule cannot prevent the consequence of that rule from being added into the resulting model. However, applying the rule early might still satisfy a non-datalog rule. We can implement a Datalog-First strategy by replacing the generic function `select` from Algorithm 2.1 with the following function `selecDLF`.

```
Function selectDLF(RuleSet R):
    R_∃ := {ρ = φ → ψ ∈ R | vars_∃(ψ) ≠ ∅};
    R_dl := R \ R_∃;
    if R_dl contains a rule which is applicable over I then
        return select(R_dl)
    end
    else
        return select(R_∃)
    end
```

This results in a Datalog-First restricted or a Datalog-First 1-parallel chase algorithm. We note that this is still a fair selection strategy because the exhaustive application of datalog rules can only introduce a finite number of new facts.

Needless to say, we do not obtain any benefit from using a Datalog-First strategy on rule sets that do not contain any datalog rules.

Example 2.3. Consider a slight variation on the rule set from Example 2.2.

$$\mathsf{book}(x) \to \exists v, i.\, \mathsf{writtenBy}(x, v, i) \wedge \mathsf{author}(v) \tag{ρ_1}$$

$$\mathsf{author}(x) \to \exists w, i.\, \mathsf{authorOf}(x, w, i) \wedge \mathsf{book}(w) \tag{ρ_2}$$

$$\mathsf{authorOf}(x, y, j) \to \exists i.\, \mathsf{writtenBy}(y, x, i) \tag{ρ_3}$$

$$\mathsf{writtenBy}(x, y, j) \to \exists i.\, \mathsf{authorOf}(y, x, i) \tag{ρ_4}$$

Here, a new existential variable is introduced into every rule. A Datalog-First chase therefore behaves the same as a normal restricted or 1-parallel chase would on the same rule set. Nevertheless, a strategy that prioritizes ρ_3 and ρ_4 is still able to break infinite cycles caused by the first two rules. We show a way to detect this case in Chapter 4. △

2.3 Cores and Core-Stratification

As seen in Example 2.2, one knowledge base may admit several non-isomorphic universal models. Of those, core models seem to be the most preferable. Intuitively, a core model is a universal model which cannot be embedded into a smaller substructure of itself [16, 10]. Although there might exist multiple non-isomorphic infinite core models [8], in the finite case, cores corresponds to the smallest possible universal models. This makes computing core models attractive as an optimization since any materialization of a core model consumes the

least amount of space possible. In addition, core models are of particular interest in a data exchange setting [11]. Formally, we define them as follows.

Definition 2.7 (Embeddings and core models)**.** Let $\mathcal{I}$ be an interpretation. A null substitution $\nu\colon \text{nulls}(\mathcal{I}) \to \mathbf{N}$ is called

- a *pre-embedding* of $\mathcal{I}$ if $\mathcal{I}\nu \subseteq \mathcal{I}$.
- an *embedding* of $\mathcal{I}$ if it is a pre-embedding, injective and if $p(\bar{t})\nu \in \mathcal{I}$ implies that $p(\bar{t}) \in \mathcal{I}$.

An interpretation $\mathcal{I}$ is a *core* if every pre-embedding over $\mathcal{I}$ is also an embedding.

For a given interpretation $\mathcal{I}$, we define $\text{core}(\mathcal{I})$ to be an interpretation $\mathcal{J}$ such that $\mathcal{J} \subseteq \mathcal{I}$ and $\mathcal{J}$ is a core.

Example 2.4. Recall the knowledge base from Example 2.2 which admits the infinite chase sequence shown in Figure 2.1. This is not a core model since the null substitution $\nu = \{b_i \mapsto b_0 \mid i > 0\} \cup \{a_i \mapsto a_0 \mid i \geq 0\}$ is a pre-embedding which is not an embedding. This suggests that every fact in the chase result containing a_i or b_i with $i > 0$ is redundant. In contrast, using the Datalog-First strategy over the same knowledge base results in a core since the only possible pre-embedding is $\nu = \{a_0 \mapsto a_0\}$, which is clearly an embedding. △

An important aspect of core models is their relation to chase termination. The connection stems from the fact that no knowledge base that admits a finite universal model can at the same time have an infinite core [10].

Lemma 2.2. *Let $\mathcal{K}$ be a knowledge base. If there exists a finite universal model for $\mathcal{K}$ then there cannot exist a infinite core model for $\mathcal{K}$ as well.*

This implies that every procedure that guarantees to produce a core model for a given knowledge base $\mathcal{K}$ terminates if any finite universal model for $\mathcal{K}$ exists. This is achieved by the core chase [10]. As the name suggests, it is a variant of the chase which produces universal core models. In each derivation step, every unsatisfied match of all rules is satisfied in parallel and the resulting interpretation is reduced to a core.

Definition 2.8. Let $(\mathcal{I}_k) = \mathcal{D}, \mathcal{I}_1, \mathcal{I}_2, \ldots$ be a fair sequence of chase steps for a knowledge base $\mathcal{K} = \langle R, \mathcal{D} \rangle$. The sequence $(\mathcal{I}_k)$ is a *core chase* sequence if

- for every $k > 0$ we have $\mathcal{I}_k = \text{core}(\bigcup_{\rho \in R} \bigcup_{h \in \mathfrak{A}(\mathcal{I}_{k-1}, \rho)} \mathcal{I}^{\rho,h})$ such that $\mathcal{I}^{\rho,h}$ was obtained by applying ρ over $\mathcal{I}_{k-1}$ for the match h.

Deciding whether an interpretation is the core of another is complete for DP w.r.t. to the size of the given interpretations, meaning that it can be expressed as the intersection of a NP-complete and a coNP-complete problem [11]. It is worth emphasizing that this is with respect to the whole interpretation, which in practice may contain millions of entries. This is in contrast to the restricted chase where finding unsatisfied matches is on the second level of the polynomial hierarchy but only w.r.t. the size of the rule. Implementing the core chase as described above therefore does not lead to practical algorithm. Thankfully, there are situations where the comparatively cheap restricted chase may produce core models, which was recently observed by Krötzsch [18]. Roughly speaking, a restricted chase run produces a core model, if no application of a rule makes the earlier application of another rule redundant. We can capture this intuition formally through the concept of alternative matches.

Definition 2.9. Let $\mathcal{I}_a \subseteq \mathcal{I}_b$ be interpretations such that $\mathcal{I}_a$ was obtained by applying the rule $\rho = \varphi \to \psi$ for the extended match h. A match h^A is an *alternative match* for ρ and h over $\mathcal{I}_b$ if

- h and h^A agree on all universal variables and
- $h(\mathsf{vars}_\exists(\psi)) \neq h^A(\mathsf{vars}_\exists(\psi))$.

Example 2.5. Recall the restricted chase run shown in Example 2.1. At the start, we apply ρ_3 with the extended match $h = \{m \mapsto \text{“Electric Sheep”}, a \mapsto n\}$, introducing the facts $\mathsf{stars}(n, \text{“Electric Sheep”})$ and $\mathsf{famous}(n)$ into the database. In the third step, we derive $\mathsf{stars}(\text{“Alice”}, \text{“Electric Sheep”})$ from ρ_1 while the initial database already contains the fact $\mathsf{famous}(\text{“Alice”})$. Therefore, the substitution $h^A = \{m \mapsto \text{“Electric Sheep”}, a \mapsto \text{“Alice”}\}$ is an alternative match for ρ_3 and h over $\mathcal{I}_3$. The existence of h^A implies that h would have been a satisfied match for ρ_1 if its application was delayed after the third step. The introduction of new nulls by ρ_3 hence proved to be unnecessary. △

As shown in the above example, the presence of alternative matches in a chase sequence suggest that the application of some rule introduced redundant facts into the result. Conversely, chase runs without alternative matches therefore always produce core models.

Theorem 2.3 ([18, Theorem 2])**.** *If a restricted chase sequence is free of alternative matches then the resulting interpretation is a core.*

However, it is important to note that a core model can still result from restricted chase sequences that contain alternative matches. The absence of alternative matches is therefore only a sufficient condition for a particular chase run to result in a core. And even then, determining if a knowledge base admits a chase sequence free of alternative matches is undecidable [18]. For this reason Krötzsch introduces a way of detecting situations where the application of a rule may possibly lead to alternative matches [18].

Definition 2.10. A rule $\rho_1 = \varphi_1 \to \psi_1$ *restrains* a rule $\rho_2 = \varphi_2 \to \psi_2$, written $\rho_1 \prec^{\square} \rho_2$, if there are interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ such that

1. $\mathcal{I}_b$ was obtained by applying ρ_1 with the extended match h_1,
2. $\mathcal{I}_a$ was obtained by applying ρ_2 with the extended match h_2,
3. there exists an alternative match h^A for h_2 and ρ_2 over $\mathcal{I}_b$ and
4. h^A is not an alternative match for h_2 and ρ_2 over $\mathcal{I}_b \setminus \psi_1 h_1$.

Intuitively, a rule restrains another if applying the second before the first may lead to (new) alternative matches. In the rule set from the introduction, for instance, we have that $\rho_1 \prec^{\square} \rho_3$. Violating this restraint by applying the rules ρ_3 before ρ_1 causes the alternative matches in the run depicted in Example 2.1.

We slightly deviate from the definition provided in the original paper which states

4. there are no alternative matches for h_2 and ρ_2 over $\mathcal{I}_b \setminus \psi_1 h_1$

as the last condition. This allows a rule ρ_1 to restrain a rule ρ_2 even if ρ_1 never introduces the first alternative match w.r.t. ρ_2. The definition stated here more closely represents our goal of avoiding alternative matches, which is independent of whether they already occurred earlier in the run. The relaxed version of the definition also simplifies the computation of determining the existence of restraint reliances between rules, as we will describe in more detail in Chapter 3.

Note that it is also possible for a rule to restrain itself, which can happen for two reasons. For one, the application of a rule might immediately produce an alternative match over the resulting interpretation. Another possibility is that an additional application of a rule introduces an alternative match w.r.t. to an earlier application of the same rule. We show an example for either case.

Example 2.6. Consider the following two rules.

$$a(x) \to \exists v.\, b(x, v) \wedge c(x) \qquad (\rho_1)$$

$$d(x, y) \to \exists v, w.\, e(x, v, w) \wedge e(x, y, v) \qquad (\rho_2)$$

For the first example, we set $\tilde{\mathcal{I}}_a \coloneqq \{a(1), b(1, 2)\}$. Applying ρ_1 over $\tilde{\mathcal{I}}_a$ we obtain $\mathcal{I}_a \coloneqq \tilde{\mathcal{I}}_a \cup \{b(1, n), c(1)\}$. This directly results in the following alternative match $\{x \mapsto 1, v \mapsto 2\}$ over $\mathcal{I}_a$. Setting $\mathcal{I}_b \coloneqq \mathcal{I}_a$ therefore satisfies Definition 2.10. In the second case, we set $\mathcal{I}_a \coloneqq \{d(1, 2), e(1, n_v, n_w), e(1, 2, n_v)\}$, which results from applying the second rule over $\{d(1, 2)\}$. At this point, no alternative match is present. However, applying ρ_2 once more but this time over $\tilde{\mathcal{I}}_b \coloneqq \mathcal{I}_a \cup \{d(1, n_v)\}$ results in $\mathcal{I}_b \coloneqq \tilde{\mathcal{I}}_b \cup \{e(1, m_v, m_w), e(1, n_v, m_w)\}$. The corresponding alternative match w.r.t. the first application is $\{x \mapsto 1, y \mapsto 2, v \mapsto n_v, w \mapsto m_w\}$. △

Given two applicable rules ρ_1 and ρ_2 with $\rho_1 \prec^{\square} \rho_2$, a chase algorithm should preferably satisfy ρ_1 first in order to prevent potential alternative matches. However, this alone might not guarantee a core model, since ρ_1 could become applicable again at some later point in the chase run. To account for that, we also consider a second type of reliance, called positive reliances, which was defined by Deutsch, Nash, and Remmel [10]. Simply put, a rule ρ_2 positively relies on another rule ρ_1, if applying ρ_1 might enable a match for ρ_2. In such cases we also say that the ρ_1 *triggers* ρ_2.

Definition 2.11. A rule $\rho_2 = \varphi_2 \to \psi_2$ *positively relies* on a rule $\rho_1 = \varphi_1 \to \psi_1$, written $\rho_1 \prec^{+} \rho_2$, if there are interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ and a substitution h_2 such that

1. $\mathcal{I}_b$ is obtained from $\mathcal{I}_a$ by applying ρ_1 for the extended match h_1,
2. h_2 is an unsatisfied match for ρ_2 on $\mathcal{I}_b$ and
3. h_2 is not a match for ρ_2 over $\mathcal{I}_a$.

For every rule ρ in a given rule set, we can use positive and restraint reliances to obtain the set of all rules that need to be applied before ρ itself in order to avoid violating a restraint reliance during a chase run.

Definition 2.12. Let R be a set of rules and $\rho \in R$. Then the *downward closure* of ρ, written $\rho{\downarrow^{\square}}$, is a set containing all rules $\rho' \in R$ such that ρ' $((\prec^{+})^{*} \circ \prec^{\square})^{+}$ ρ where $\circ$ is the composition of relations, $+$ the transitive and $*$ the reflexive-transitive closure.

Definition 2.13. A set of rules R is *core-stratified* if $\rho \notin \rho{\downarrow^{\square}}$ for every $\rho \in R$.

Example 2.7. The rule set given in Example 1.1 is core-stratified. We obtain the downward closures $\rho_1{\downarrow^{\square}} = \rho_2{\downarrow^{\square}} = \emptyset$ and $\rho_3{\downarrow^{\square}} = \{\rho_1\}$. △

Core-stratified rule sets allow for a rule application strategy where restraining rules are exhaustively applied before the rules they restrain. The resulting sequence of chase steps therefore cannot contain any alternative matches, which implies that the result is a core. However, a sequence of chase steps produced in this way might not be fair. In order to show that core-stratified rule sets always admit a (fair) restricted chase sequence without alternative matches, Gerlach introduced the concept of a transfinite chase [13].

Definition 2.14. Let $\bar{R} = R_1, \ldots, R_n$ be a list of rule sets and $\mathcal{D}$ any arbitrary database. A *transfinite chase sequence* for $\mathcal{K}$ and $\bar{R}$ is a sequence of interpretations $\mathcal{I}^0, \mathcal{I}^1_\infty, \ldots, \mathcal{I}^n_\infty$ such that

- $\mathcal{I}^0 = \mathcal{D}$ and
- $\mathcal{I}^j_\infty$ is the result of the restricted chase on $\langle R^{\leq j}, \mathcal{I}^{j-1}_\infty \rangle$ for all $j \in [n]$

where $R^{\leq j} \coloneqq \bigcup_{1 \leq i \leq j} R_i$. The *result* of a transfinite chase sequence is the interpretation $\mathcal{I}^n_\infty$. Furthermore, a transfinite chase sequence is *infinite* when its result contains an infinite amount of facts.

The transfinite chase is a variant of the chase which essentially strings together multiple runs of the restricted chase. Each run takes as input the result of the previous one and extends the sets of rules by set which comes up next in the provided list of rule sets.

Definition 2.15 (Partitionings)**.** Let R be a set of rules. We call a series of pairwise disjoint sets $R_1, \ldots, R_n$ with $R = \bigcup_{i \in [n]} R_i$ an *ordered partitioning* of R. Furthermore, an ordered partitioning is a

- *restrained partitioning* if $R^{\geq i} \cap \rho{\downarrow^{\square}} = \emptyset$ for every $i \in [n]$ and $\rho \in R_i$.
- *relaxed restrained partitioning* if $R^{\geq i} \cap \rho{\downarrow^{\square}} = \emptyset$ for every $i \in [n-1]$ and $\rho \in R_i$.

Restrained partitionings can only be obtained from core-stratified rule sets while the relaxed version can be constructed for arbitrary rule sets. Using the transfinite chase on a restrained partitioning enforces the rule application order implied by the downward closures of the rules. This leads to a transfinite chase sequence without alternative matches.

Lemma 2.4 ([13, Lemma 4.8])**.** *Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base and $\bar{R}$ a restrained partitioning of R. Any transfinite chase sequence for $\mathcal{K}$ and $\bar{R}$ does not contain any alternative matches.*

It should be noted that the transfinite chase does not directly correspond to consecutively executing multiple restricted chases one after another, because it admits its sub-sequences to be infinite. However, for core-stratified rule sets an equivalent restricted chase sequence always exists.

Lemma 2.5 ([13, Lemma 4.10])**.** *Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base with a core-stratified rule set, $\bar{R}$ a restrained partitioning of R and $\mathcal{I}_\infty^n$ be the result of a transfinite chase sequence for $\mathcal{K}$ and $\bar{R}$. There exists a restricted chase sequence for $\mathcal{K}$ with the same result and using the same unsatisfied matches in its construction.*

Lemma 2.4 and Lemma 2.5 together imply that for any knowledge base $\mathcal{K}$ containing a core-stratified rule set there exists a restricted chase run for $\mathcal{K}$ without alternative matches as well. For rule sets that are not core-stratified Gerlach proposed the *hybrid chase*. This variant works similarly to the transfinite chase when performed on a relaxed partitioning of the rule sets but uses a core or eam chase in the last partition [13].

2.4 The Rule Engine VLog

VLog is a rule engine capable of reasoning with existential rules [23, 24, 9]. It is able to perform a full materialization of the implicit facts contained in a knowledge base by implementing a version of the restricted chase. VLog differs from other reasoners in its vertical or columnar approach for storing derived facts. This enables optimizations which reduce memory consumption and lead to competitive run times.

VLog uses different data structures for storing facts that are present in the initial database than for storing facts that are inferred during the chase. At the user-level, this is represented by a separation of the predicate names into EDB and IDB predicates. All tables corresponding to EDB-predicates together form the EDB-layer, which is loaded from disk into memory at the start and remains unchanged during the materialization. EDB-predicates may therefore not be used in the head of a rule. Information from the EDB-layer is extracted by asking conjunctive queries using established approaches [21].

Facts are stored in tables that use a columnar layout, meaning that each column occupies a continuous chunk of memory [23]. Additionally, entries in a table are sorted using the lexicographical order of the numerical values each constant or null is assigned to. Sorting allows the table to be used in merge joins, which is a very efficient type of join especially for large tables.

The materialization is performed based on the 1-parallel chase using a Datalog-First selection strategy. The computation on every derivation step can be divided into two parts. First, finding all matches over the current interpretation and second, checking which of them are unsatisfied. We can calculate the set of all matches for a rule by joining together every table corresponding to atoms contained in the rule's body. However, such an approach would lead to the derivation of many duplicate facts since every repeated application of a rule would recompute all of the matches used in previous steps. Instead, VLog uses the semi-naive evaluation strategy which avoids doing the same computation twice by keeping track of the step number during which a fact was derived [1]. We write Δ_p^i for the set of facts derived at step i for the predicate p. Furthermore, we define $\Delta_p^{[i,j]} \coloneqq \bigcup_{i \leq k \leq j} \Delta_p^k$ as the set of facts that were derived between step i and j for $i < j$. Assume that the rule $\rho = \varphi \to \psi$ with

$$\varphi = e_1(\bar{s}_1) \wedge \cdots \wedge e_m(\bar{s}_m) \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n)$$

is selected in the current step $i+1$ and has been applied the last time in step $j < i+1$ where e_k and $p_{k'}$ are EDB and IDB predicates respectively. Then the set of all matches is computed by performing the following joins.

$$\begin{aligned}\mathsf{tmp} = \bigcup_{\ell \in [n]} (e_1 \bowtie \cdots \bowtie e_m) \bowtie \Delta_{p_1}^{[0,i]} \bowtie \cdots \bowtie \Delta_{p_{\ell-1}}^{[0,i]} \\ \bowtie \Delta_{p_\ell}^{[j,i]} \bowtie \Delta_{p_{\ell+1}}^{[0,j-1]} \bowtie \cdots \bowtie \Delta_{p_n}^{[0,j-1]}\end{aligned}$$

To filter satisfied matches, VLog partially instantiates all of the head atoms and joins them with the tables of the already existing facts.

Although the columnar storage used is well suited for the operations describes above, updating the tables can be expensive when compared to traditional row-based layouts. VLog avoids this problem by storing each of the facts from Δ_p^i in a separate table. We will refer to these tables as *predicate-blocks*. However, splitting the facts into multiple blocks limits the effectiveness of the merge join,

since now multiple smaller join operations are required for every block instead of just one large join thatwould cover the whole table. For computing matches, this problem is alleviated by concatenating multiple predicate-blocks into a temporary table if needed, on which the joins are then performed. These are deleted after each step. If the number of predicate-blocks passes a certain threshold, all blocks are collapsed into one. Although these techniques improve the time needed for joining the tables, concatenating the blocks may still take a considerable amount of time itself. VLog therefore provides many optimizations that aim to exclude predicate-blocks from being considered, leaving only one per predicate in the ideal case.

We observe that the order in which rules are applied in may influence the amount of predicate-blocks that are introduced during the materialization.

Example 2.8. Consider again the restricted chase run that was used in Example 2.1. Using the same order but with the semi-naive evaluation described above, we obtain the following predicate-blocks for the relation costar from applying ρ_2 two times: $\Delta^2_{\text{costar}} = \{\text{costar}(n, n, \text{“Electric Sheep”}))\}$ and

$$\begin{aligned}\Delta^4_{\text{costar}} = \{&\text{costar}(\text{“Alice”}, \text{“Alice”}, \text{“Electric Sheep”}),\\ &\text{costar}(\text{“Alice”}, n, \text{“Electric Sheep”})\\ &\text{costar}(n, \text{“Alice”}, \text{“Electric Sheep”})\}.\end{aligned}$$

Every subsequent rule application that uses the predicate costar in its body, would therefore need to form the union of both blocks. In contrast, consider a situation where ρ_1 was exhaustively applied before ρ_2. Here, only one predicate-block, which contains all of the facts in Δ^2_{costar} and Δ^4_{costar}, would have been derived. $\triangle$

As shown in the example above, delaying the application of a rule may prevent VLog from performing redundant computations. As we will see in Chapter 4, we can use the positive reliance relation to determine an order that minimizes the amount of times a rule is applied and therefore the amount of predicate-blocks that are introduced. Note that in the introductory example, ρ_2 positively relies on ρ_1. The same technique can be used to reduce the total number of times the EDB-tables of the body of a rule have to joined, which is an operation performed upon every rule application.

Chapter 3

Computation of the Reliance Relationship

In this chapter, we provide algorithms for checking the existence of a positive or a restraint reliance between two given rules ρ_1 and ρ_2. The algorithms are based on unpublished work from Larry González. Although each type of reliance serves a very different purpose, they are both defined in a similar way. This allows us to abstract major parts of the required functionality into subroutines used by both algorithms. Sections 3.1 and 3.2 of this chapter therefore outline the general strategy used in both cases as well as some common notions and algorithms. We then provide the parts unique for each relationship separately in sections 3.3 and 3.4. In the last section, we prove the correctness of the described procedures.

3.1 General Strategy

Looking at the definitions of positive and restraint reliances, we notice an important similarity, which will allow us to use a common method for computing both. In each case, the reliance relationship is predicated upon the existence of two interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ such that the application of the first rule obtaining $\mathcal{I}_b$ introduces some new facts that "complete" a homomorphism. For positive reliances, the completed homomorphism represents an unsatisfied match that is made possible by applying the first rule. In the case of restraint reliances the homomorphism is an alternative match introduced by the application of the first rule.

To decide whether such a situation can possibly occur, we search for a subset of the body or head of ρ_2, for positive and restraint reliances respectively, which unifies with a subset of the head of ρ_1. The existence of the corresponding unifier shows that there is a variable assignment that maps the respective part of ρ_2 to some possible product of applying ρ_1. For every unifiable subset, we use a certain unifier as a variable assignment for the atoms in ρ_1 and ρ_2 to construct two representative interpretations $\mathcal{I}_a$ and $\mathcal{I}_b$. These interpretations are then used to test each requirement of the respective defintion. If the created interpretations satisfy each condition, then we can conclude that the tested relationship indeed holds between the two rules. Conversely, we need to make sure that a failure to satisfy the definition for the representative interpretations truly implies that this is impossible in general, that is, for every possible pair of interpretations. As we will see later, this follows from the fact that the representative interpretations were constructed by using a unifier that can be transformed into any other unifier. We refer to such unifiers as the most *flexible*. In summary, we use the following general strategy for both reliances:

- Iterate over all unifiable subsets of the head/body of ρ_2 and head of ρ_1 computing the most flexible unifier η; for every such unifier:
  - Construct $\mathcal{I}_a$ and $\mathcal{I}_b$ from the atoms of both rules and η
  - Check if the definition of the respective reliance is satisfied

The exact construction of the interpretations and the checks that need to be performed of course depend on the reliance that is being tested. But in each case the tests mostly consists of performing model checking as a way to verify that both of the rules are applicable. With this in mind, iterating over each of the possible subsets and performing multiple expensive operations each time seems costly. Although we cannot alleviate this concern in every case since the underlying problem is Σ_2^P-complete, we do not always need to consider all of the possibilities. Some of the time, the constructed interpretations from one unifer violate the definition in such a way that cannot be "repaired" by considering a more specific unifier one would obtain by unifying a larger subset. We take advantage of that by iterating through the subsets from smallest to largest in an attempt to terminate certain computation paths early. This procedure is described in Algorithm 3.1.

But before explaining the algorithm, we need to establish some common notions. In the following, we will assume that the body and head atoms φ_i and ψ_i from

both rules $\rho_1 = \varphi_1 \to \psi_1$ and $\rho_2 = \varphi_2 \to \psi_2$ are globally available in each algorithm. In addition, the atoms in the head and body of the rules are assumed to be in some (arbitrary) order such that we can index into them. So for example, $\psi_1[2]$ refers to the second atom in the head of the first rule. Furthermore, we assume there to be a variable substitution $\omega_\exists \colon \mathbf{V}_\exists \to \mathbf{T}$ that assigns a unique null to each existential variable. We will interpret $\omega_\exists$ as a function that assigns the existential variables to the nulls introduced by applying ρ_i over a given interpretation. In this context, we only consider one single application of each rule. Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms. An *atom mapping* from $\mathcal{A}$ to $\mathcal{B}$ is a surjective function $m \colon \mathcal{A} \to \mathcal{B}$.

Definition 3.1. Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms and m a atom mapping from $\mathcal{A}$ to $\mathcal{B}$. A variable substitution $\eta \colon \mathbf{V} \to \mathbf{T}$ is

- a *unifier* of m, if $p(\bar{s})\eta = q(\bar{t})\eta$ for every $p(\bar{s}) \mapsto q(\bar{t}) \in \mathsf{graph}(m)$.
- the *most flexible unifier*, if it is a unifier such that for every unifier σ of m there exists a total substitution τ with $\sigma = \tau \circ \eta$.

Atom mappings are used as input for the unification algorithm, specifying which atom from $\mathcal{A}$ should unify with which atom of $\mathcal{B}$. The final reliance algorithms will start with $m = \emptyset$ and systematically extend it until every possible mapping is covered, finishing computation early when appropriate.

Algorithm 3.1: extend

Input: AtomSet $\mathcal{D}$, AtomMapping m, int index
Output: true iff the atom mapping can be extended successfully

```
1  for i ∈ {index, ..., |D|} do
2      for j ∈ {1, ..., |ψ1|)} do
3          m' := m ∪ {D[i] ↦ ψ1ω∃[j]};
4          if η := unify(m') then
5              if check(m', η, index + 1) then
6                  return true;
7              end
8          end
9      end
10 end
11 return false;
```

Algorithm 3.1 serves as the backbone in the computation for both reliance relationships. Its main task is to facilitate a depth-first search through all of the possible atom mappings. Given an atom mapping m, a set $\mathcal{D}$ and and index, the algorithm iterates over each possible extension m, assigning one additional element from the domain $\mathcal{D}$ to one element of $\psi_1\omega_\exists$. For positive reliances the domain is

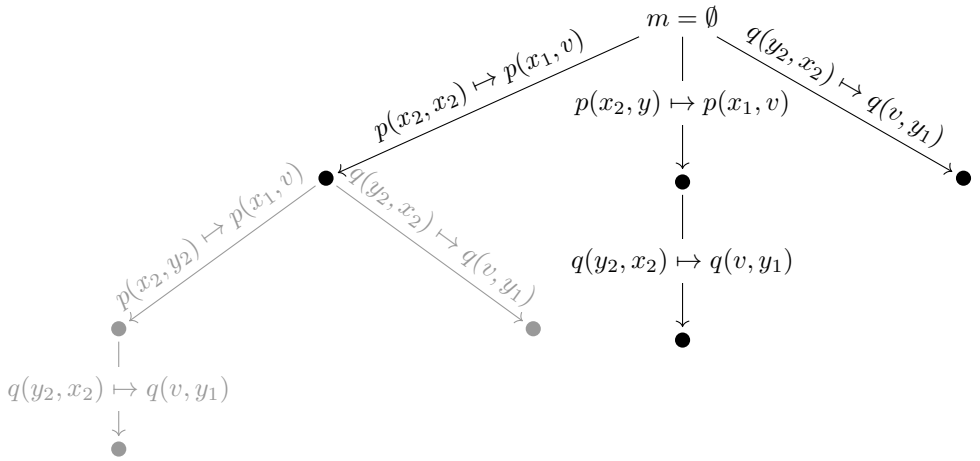


Figure 3.1: Example of a computation tree for testing positive reliances

the body while for restraint reliances the domain is the head of the second rule. We avoid examining the same atom mapping twice by extending a mapping only by domain elements whose index is greater or equal than the provided index as seen in Line 1. Each call to `extend` on each subsequent layer increases the index by one. Line 4 calls the function `unify` on the new atom mapping m' computed in the previous line. If successful, the most flexible unifier of m will be assigned to the variable η, which is passed to the function `check`. There, we construct the interpretations from η and verify each condition of the definition of the computed reliance. If it is satisfied then we can terminate early and return true. If not then `extend` is called from within `check` but only if it is possible that an extended atom mapping proves the relationship to hold. The implementation of `check` depends on which reliance is being computed. Each version is provided in sections 3.3 and 3.4. If no computation path is successful then we return false.

As an example consider the following two rules.

$$a(x_1, y_1) \to \exists v.\, p(x_1, v) \wedge q(v, y_1) \qquad (\rho_1)$$

$$p(x_2, x_2) \wedge p(x_2, y_2) \wedge q(y_2, x_2) \to \exists w.\, b(w, w) \qquad (\rho_2)$$

Figure 3.1 shows the computation tree for testing $\rho_1 \prec^+ \rho_2$. Each node represents the atom mapping obtained by combining each pair of atoms along its path. Since we are testing for a positive reliance, the atom mappings assign atoms from the body of the second rule to the head of the first rule. First note how every path corresponds to a unique atom mapping. Furthermore, we are able to exclude certain calculation paths from consideration. These are greyed out in the illustration.

In this case we know that no fact produced from the atom $p(x_1, v)$ in ρ_1 can ever match with $p(x_2, x_2)$ from ρ_2. This is because v is an existential variable and therefore introduces a fresh null into the database. For this reason, it cannot be equal to the term supplied for x_2 while applying ρ_2. This holds true in every atom mapping which contains $p(x_2, x_2) \mapsto p(x_1, v)$. This allows us to exclude all computation paths that would otherwise include this pair in their atom mapping.

3.2 Unification

Unification is a well-studied problem ever since it was first introduced in the context of resolution by Robinson [22]. Many improvements to the original algorithm have been presented [17]. However, most of the difficulty of the problem stems from the fact that terms may include function symbols. Since the fragment of first order logic used here does not contain any function symbols, unifying an atom mapping becomes much more straightforward. In this section, we present a simple linear-time unification algorithm. Given an atom mapping m, it will return the most flexible unifier of m. This property is crucial for showing the correctness of the reliance algorithms.

The complete procedure is shown in Algorithm 3.2. It maintains a graph G and a partial function $\nu\colon \mathbf{V} \to \mathbf{T}$, which we will call a variable assignment. Both start out empty. The basic idea is that variables connected in G are supposed to be assigned to the same term by every unifier. And likewise, any unifier should assign a variable x in $\mathrm{dom}(\nu)$ to $\nu(x)$. After the initialization step, the algorithm iterates over each pair of atoms contained in m. Line 16 verifies that the atoms are compatible, in the sense that they share the same predicate and have the same arity. Passing this check, we iterate over each possible term position in each pair. In Line 18 we define c_s to be the constant/null assigned to s_i by ν, or alternatively, c_s is assigned to s_i if s_i is a constant or null itself. We define c_t analogously in the next line. We leave c_s and c_t undefined if both terms are variables that have not been assigned to a value yet. In the case where they are constants/nulls or assigned to such, we check if they are equal in Line 20. If not then we know that m is not unifiable and immediately abort. If at least one of the terms is a variable we continue in Line 24. Here, we test whether one of the terms is a constant/null or assigned to such. If so then we need to update ν by assigning each connected variable to c_s or c_t. This is accomplished by the function `assign` which performs a depth-first-search in G, assigning each connected variable to the given constant/null.

Algorithm 3.2: unify

Input: AtomMapping m
Output: Most flexible unifier η or 'not unifiable'

Function `get`(*VariableAssignment* ν, *Term* t)**:**
  if $t \in \mathbf{C} \cup \mathbf{N}$ **then return** t ;
  else
    if $t \in dom(\nu)$ **then return** $\nu(t)$;
    else return *undefined*;
  end

Function `assign`(*Graph* G, *VariableAssignment* ν, *Variable* x, *Term* c)**:**
  if $x \in dom(\nu)$ **then return**;
  $\nu(x) \coloneqq c$;
  foreach $y \in succ_G(x)$ **do**
    $\nu \coloneqq \mathsf{assign}(G, \nu, y, c)$;
  end

$G \coloneqq \emptyset$;
$\nu \coloneqq \emptyset$;

foreach $p(s_1, \ldots, s_n) \mapsto q(t_1, \ldots, t_m) \in graph(m)$ **do**
  if $p \neq q$ *or* $n \neq m$ **then return** *not unifiable*;
  for $i \in [n]$ **do**
    $c_s \coloneqq \mathsf{get}(\nu, s_i)$;
    $c_t \coloneqq \mathsf{get}(\nu, t_i)$;
    if c_s *and* c_t *are defined* **then**
      if $c_s = c_t$ **then skip**;
      else return *not unifiable*;
    end
    if c_s *or* c_t *is defined* **then**
      if $s_i \in \mathbf{V}$ *and* $s_i \notin dom(\nu)$ **then** $\nu \coloneqq \mathsf{assign}(G, \nu, s_i, c_t)$;
      else if $t_i \in \mathbf{V}$ *and* $t_i \notin dom(\nu)$ **then** $\nu \coloneqq \mathsf{assign}(G, \nu, t_i, c_s)$;
    end
    if $s_i \in \mathbf{V}$ *and* $t_i \in \mathbf{V}$ **then**
      $G \coloneqq G \cup \{(s_i, t_i), (t_i, s_i)\}$;
    end
  end
end

$\eta \coloneqq \nu\omega_G\omega_\forall\omega_\exists$;
return η;

Line 28 checks if both terms are variables. In this case they must be assigned to the same value in every unifier. We therefore add two edges between the two variables in G. Lastly, we compute η in Line 33, which is the unifier we return. Besides $\omega_\exists$, we also assume there to be two additional substitutions ω_G and $\omega_\forall$. The former assigns a unique constant to each strongly connected component in G. The latter assigns each universal variable to a unique constant different from the ones in ω_G. For reasons which will become apparent later, we require that $\omega_\forall$ and ω_G do not assign any variable to a constant used in ρ_1 or ρ_2.

From this description it should be clear that any returned substitution is a unifier. We will now show that if m is unifiable, then Algorithm 3.2 returns the most flexible unifier. For that, we need some auxiliary results, which show that G and ν have the correct semantics.

Lemma 3.1. *Let m be an atom mapping from $\mathcal{A}$ to $\mathcal{B}$ and σ be a unifier of m. Furthermore, let G be the graph computed in Algorithm 3.2 when called on m. Then reach$_G(x, y)$ implies $\sigma(x) = \sigma(y)$ for every $x, y \in \mathbf{V}$.*

Proof. If reach$_G(x, y)$ then either $x = y$, in which case the statement holds trivially, or there exists a path $v_1, \ldots, v_n$ in G with $v_1 = x$ and $v_n = y$. It follows from the way that G is constructed that if two variables $v', v'' \in \mathbf{V}$ are connected in G then there exists an assignment $p(s_1, \ldots, s_n) \mapsto p(t_1, \ldots, t_n) \in \text{graph}(m)$ and $i \in [n]$ such that $v' = s_i$ and $v'' = t_i$ or vice versa. The fact that σ is a unifier of m implies that $\sigma(v') = \sigma(v'')$. Consequently, we have $\sigma(v_1) = \cdots = \sigma(v_n)$ and therefore $\sigma(x) = \sigma(y)$. □

Lemma 3.2. *Let m be an atom mapping from $\mathcal{A}$ to $\mathcal{B}$ and σ be a unifier of m. Furthermore, let ν be the variable assignment computed in Algorithm 3.2 when called on m. Then $\nu(x) = \sigma(x)$ for every $x \in \text{dom}(\nu)$.*

Proof. Let G be the graph computed in Algorithm 3.2 when called on m. Furthermore, let $x \in \text{dom}(\nu)$. Then there exists a variable $y \in \mathbf{V}$ with the following properties:

- x is reachable from y in G and
- there is an assignment $p(s_1, \ldots, s_n) \mapsto p(t_1, \ldots, t_n) \in \text{graph}(m)$ and $i \in [n]$ such that $s_i = y$ and $t_i = \nu(x)$ or vice versa.

Since σ is a unifier of m, we obtain $\sigma(y) = \nu(x)$. From Lemma 3.1 we get $\sigma(x) = \sigma(y)$ and therefore $\nu(x) = \sigma(x)$. □

With these results in place we will prove that Algorithm 3.2 returns the most flexible unifier η meaning that it can be morphed into every other unifier σ with a total substition τ by concatinating τ and η. Here, we show an additional property that will become useful in later proofs, namely that τ can be selected such that $\tau(t) = t$ for every $t \in \mathsf{terms}(\rho_1) \cup \mathsf{terms}(\rho_2)$.

Proposition 3.3. *Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms. If $\mathcal{A}$ and $\mathcal{B}$ are unifiable then there exists a set of atoms $\mathcal{B}' \subseteq \mathcal{B}$ and an atom mapping m from $\mathcal{A}$ to $\mathcal{B}'$ such that Algorithm 3.2 when called on m returns the most flexible unifier η between $\mathcal{A}$ and $\mathcal{B}'$. Furthermore, for every unifier σ of m there exists a total substitution τ with $\sigma = \tau \circ \eta$ and $\tau(t) = t$ for all $t \in$ terms$(\rho_1) \cup$ terms(ρ_2).*

Proof. Assume that there is a unifier σ of $\mathcal{A}$ and $\mathcal{B}$. For each atom $p(\bar{s}) \in \mathcal{A}$ there must be an atom $q(\bar{t}) \in \mathcal{B}$ with $p(\bar{s})\sigma = q(\bar{t})\sigma$. Hence, we can pick an atom mapping m where $m(p(\bar{s})) = q(\bar{t})$ and the set $\mathcal{B}'$ to $\mathsf{im}(m)$.

First, we will show that calling Algorithm 3.2 on m does not return 'not unifiable'. Failing the check in Line 16 would imply that m assigns an atom $p(\bar{s})$ to $q(\bar{t})$ with either $p \neq q$ or $|\bar{s}| \neq |\bar{t}|$. In either case, σ cannot be a unifier for m. Therefore, the only way to return 'not unifiable' left is if $c_s \neq c_t$ in Line 20. We know that either s_i or t_i has to be a variable. Because if not, then $s_i\sigma = s_i = c_s \neq c_t = t_i = t_i\sigma$ would violate the assumption that σ is a unifier. Suppose then, that s_i is a variable and t_i is not. Since σ is a unifier, we find that $\sigma(s_i) = t_i = c_t$. We assumed that $c_s = \nu(s_i) \neq c_t = t_i$. But Lemma 3.2 implies that $\nu(s_i) = \sigma(s_i)$, which is a contradiction. A similar argument can be made in the case when t_i is a variable and s_i is not. So let $s_i, t_i \in \mathbf{V}$. We have that $\sigma(s_i) = \sigma(t_i)$ but $\nu(s_i) = c_s \neq c_t = \nu(t_i)$. Using Lemma 3.2 again, we get that $\sigma(s_i) = \nu(s_i)$ and $\sigma(t_i) = \nu(t_i)$, which contradicts the earlier statement. Thus, the returned substitution by Algorithm 3.2 η is a unifier of m.

We will now show that η is the most flexible unifer by proving the existence of a total substitution τ with $\sigma = \tau \circ \eta$. For that, we define $\tau(\eta(x)) \coloneqq \sigma(x)$ for every $x \in \mathbf{V}$ and $\tau(t) = t$ for every $t \in \mathbf{T} \setminus \mathsf{im}(\eta)$. It is clear that $\sigma = \tau \circ \eta$. But we need to show that τ is well-defined. For that, assume $\eta(x) = \eta(y)$ for some $x, y \in \mathbf{V}$. This is only possible if $x, y \in \mathsf{dom}(\nu)$ and $\nu(x) = \nu(y)$, or if $\mathsf{reach}_G(x, y)$, since ω_G, $\omega_\forall$ and $\omega_\exists$ assign a unique terms to every variable. In the first case we

can use Lemma 3.2 and in the second case we can use Lemma 3.1 to show that $\sigma(x) = \sigma(y)$. This means that τ is well defined.

For the total substition τ defined above we now prove that $\tau(t) = t$ for every $t \in \text{terms}(\rho_1) \cup \text{terms}(\rho_2)$. Assume that $t \in \text{terms}(\rho_1) \cup \text{terms}(\rho_2)$ and $t \in \text{im}(\eta)$. This implies that there is a $x \in \mathbf{V}$ such that $t = \eta(x)$. But since ω_G, $\omega_\forall$ and $\omega_\exists$ do not assign any variable to a term in ρ_1 or ρ_2 we can infer that $x \in \text{dom}(\nu)$ with $\nu(x) = t$. From Lemma 3.2 it follows that $\sigma(x) = t$ and therefore that $\tau(t) = t$. If $t \notin \text{im}(\eta)$ then $\tau(t) = t$ by definition. □

The complexity of the unifiation algorithm is linear in the size of m. Both for-loops iterate over each term in m. In the for-loop each operation takes up constant time except for `assign`. But since every variable can only be assigned at most one time, the depth-first search performed by `assign` can only ever be executed once on each strongly connected component in G. Both the number of vertices and the number of edges is bounded linearly by the size of m. Hence, the overall algorithm only takes a linear amount of time.

3.3 Positive Reliances

We now combine the procedures from the previous sections into an algorithm that detects positive reliances. Algorithm 3.3 serves as the entry point. It receives two rules ρ_1 and ρ_2 and returns true if $\rho_1 \prec^+ \rho_2$ and false otherwise. First, we uniformly replace each of the variables in ρ_1 and ρ_2 in Line 1 such that no variable in ρ_1 is contained in ρ_2. This allows us to use a single substitution as a match for both rules without the worry of conflicting assignments. Line 2 returns the result of calling Algorithm 3.1 on the body atoms of the second rule φ_2, the empty atom mapping $\emptyset$ and the index 1.

Algorithm 3.3: positive reliance

Input: Rules $\rho_1 = \varphi_1 \to \psi_1$, $\rho_2 = \varphi_2 \to \psi_2$
Result: true iff $\rho_1 \prec^+ \rho_2$
1 Uniformly replace all variables in ρ_1, ρ_2 such that both rules don't share a variable;
2 **return** *extend*(φ_2, $\emptyset$, 1) ;

It is worth pointing out the special case where $\rho_1 = \rho_2 = \rho$. For positive reliances, we can treat it the same way as computing the existence of a positive reliance

between two distinct rules ρ and ρ', where ρ' is the same as ρ except for renamed variables.

Lemma 3.4. *Let $\rho = \varphi \to \psi$ be a rule. If $\rho \prec^+ \rho$ then $\rho \prec^+ \rho'$ where $\rho' = \varphi' \to \psi'$ is obtained from ρ by uniformly renaming every variable in ρ.*

Proof. Let $\nu\colon \mathsf{vars}(\rho) \to \mathsf{vars}(\rho')$ be a bijection such that $\varphi\nu = \varphi'$ and $\psi\nu = \psi'$. Furthermore, let $\mathcal{I}$ be any interpretation and $\mathcal{A} \in \{\varphi, \psi\}$. If h is a homomorphism from $\mathcal{A}$ to $\mathcal{I}$ then any variable substitution h' with $h'(x) = h(\nu(x))$ for $x \in \mathsf{vars}(\rho)$ is a homomorphism from φ' to $\mathcal{I}$. Also, if $\hat{h}$ existentially extends h, then $\hat{h}'$ with $\hat{h}'(x) = \hat{h}(\nu(x))$ existentially extends h'. The reverse of the above statements also holds when considering $\mathcal{A} = \{\varphi', \psi'\}$ and ν^{-1}. Hence, a match for ρ over $\mathcal{I}$ exists iff a match for ρ' over $\mathcal{I}$ exists and the former is satisfied over $\mathcal{I}$ iff the latter is.

If $\rho \prec^+ \rho$ there exist interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ and a variable substitution h_2 such that $\mathcal{I}_b$ was obtained by applying ρ, h_2 is an unsatisfied match for ρ over $\mathcal{I}_b$ and h_2 is not a match for ρ over $\mathcal{I}_a$. But then $\mathcal{I}_a$, $\mathcal{I}_b$ with $h_2'(x) = h_2(\nu(x))$ satisfies the definition as well and we obtain $\rho \prec^+ \rho'$. □

The only part missing now is to define the algorithm which constructs the interpretations from a given unifier and checks if they satisfy the requirements of Definition 2.11. This task is performed by Algorithm 3.4. It receives an atom mapping m, a unifier η and an index. The interpretations will be constructed in such a way that, if possible, η will be the unsatisfied match in the application of both rules. The algorithm starts by splitting the body of ρ_2 into two parts φ_{21} and φ_{22} in lines 1 and 2. The former is the part of the body of ρ_2 that will match with the new atoms produced by applying ρ_1. The atoms in $\varphi_{22}\eta$ together with the atoms in $\varphi_1\eta$ form the interpretation $\mathcal{I}_a$ in Line 9. In order for this to be a valid interpretation, it must not contain any new nulls introduced by the application of ρ_1. The relevant checks are performed in lines 3 and 6. Note that if Algorithm 3.2 assigns a null to any variable in φ_1, that it will do so in every extension of the current atom mapping. Atoms in φ_{22} on the other hand may move into φ_{21} on subsequent calls. Hence, we call into `extend` on Line 7 but simply return false on Line 4. The purpose of Line 10 is to check whether η can be existentially extended to a homomorphism from ψ_1 to $\mathcal{I}_a$. We justify the notation used here in Section 3.5. If η cannot be extended then it is an unsatisfied match for the first rule and we continue in Line 13. Here, we make sure that η is not an unsatisfied match for ρ_2. If this is not the case, we construct the interpretation $\mathcal{I}_b$ in Line 16. Lastly, we

Algorithm 3.4: check (positive)

Input: AtomMapping m, Unifier η, int index
Output: Returns true iff the given mapping is valid

```
1  φ21 ← dom(m);
2  φ22 ← φ2 \ φ21;
3  if {v ↦ t ∈ graph(η) | v ∈ vars(φ1), t ∈ N} ≠ ∅ then
4  |  return false;
5  end
6  if {v ↦ t ∈ graph(η) | v ∈ vars(φ22), t ∈ N} ≠ ∅ then
7  |  return extend(φ2, m, index);
8  end
9  I_a ← φ1η ∪ φ22η;
10 if I_a ⊨ ψ1η∀ then
11 |  return extend(φ2, m, index);
12 end
13 if φ2η ⊆ I_a then
14 |  return extend(φ2, m, index);
15 end
16 I_b ← I_a ∪ ψ1ω∃η;
17 if I_b ⊨ ψ2η∀ then
18 |  return false;
19 end
20 return true;
```

check if η is an unsatisfied match for ρ_2 over $\mathcal{I}_b$ in Line 17. Failing all of the above if-conditions implies that the constructed interpretations meets the requirements of Definition 2.11, in which case we return true.

3.4 Restraint Reliances

The algorithm for detecting a restraint reliance is constructed in a similar fashion. Algorithm 3.5 shows the main procedure. As with positive reliances, we uniformly replace all variables in both rules such that they do not share any variables. We then return the result of calling `extend` on the head of the second rule, the empty atom mapping and the index 1.

However, unlike with positive reliances, we cannot reduce the problem of verifying $\rho \prec^{\square} \rho$ to checking $\rho' \prec^{\square} \rho$ where ρ' is obtained by renaming variables from ρ. This is because alternative matches can be introduced by a single application of a rule and do not require ρ to be applicable a second time. To illustrate, consider the next example.

Algorithm 3.5: restraint

Input: Rules ρ_1, ρ_2
Result: $\rho_1 \prec^{\square} \rho_2$
1 Uniformly replace all variables in ρ_1, ρ_2 such that both rules don't share a variable;
2 **return** *extend(ψ_2, $\emptyset$, 1)*;

Example 3.1. Recall the rule from Example 2.6.

$$a(x) \to \exists v.\, b(x, v) \wedge t(x) \qquad (\rho)$$

$$a(y) \to \exists v.\, b(y, v) \wedge t(y) \qquad (\rho')$$

Let $\tilde{\mathcal{I}}_a = \{a(1), b(1, 2)\}$. Applying ρ results in $\mathcal{I}_a = \tilde{\mathcal{I}}_a \cup \{b(1, n_v)\}$ and introduces an alternative match $\{x \mapsto 1, v \mapsto 1\}$. Setting $\mathcal{I}_a = \mathcal{I}_b$ satisfies Definition 2.10 and hence $\rho \prec^{\square} \rho$. However, we have that $\rho' \not\prec^{\square} \rho$ because ρ' is satisfied over $\mathcal{I}_a$ and therefore cannot be applied to obtain a second interpretation $\mathcal{I}_b \supset \mathcal{I}_a$. △

For the sake of simplicity, we will disregard this case and provide a function `check` that assumes that both rules are applied. The corresponding procedure is shown in Algorithm 3.6. This time, the unifier η represents the alternative match introduced by applying the second after the first. The procedure starts by dividing the head of the second rule into two parts ψ_{21} and ψ_{22} in lines 1 and 2. The first part is the portion of the head of ρ_2 for which the application of ρ_1 will produce the alternative match. Existential variables in ψ_{22} that are not present in ψ_{11} are simply assigned to their corresponding nulls created during the application of ρ_2. In Line 9 we check whether ψ_{11} contains an existential variable. This ensures that at least one existential variable in ψ_2 is assigned to a term not present in $\omega_{\exists}(\text{vars}_{\exists}(\psi_2))$, which is necessary for η to be an alternative match. Algorithm 3.6 constructs three interpretations: $\tilde{\mathcal{I}}_a$, which represents the interpretation right before the application of ρ_2; $\mathcal{I}_a$, which results from applying ρ_2 over $\tilde{\mathcal{I}}_a$; and $\tilde{\mathcal{I}}_b$, which represents the interpretation immediately before applying ρ_1. The checks in lines 3 and 6 ensure that the constructed interpretations do not contain any nulls that are introduced by a later application of ρ_1 or ρ_2. The condition on Line 13 and the second condition on Line 18 verify that η is indeed an unsatisfied match for ρ_1 and ρ_2 over the corresponding interpretations. The first condition on Line 18 checks that the alternative match η is not already present in $\tilde{\mathcal{I}}_b$ before the application of ρ_1. If all checks are passed then we return true.

Algorithm 3.6: check (restraint)

Input: AtomMapping m, Unifier η, int index
Output: Returns true iff the given mapping is valid

1 $\psi_{21} \leftarrow \mathsf{dom}(m)$;
2 $\psi_{22} \leftarrow \psi_2 \setminus \psi_{21}$;
3 **if** $\{v \mapsto t \in \mathit{graph}(\eta) \mid v \in \mathbf{V}_\forall, t \in \mathbf{N}\} \neq \emptyset$ **then**
4 | **return** *false*;
5 **end**
6 **if** $\{v \mapsto t \in \mathit{graph}(\eta) \mid v \in \mathit{vars}_\exists(\psi_{22}), t \in \mathbf{N}\} \neq \emptyset$ **then**
7 | **return** *extend(ψ_2, m, index)*;
8 **end**
9 **if** $\mathit{vars}_\exists(\psi_{21}) = \emptyset$ **then**
10 | **return** *extend(ψ_2, m, index)*;
11 **end**
12 $\tilde{\mathcal{I}}_a \leftarrow \varphi_2\eta$;
13 **if** $\tilde{\mathcal{I}}_a \models \psi_2\eta_\forall$ **then**
14 | **return** *false*;
15 **end**
16 $\mathcal{I}_a \leftarrow \tilde{\mathcal{I}}_a \cup \psi_2\omega_\exists\eta$;
17 $\tilde{\mathcal{I}}_b \leftarrow \mathcal{I}_a \cup \varphi_1\eta \cup \psi_{22}\eta$
18 **if** $\psi_2\eta \subseteq \tilde{\mathcal{I}}_b$ *or* $\tilde{\mathcal{I}}_b \models \psi_1\eta_\forall$ **then**
19 | **return** *extend(ψ_2, m, index)*;
20 **end**
21 **return** *true*;

We are now able to offer a more detailed explanation as to why computing the original restraint relation provided by Krötzsch is more difficult. Recall that in Definition 2.10, the condition

- there is no alternative match for h_2 and ρ_2 over $\mathcal{I}_b \setminus \psi_1 h_1'$

was relaxed to

- h^A is not an alternative match for h_2 and ρ_2 over $\mathcal{I}_b \setminus \psi_1 h_1'$.

We could imagine an alternative description of Algorithm 3.6 that is based on the stricter defintion. It would replace the first condition on Line 18 with a condition that checks that no alternative matches over $\tilde{\mathcal{I}}_b$ exist. But such an approach would reject too often as seen in the next example.

Example 3.2. Consider the following two rules:

$$b(x) \rightarrow \exists v.h(x,v) \qquad (\rho_2)$$

$$h(y,z_1) \wedge d(z_1) \wedge h(y,z_2) \wedge e(z_2) \rightarrow \exists w_1,\ w_2.\, h(y,w_1) \wedge f(w_2) \qquad (\rho_1)$$

Assume that we want to check whether $\rho_1 \prec^{\square} \rho_2$. Since there is only one possbile way to unify the two heads, Algorithm 3.6 would be called once producing the following interpretation

$$\tilde{\mathcal{I}}_b = \{b(c_{xy}), h(c_{xy}, n_v), h(c_{xy}, c_{z_1}), d(c_{z_1}), h(c_{xy}, c_{z_2}), e(c_{z_2})\}$$

where c_* is the constant assigned in place of the variables in $*$ and n_v is a null. It is clear that there already is an alternative match over $\tilde{\mathcal{I}}_b$, for example $\{x \mapsto c_{xy}$, $v \mapsto c_{z_1}\}$, and hence the alternative version of Algorithm 3.6 would return false. But this is incorrect. Consider the interpretations $\mathcal{I}_a = \{b(1), h(1,n)\}$, which results from applying ρ_2 over $\{b(1)\}$ and $\tilde{\mathcal{I}}_b = \mathcal{I}_a \cup \{d(n), e(n)\}$. At this point, no alternative match for the application of ρ_2 exists, but ρ_1 is still applicable over $\tilde{\mathcal{I}}_b$ while also introducing an alternative match. Hence, $\rho_1 \prec^{\square} \rho_2$ does hold in the stricter defintion. △

The problem shown in Example 3.2 arises because Algorithm 3.6. does not take into account that ρ_1 might match with a product of applying ρ_2, which might avoid unwanted alternative matches. One possible way to fix this would be to unify the body of ρ_1 with the head of ρ_2 in similar way that was described for positive reliances. But this search would have to be done for every call of Algorithm 3.6.

3.5 Proving Correctness

After presenting the algorithms, we will now show that they are sound and complete. Proving soundness will come down to showing that the computed interpretations in each version of the function `check` do indeed satisfy the respective definitions of the tested reliance. The key property which we use to show completeness is the fact that the unifier computed Algorithm 3.2 is the most flexible one. This ensures that the constructed interpretations are the most general, in the sense that if they fail to meet the definition then no other pair of interpretations will.

We start by showing some auxiliary results. The first one justifies our use of the statement $\mathcal{I} \not\models \mathcal{A}h_\forall$ throughout the algorithms to mean that h cannot be existentially extended to a homomorphism from $\mathcal{A}$ to $\mathcal{I}$.

Lemma 3.5. *Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms, $\mathcal{I}$ an interpretation and $h\colon \mathbf{V} \to \mathbf{T}$ a homomorphism from $\mathcal{A}$ to $\mathcal{I}$. Then $\mathcal{I} \models \mathcal{B}h_\forall$ if and only if h can be existentially extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}$.*

Proof. If $\mathcal{I} \models \mathcal{B}h_\forall$ then there is some variable substitution h' such that $\mathcal{B}h_\forall h' \subseteq \mathcal{I}$. Since $h_\forall$ is applied first $h_\forall h'$ has to agree with h on all universal variables. Hence, $h_\forall h'$ existentially extends h and is a homomorphism from $\mathcal{B}$ to $\mathcal{I}$.

Assume on the other hand that there is a variable substitution $h' = h_\forall h'_\exists$, which extends h to a homomorphism from $\mathcal{B}$ to $\mathcal{I}$. Then $h'_\exists$ is a homomorphism from $\mathcal{B}h_\forall$ to $\mathcal{I}$. □

Lemma 3.6. *Let $\mathcal{A}$ and $\mathcal{B}$ be two sets of atoms. Moreover, let σ and η be some substitutions such that $\mathcal{A}\sigma \subseteq \mathcal{B}\eta$ and τ be a total substitution such that $\tau(t) = t$ for every $t \in$ terms$(\mathcal{A}) \cup$ terms$(\mathcal{B})$. It follows that $\mathcal{A}(\tau \circ \sigma) \subseteq \mathcal{B}(\tau \circ \eta)$.*

Proof. Let $p(\bar{s}) \in \mathcal{A}$. Since $\mathcal{A}\sigma \subseteq \mathcal{B}\eta$ there exists some $p(\mathbf{t}) \in \mathcal{B}$ where $|\bar{s}| = |\bar{t}|$ such that $p(\bar{s})\sigma = p(\bar{t})\eta$. Let $s_i \in \bar{s}$ and $t_i \in \bar{t}$. If $s_i \in \mathsf{dom}(\sigma)$ and $t_i \in \mathsf{dom}(\eta)$ then $\sigma(s_i) = \eta(t_i)$. But then $(\tau \circ \sigma)(s_i) = (\tau \circ \eta)(t_i)$. If $s_i \in \mathsf{dom}(\sigma)$ and $t_i \notin \mathsf{dom}(\eta)$ then $\sigma(s_i) = t_i$. Because t_i is a term in $\mathcal{A}$ we have that $\tau(t_i) = t_i$ and therefore $(\tau \circ \sigma)(s_i) = t_i$. Also note that $t_i \notin \mathsf{dom}(\tau \circ \eta)$ and thus $t_i(\tau \circ \eta) = t_i$. The argument works analogously in the case where $s_i \notin \mathsf{dom}(\sigma)$ and $t_i \in \mathsf{dom}(\eta)$. Lastly, if $s_i \notin \mathsf{dom}(\sigma)$ and $t_i \notin \mathsf{dom}(\eta)$ then $s_i = t_i$ as well as $s_i \notin \mathsf{dom}(\tau \circ \sigma)$ and $t_i \notin \mathsf{dom}(\tau \circ \eta)$. Hence, $p(\mathbf{s})(\tau \circ \sigma) = p(\mathbf{t})(\tau \circ \eta)$. □

The next lemma is crucial for showing that the constructed interpretations during Algorithm 3.4 and 3.6 are indeed representative of all possible pairs of interpretations.

Lemma 3.7. *Let $\mathcal{A}$ and $\mathcal{B}$ be sets of atoms, $\mathcal{I}$ an interpretation and $h\colon \mathbf{V} \to \mathbf{T}$ an homomorphism from $\mathcal{A}$ to $\mathcal{I}$. Let η be a variable substitution such that there is a total substitution τ with $h = \tau \circ \sigma$ where $\tau(t) = t$ for every $t \in$ terms$(\mathcal{A}) \cup$ terms$(\mathcal{B})$. If h cannot be existentially extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}$ then $\mathcal{A}\eta \not\models \mathcal{B}\eta_\forall$.*

Proof. Assume for a contradiction that $\mathcal{A}\eta \models \mathcal{B}\eta_\forall$. By Lemma 3.5 there is a substitution $\eta' = \eta'_\exists\eta_\forall = \eta_\forall\eta'_\exists$ which existentially extends η such that $\mathcal{B}\eta' \subseteq \mathcal{A}\eta$. Note

that $h_\forall = \tau \circ \eta_\forall$. We define $h' = (\tau \circ \eta'_\exists)h_\forall$. It is obvious that h' existentially extends h. By Lemma 3.6, $\mathcal{B}\eta' = \mathcal{B}\eta'_\exists\eta_\forall \subseteq \mathcal{A}\eta$ implies that $\mathcal{B}(\tau \circ \eta'_\exists\eta_\forall) = \mathcal{B}(\tau \circ \eta'_\exists)(\tau \circ \eta_\forall) \subseteq \mathcal{A}(\tau \circ \eta)$. Therefore, $\mathcal{B}(\tau \circ \eta'_\exists)h_\forall = \mathcal{B}h' \subseteq \mathcal{A}h \subseteq \mathcal{I}$. Hence, h' is a homomorphism from $\mathcal{B}$ to $\mathcal{I}$ which existentially extends h. □

We can now prove that Algorithm 3.3 is sound and complete.

Theorem 3.8. *Let $\rho_1 = \varphi_1 \to \psi_1$ and $\rho_2 = \varphi_2 \to \psi_2$ be rules. Then Algorithm 3.3 returns true if and only if $\rho_1 \prec^+ \rho_2$.*

Proof. $\implies$ Assume that Algorithm 3.3 returns true. This means that there is an atom mapping $m\colon \varphi_2 \to \psi_1\omega_\exists$ such that $\eta = \mathsf{unify}(m)$ and Algorithm 3.4 called on m and η returns true. We define $\varphi_{21} \coloneqq \mathsf{dom}(m)$ and $\varphi_{22} \coloneqq \varphi_2 \setminus \varphi_{22}$ like in Algorithm 3.4 and in addition set $\psi_{11} \coloneqq \mathsf{im}(m) \subseteq \psi_1\omega_\exists$. Because η is a unifier we have that $\varphi_{21}\eta = \psi_{11}\omega_\exists\eta$. Algorithm 3.4 constructs the two interpretations $\mathcal{I}_a = \varphi_1\eta \cup \varphi_{22}\eta$ and $\mathcal{I}_b = \mathcal{I}_a \cup \psi_1\omega_\exists\eta$. It follows that $\mathcal{I}_a \subseteq \mathcal{I}_b$. We now show that each condition in Definition 2.11 holds.

We start with the statement that $\mathcal{I}_b$ can be obtained by applying ρ_1 over $\mathcal{I}_a$. The unifier η is also a match for ρ_1 over $\mathcal{I}_a$ because $\varphi_1\eta \subseteq \mathcal{I}_a$ by construction. We know that the if-condition on Line 10 was not satisfied, hence $\mathcal{I}_a \not\models \psi_1\eta_\forall$. It follows from Lemma 3.5 that η cannot be existentially extended to a homomorphism from ψ_1 to $\mathcal{I}_a$ and therefore that η is an unsatisfied match. We existentially extend η to $\bar{\eta} \coloneqq \omega_\exists\eta_\forall$. Because the conditions in lines 3 and 6 were not satisfied, we know that $\eta_\forall$ does not assign any variable to an element of $\mathsf{im}(\omega_\exists)$. Hence, $\bar{\eta}$ assigns a null that was not used in $\mathcal{I}_a$ to each existential variable in ψ_1. Therefore, $\mathcal{I}_b$ is obtained by applying ρ_1 over $\mathcal{I}_a$.

Next, we show that η is an unsatisfied match for ρ_2 over $\mathcal{I}_b$. Since $\varphi_{21}\eta = \psi_{11}\omega_\exists\eta = \psi_{11}\bar{\eta} \subseteq \mathcal{I}_b$ and $\varphi_{22}\eta \subseteq \mathcal{I}_a \subseteq \mathcal{I}_b$ we have that $\varphi_2\eta \subseteq \mathcal{I}_b$, which means that η is a match for ρ_2 over $\mathcal{I}_b$. Because the if-condition on Line 17 was not satisfied and therefore $\mathcal{I}_b \not\models \psi_2\eta_\forall$ we can use Lemma 3.5 again to show that η is an unsatisfied match for ρ_2 over $\mathcal{I}_b$.

The only condition left is that η is not a match for ρ_2 over $\mathcal{I}_a$. But this directly follows from $\varphi_2\eta \not\subseteq \mathcal{I}_a$, which is the case since the check on Line 13 failed.

Therefore, all of the conditions in Definition 2.11 have been satisfied and we can conclude that $\rho_1 \prec^+ \rho_2$.

$\Longleftarrow$ To prove completeness, we assume that $\rho_1 \prec^+ \rho_2$ and show that Algorithm 3.3 returns true. We may assume w.l.o.g. that ρ_2 does not share any variables with ρ_1. In the case where $\rho_1 = \rho_2$ we proceed as if ρ_2 was a distinct rule, which is obtained from ρ_1 by uniformly renaming all of its variables. This step is justified by Lemma 3.4. Therefore there exist interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ and a variable substitution $h\colon \mathbf{V} \to \mathbf{T}$ such that

(1) h is a homomorphism from φ_1 to $\mathcal{I}_a$, that is, $\varphi_1 h \subseteq \mathcal{I}_a$,

(2) h cannot be existentially extended to a homomorphism from ψ_1 to $\mathcal{I}_a$,

(3) $\mathcal{I}_b$ is obtained by applying ρ_1 over $\mathcal{I}_a$ with match h extended to (w.l.o.g.) $h' = \omega_\exists h$, hence $\mathcal{I}_b = \mathcal{I}_a \cup \psi_1 \omega_\exists h$,

(4) h is a homomorphism from φ_2 to $\mathcal{I}_b$, that is, $\varphi_2 h \subseteq \mathcal{I}_b$,

(5) h cannot be existentially extended to a homomorphism from ψ_2 to $\mathcal{I}_b$, and

(6) h is not a homomorphism from φ_2 to $\mathcal{I}_a$ or $\varphi_2 h \not\subseteq \mathcal{I}_a$.

Since $\varphi_2 h \subseteq \mathcal{I}_b$ but $\varphi_2 h \not\subseteq \mathcal{I}_a$ by (4) and (6) respectively, there must be a partition $\psi_1 = \psi_{11} \dot\cup \psi_{12}$ and a partition $\varphi_2 = \varphi_{21} \dot\cup \varphi_{22}$ where $\varphi_{21} \neq \emptyset$ such that $\varphi_{21} h = \psi_{11} \omega_\exists h$ (†) and $\varphi_{22} h \subseteq \mathcal{I}_a$ (‡). This means that φ_{21} and $\psi_{11}\omega_\exists$ are unifiable and by Proposition 3.3 that there exists an atom mapping m with $\varphi^m_{21} \coloneqq \mathsf{dom}(m) = \varphi_{21}$ such that Algorithm 3.2 will produce a most flexible unifier η when called on m. As mentioned in Proposition 3.3 we may also assume that there exists a total substitution τ with $h = \tau \circ \eta$ such that $\tau(t) = t$ for every $t \in \mathsf{terms}(\rho_1) \cup \mathsf{terms}(\rho_2)$. We define $\varphi^m_{22} \coloneqq \varphi_2 \setminus \varphi^m_{21} = \varphi_{22}$ as well as the interpretations $\mathcal{I}^m_a = \varphi_1\eta \cup \varphi^m_{22}\eta$ and $\mathcal{I}^m_b = \mathcal{I}^m_a \cup \psi_1\omega_\exists\eta$ as in Algorithm 3.4. We will first establish that Algorithm 3.4 returns true when called on η by showing that each of the if-conditions in Algorithm 3.4 evaluate to false.

We start with the conditions on lines 3 and 6. Assume that η assigns a null to some variable $x \in \mathsf{vars}(\varphi_1 \cup \varphi^m_{22})$. Since x has to be a universal variable it must mean that $\nu(x) = \omega_\exists(v)$ for some $v \in \mathsf{vars}_\exists(\psi_1)$ where ν is the variable assignment computed during the execution of Algorithm 3.2. By Lemma 3.2 we get that $h(x) = \omega_\exists(v)$ as well. But since h is a homomorphism from φ_1 and φ_{22} to $\mathcal{I}_a$ by (1) and (‡) this would imply that $\mathcal{I}_a$ contains a null introduced by the application of ρ_1, which is a contradiction to (3).

The rest of the checks can all be handled by using Lemma 3.7. Let $\mathcal{A} \coloneqq \varphi_1 \cup \varphi_{22}$ and $\mathcal{B} \coloneqq \psi_1$. Then h is a homomorphism from $\mathcal{A}$ to $\mathcal{I}_a$ which by (2) cannot be

extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}_a$ and $\mathcal{I}_a^m = \mathcal{A}\eta$. By Lemma 3.7 we have that $\mathcal{I}_a^m \not\models \psi_1\eta_\forall$ and therefore that the check in Line 10 fails. To see that the check in Line 13 also fails first note that $\varphi_2\eta \subseteq \mathcal{I}_a$ is equivalent to $\mathcal{I}_a \models \varphi_2\eta_\forall$ since φ_2 does not contain any existential variables. Also note that if h is not a homomorphism from φ_2 to $\mathcal{I}_a$ that it also cannot be existentially extended to one. Hence, with $\mathcal{B} \coloneqq \varphi_2$ and $\mathcal{A}$ as before we can again apply Lemma 3.7. Lastly, with $\mathcal{A} \coloneqq \varphi_1 \cup \varphi_{22} \cup \psi_1\omega_\exists$ and $\mathcal{B} \coloneqq \psi_2$ we have that h is a homomorphism from $\mathcal{A}$ to $\mathcal{I}_b$ which by (5) cannot be extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}_b$. Lemma 3.7 therefore proves that the check in Line 17 fails as well.

The last step is to show that $\emptyset$ is extended to m at some point during the execution of Algorithm 3.3 or alternatively that the algorithm returns true earlier. Let $m' \neq \emptyset$ be any atom mapping which can be extended to m. It is clear that if m is unifiable then m' is also giving us the unifier $\eta^{m'} \coloneqq \mathsf{unify}(m')$. We define $\varphi_{21}^{m'} \coloneqq \mathsf{dom}(m') \subseteq \mathsf{dom}(m)$, $\varphi_{22}^{m'} \coloneqq \varphi_2 \setminus \varphi_{21}^{m'} \supseteq \varphi_{22}$ as well as $\mathcal{I}_a^{m'} \coloneqq \varphi_1\eta^{m'} \cup \varphi_{22}^{m'}\eta^{m'}$ and $\mathcal{I}_b^{m'} \coloneqq \mathcal{I}_a^{m'} \cup \psi_1\omega_\exists\eta^{m'}$. Since $\eta^{m'}$ is the most flexible unifier we also obtain a total substitution τ such that $\eta = \tau \circ \eta^{m'}$ with $\tau(t) = t$ for every $t \in \mathsf{terms}(\rho_1) \cup \mathsf{terms}(\rho_2)$ by Proposition 3.3.

We show that Algorithm 3.4 when called on m' does not reach Line 4 or 18. Assume that $\eta^{m'}(x) = \omega_\exists(v)$ for some $x \in \mathbf{V}_\forall$ and $v \in \mathsf{vars}_\exists(\psi_1)$. Then $\nu^{m'}(x) = \omega_\exists(v)$ where $\nu^{m'}$ is the variable assignment computed in Algorithm 3.2 when called on m'. But since η is also a unifer of m' we obtain by Lemma 3.2 that $\eta(x) = \omega_\exists(v)$, which is impossible because Algorithm 3.4 called on m failed the if-condition on Line 3 as established earlier. Thus, Line 4 cannot have been reached for m'. It remains to be shown that this is also the case for Line 18. Let $\mathcal{A} \coloneqq \varphi_1 \cup \varphi_{22}^{m'} \cup \psi_1\omega_\exists$ and $\mathcal{B} \coloneqq \psi_2$. We further define $\varphi_2^\Delta \coloneqq \varphi_{22}^{m'} \cap \varphi_{21}^m$. Then, $\varphi_{22}^{m'} = \varphi_{22}^m \cup \varphi_2^\Delta$. It is clear that $\mathcal{I}_b^{m'} = \mathcal{A}\eta^{m'}$. We now show that also $\mathcal{I}_b^m = \mathcal{A}\eta$. From the definition of $\mathcal{I}_b^m$ it is apparent that $\mathcal{I}_b^m \subseteq \mathcal{A}\eta$. We also know that $\varphi_1\eta \subseteq \mathcal{I}_b^m$, $\psi_1\omega_\exists\eta \subseteq \mathcal{I}_b^m$ and $\varphi_{22}^m\eta \subseteq \mathcal{I}_b^m$. From (†) it follows that $\varphi_2^\Delta\eta \subseteq \varphi_{21}^m\eta = \psi_{11}\omega_\exists\eta$ and we get $\varphi_2^\Delta\eta \subseteq \varphi_{21}^m\eta \subseteq \psi_1\omega_\exists\eta \subseteq \mathcal{I}_b^m$. Thus, $\mathcal{A}\eta \subseteq \mathcal{I}_b^m$ and therefore $\mathcal{I}_b^m = \mathcal{A}\eta$. Hence, η is a homomorphism from $\mathcal{A}$ to $\mathcal{I}_b^m$ which cannot be existentially extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}_b^m$. Lemma 3.7 now gives us that $\mathcal{I}_a^{m'} \not\models \psi_2\eta_\forall^{m'}$. In summary, calling Algorithm 3.4 on m' cannot return false.

This means that Algorithm 3.4 when called on an atom mapping that can be extended to m either returns true or calls extend and eventually reaches m. In either case, Algorithm 3.3 returns that $\rho_1 \prec^+ \rho_2$. □

We now show the analogous result for restraint reliances. Note that the provided algorithm is not complete for self-reliances as was established in Section 3.4. We therefore exclude this case in the proof.

Theorem 3.9. *Let $\rho_1 = \varphi_1 \to \psi_1$ and $\rho_2 = \varphi_2 \to \psi_2$ be rules. If Algorithm 3.5 returns true, then $\rho_1 \prec^{\square} \rho_2$. If $\rho_1 \prec^{\square} \rho_2$ and $\rho_1 \neq \rho_2$ then Algorithm 3.5 returns true.*

Proof. $\implies$ Assume that Algorithm 3.5 returns true. This means that there is an atom mapping $m\colon \psi_2 \to \psi_1\omega_\exists$ such that $\eta = \mathsf{unify}(m)$ and Algorithm 3.6 called on m and η returns true. We define $\psi_{21} \coloneqq \mathsf{dom}(m) \neq \emptyset$ and $\psi_{22} \coloneqq \psi_2 \setminus \psi_{21}$ like in Algorithm 3.6 and in addition $\psi_{11} \coloneqq \mathsf{im}(m) \subseteq \psi_1\omega_\exists$. Because η is an unifier we have that $\psi_{21}\eta = \psi_{11}\omega_\exists\eta$. Algorithm 3.6 constructs the interpretations $\tilde{\mathcal{I}}_a = \varphi_2\eta$, $\mathcal{I}_a = \tilde{\mathcal{I}}_a \cup \psi_2\omega_\exists\eta$ and $\tilde{\mathcal{I}}_b = \mathcal{I}_a \cup \varphi_1\eta \cup \psi_{22}\eta$. So $\tilde{\mathcal{I}}_a \subseteq \mathcal{I}_a \subseteq \tilde{\mathcal{I}}_b$.

We start by showing that the first two conditions of Definition 2.10 hold, namely that $\mathcal{I}_a$ is obtained by applying ρ_1 and there is a interpretation $\mathcal{I}_b$ which is obtained by applying ρ_2. Since $\varphi_1\eta \subseteq \tilde{\mathcal{I}}_b$ and $\varphi_2\eta \subseteq \tilde{\mathcal{I}}_a$ the unifier η is also a match for ρ_1 over $\tilde{\mathcal{I}}_b$ and for ρ_2 over $\tilde{\mathcal{I}}_a$. The match η is unsatisfied for both rules because $\tilde{\mathcal{I}}_b \not\models \psi_1\eta_\forall$ and $\tilde{\mathcal{I}}_a \not\models \psi_2\eta_\forall$, which follows because the checks in Line 13 and the second part in Line 18 both failed. The unifier η does not assign any variable to a null in $\omega_\exists(\mathsf{vars}_\exists(\psi_2))$ since the only nulls in m can come from $\omega_\exists(\mathsf{vars}_\exists(\psi_1))$. That the checks in Line 3 and Line 6 failed implies that there is also no null from $\omega_\exists(\mathsf{vars}_\exists(\psi_1))$ in $\tilde{\mathcal{I}}_b$. Therefore, the interpretation $\mathcal{I}_a$ was constructed from $\tilde{\mathcal{I}}_a$ in Line 16 by applying the rule ρ_2 with match η that was extended to $\bar{\eta} \coloneqq \omega_\exists\eta$. We also define $\mathcal{I}_b \coloneqq \tilde{\mathcal{I}}_b \cup \psi_1\omega_\exists\eta$, which is obtained by applying ρ_1 with match η extended to $\bar{\eta}$ from $\mathcal{I}_b$. Clearly, $\mathcal{I}_a \subseteq \mathcal{I}_b$.

To show that the third condition also holds we need to prove that η is an alternative match w.r.t. $\bar{\eta}$ and ρ_1. Since η is a unifier of ψ_{21} and $\psi_{11}\omega_\exists$ we know that $\psi_{21}\eta = \psi_{11}\omega_\exists\eta \subseteq \psi_1\bar{\eta} \subseteq \mathcal{I}_b$. By construction of $\tilde{\mathcal{I}}_b$ we also have that $\psi_{22}\eta \subseteq \mathcal{I}_b$. Therefore, η is a homomorphism from ψ_2 to $\mathcal{I}_b$. The defintion of $\bar{\eta}$ gives us that η agrees with $\bar{\eta}$ on all universal variables. And finally, η is an alternative match for ρ_1 and $\bar{\eta}$. This follows from the check in Line 9, which assures that η assigns an existential variable to either a constant or a null introduced by applying ρ_1.

Lastly, η is not an alternative match for $\bar{\eta}$ and ρ_2 over $\tilde{\mathcal{I}}_b$ because of the check in the first half of Line 18, which states that η is not a homomorphism from ψ_2 to $\tilde{\mathcal{I}}_b$. Since all of the conditions in Definition 2.10 have been satisfied we conclude that that $\rho_1 \prec^{\square} \rho_2$.

$\Longleftarrow$ To prove completeness, we assume that $\rho_1 \prec^{\square} \rho_2$ and show that Algorithm 3.5 returns true. Since we only consider the case where $\rho_1 \neq \rho_2$ we may assume that ρ_1 and ρ_2 do not share any variables. Therefore there exist interpretations $\tilde{\mathcal{I}}_a \subseteq \mathcal{I}_a \subseteq \tilde{\mathcal{I}}_b \subseteq \mathcal{I}_b$ and a variable substitution $h\colon \mathbf{V} \to \mathbf{T}$ such that

(1) h is a homomorphism from φ_2 to $\tilde{\mathcal{I}}_a$, that is, $\varphi_2 h \subseteq \tilde{\mathcal{I}}_a$,

(2) h cannot be extended to a homomorphism from ψ_2 to $\tilde{\mathcal{I}}_a$

(3) $\mathcal{I}_a$ was obtained by applying ρ_2 with match h extended to (w.l.o.g.) $h' = \omega_{\exists} h$, hence $\mathcal{I}_a = \tilde{\mathcal{I}}_a \cup \psi_2 \omega_{\exists} h$,

(4) h is a homomorphism from φ_1 to $\tilde{\mathcal{I}}_b$, that is, $\varphi_1 h \subseteq \tilde{\mathcal{I}}_b$,

(5) h cannot be extended to a homomorphism from ψ_1 to $\tilde{\mathcal{I}}_b$

(6) $\mathcal{I}_b$ was obtained by applying ρ_1 with match h extended to (w.l.o.g.) h', hence $\mathcal{I}_b = \tilde{\mathcal{I}}_b \cup \psi_1 \omega_{\exists} h$,

(7) there is a homomorphism h^A which extends h such that $\psi_2 h^A \subseteq \mathcal{I}_b$ and $h^A(\mathsf{vars}_{\exists}(\psi_2)) \neq h(\mathsf{vars}_{\exists}(\psi_2))$, and

(8) h^A is not a homomorphism from ψ_2 to $\tilde{\mathcal{I}}_b$.

Since $\psi_2 h^A \subseteq \mathcal{I}_b$ but $\psi_2 h^A \not\subseteq \tilde{\mathcal{I}}_b$ by (7) and (8) respectively, there must be a partition $\psi_1 = \psi_{11} \dot{\cup} \psi_{12}$ and a partition $\psi_2 = \psi_{21} \dot{\cup} \psi_{22}$ where $\psi_{21} \neq \emptyset$ such that $\psi_{21} h^A = \psi_{11} \omega_{\exists} h^A$ (†) and $\psi_{22} h^A \subseteq \tilde{\mathcal{I}}_b$ (‡). This means that ψ_{21} and $\psi_{11}\omega_{\exists}$ are unifiable and by Proposition 3.3 that there exists an atom mapping m with $\psi_{21}^m \coloneqq \mathsf{dom}(m) = \psi_{21}$ such that Algorithm 3.2 when called on m will produce a most flexible unifier η. We may also assume that there exists a total substitution τ with $h^A = \tau \circ \eta$ such that $\tau(t) = t$ for every $t \in \mathsf{terms}(\rho_1) \cup \mathsf{terms}(\rho_2)$. We define $\psi_{22}^m \coloneqq \psi_2 \setminus \psi_{21}^m = \psi_{22}$ as well as $\tilde{\mathcal{I}}_a^m \coloneqq \varphi_2\eta$, $\mathcal{I}_a^m \coloneqq \tilde{\mathcal{I}}_a^m \cup \psi_2\omega_{\exists}\eta$ and $\tilde{\mathcal{I}}_b^m \coloneqq \mathcal{I}_a^m \cup \varphi_1\eta \cup \psi_{22}^m\eta$. First, we establish that Algorithm 3.4 returns true when called on η by showing that each of the if-conditions in Algorithm 3.6 evaluate to false.

We start with the condition on Line 3. Assume that η assigns a null to some variable $x \in \mathbf{V}$. Hence, $\nu(x) = \omega_{\exists}(v)$ for some $v \in \mathsf{vars}_{\exists}(\psi_1)$ where ν is the variable assignment computed during the execution of Algorithm 3.2. By Lemma 3.2 we get that $h^A(x) = h(x) = \omega_{\exists}(v)$ as well. But since h is a homomorphism from φ_1 to $\tilde{\mathcal{I}}_b$ and φ_2 to $\tilde{\mathcal{I}}_a$ by (4) and (1) respectively, this would imply that $\tilde{\mathcal{I}}_a$ or $\tilde{\mathcal{I}}_b$ contain nulls introduced by the application ρ_1 over $\tilde{\mathcal{I}}_b$, which is a contradiction to (6).

We continue with the condition on Line 6. Let us assume that η assigns a null to some existential variable x in ψ_{22}. Again, we have that $\nu(x) = \omega_\exists(v)$ for some $v \in \mathsf{vars}_\exists(\psi_1)$. Lemma 3.2 gives us $h^A(x) = \omega_\exists(v)$ as well. But by $(\ddagger)$ we have $\psi_{22}h^A \subseteq \tilde{\mathcal{I}}_b$ which means that $\tilde{\mathcal{I}}_b$ contain nulls introduced by the application ρ_1 over $\tilde{\mathcal{I}}_b$, which once again is a contradiction to (6).

To show that the if-condition on Line 9 fails we assume for a contradiction that ψ_{21} does not contain any existential variables. Together with (3), this implies that $\psi_{21}h^A = \psi_{21}\omega_\exists h \subseteq \mathcal{I}_a \subseteq \tilde{\mathcal{I}}_b$. From $(\ddagger)$ we further obtain that $\psi_{22}h^A \subseteq \tilde{\mathcal{I}}_b$. Overall this means that $\psi_{21}h^A \subseteq \tilde{\mathcal{I}}_b$, which contradicts (8).

We continue with the if-statements in Line 13 and the second part of the if-statement in Line 18. Both can be handled with Lemma 3.7. First, we set $\mathcal{A}_1 \coloneqq \varphi_2$, $\mathcal{B}_1 \coloneqq \psi_2$ and $\mathcal{I}_1 \coloneqq \tilde{\mathcal{I}}_a$. Since h^A and h agree on all universal variables and (1) we have that h^A is a homomorphism from $\mathcal{A}_1$ to $\mathcal{I}_1$. It cannot be extended to a homomorphism from $\mathcal{B}_1$ to $\mathcal{I}_1$ by (2). Lemma 3.7 then implies that $\tilde{\mathcal{I}}_a^m = \mathcal{A}_1\eta \not\models \mathcal{B}_1\eta_\forall = \psi_2\eta_\forall$. We now set $\mathcal{A}_2 \coloneqq \varphi_2 \cup \psi_2\omega_\exists \cup \varphi_1 \cup \psi_{22}$, $\mathcal{B}_2 \coloneqq \psi_1$ and $\mathcal{I}_2 \coloneqq \tilde{\mathcal{I}}_b$. We already know that h^A is a homomorphism from φ_1 to $\tilde{\mathcal{I}}_a \subseteq \tilde{\mathcal{I}}_b$. It is also a homomorphism from φ_2 to $\tilde{\mathcal{I}}_b$ by the same reasoning and using the condition (1). In addition, we have that $\psi_2\omega_\exists h^A = \psi_2\omega_\exists h \subseteq \mathcal{I}_a \subseteq \tilde{\mathcal{I}}_b$ using (3) and $\psi_{22}h^A \subseteq \tilde{\mathcal{I}}_b$ by $(\ddagger)$. Hence, h^A is a homomorphism from $\mathcal{A}_2$ to $\mathcal{I}_2$ which cannot be extended to a homomorphism from $\mathcal{B}_2$ to $\mathcal{I}$ by (5) and $\tilde{\mathcal{I}}_b^m = \mathcal{A}_2\eta$. By Lemma 3.7 we obtain that $\tilde{\mathcal{I}}_b^m \not\models \psi_1\eta_\forall$. Therefore, both conditions evaluate to false.

For the first part of the if-condition in Line 18 assume that $\psi_2\eta \subseteq \tilde{\mathcal{I}}_b^m = \mathcal{A}_2\eta$. Lemma 3.6 implies $\psi_2 h^A = \psi_2(\tau \circ \eta) \subseteq \mathcal{A}_2(\tau \circ \eta) = \mathcal{A}_2 h^A \subseteq \tilde{\mathcal{I}}_b$, which would contradict (8). The whole if-condition on Line 18 therefore fails.

We have now proven that Algorithm 3.6 returns true when called on η and m. The only part left is to show that m is reached by Algorithm 3.5 or alternatively that it returns true at an earlier point. Let $m' \neq \emptyset$ be any atom mapping which can be extended to m. It is clear that if m is unifiable so is m', giving us the unifier $\eta^{m'} \coloneqq \mathsf{unify}(m')$. We define $\psi_{21}^{m'} \coloneqq \mathsf{dom}(m') \subseteq \mathsf{dom}(m)$, $\psi_{22}^{m'} \coloneqq \psi_2 \setminus \psi_{21}^{m'} \supseteq \psi_{22}$ as well as $\tilde{\mathcal{I}}_a^{m'} \coloneqq \varphi_1\eta^{m'} \cup \varphi_{22}^{m'}\eta^{m'}$. Since $\eta^{m'}$ is the most flexible unifier we also obtain a total substitution τ such that $\eta = \tau \circ \eta^{m'}$ with $\tau(t) = t$ for every $t \in \mathsf{terms}(\rho_1) \cup \mathsf{terms}(\rho_2)$ by Proposition 3.3.

We now show that Algorithm 3.6 called on m' will not reach Line 4 or 14. Assume that $\eta^{m'}(x) = \omega_\exists(v)$ for some $x \in \mathbf{V}_\forall$ and $v \in \mathsf{vars}_\exists(\psi_1)$. Then $\nu^{m'}(x) = \omega_\exists(v)$ where $\nu^{m'}$ is the variable assignment computed in Algorithm 3.2 when called on

m'. Since η is also a unifier of m' we obtain by Lemma 3.2 that $\eta(x) = \omega_{\exists}(v)$ which is impossible because Algorithm 3.6 when called on m did not pass the if-condition on Line 3 as shown earlier. Thus, Line 4 cannot have been reached for m'. The only thing left is to show that this is also the case for Line 14. Let $\mathcal{A} \coloneqq \varphi_2$, $\mathcal{B} \coloneqq \psi_2$ and $\mathcal{I} \coloneqq \tilde{\mathcal{I}}_a^m$. Then η is a homomorphism from $\mathcal{A}$ to $\mathcal{I}$ which cannot be extended to a homomorphism from $\mathcal{B}$ to $\mathcal{I}$. Using Lemma 3.7 we obtain that $\tilde{\mathcal{I}}_a^{m'} = \mathcal{A}\eta^{m'} \not\models \mathcal{B}\eta^{m'} = \psi_2\eta^{m'}$ and hence that the check in Line 13 fails when Algorithm 3.6 is called on m'.

Thus, Algorithm 3.5 returns that $\rho_1 \prec^{\square} \rho_2$ which concludes the proof.

□

Chapter 4

Optimizing the Chase

The algorithms presented in the previous chapter allow us to compute the existence of positive and restraint reliances between every possible pair of rules from a given rule set. Based on this information, we devise a strategy for applying rules that optimizes the chase procedure. Our main objectives are:

1. Minimizing the number of redundant derivations, producing core-models if possible.
2. Minimizing the number of produced predicate-blocks by applying rules as rarely as possible.

Krötzsch already gave a brief description of a suitable strategy that guarantees to produce core models [18]. However, it is limited to core-stratified rule sets. Gerlach relaxed this requirement by introducing the hybrid chase [13]. This variant switches from the restricted to the core or the eam chase in the last layer of a relaxed restrained partitioning. The strategy presented here uses the cheaper restricted chase for its entire run, but unlike the approach outlined by Krötzsch is also defined for rule sets that are not core-stratified. Although the procedure shown here does not ensure to generate a core model for every input, we hope to minimize the amount of redundant facts by avoiding alternative matches as best as possible. Unlike the strategies mentioned thus far, our approach also takes practical concerns in regard to the semi-naive evaluation performed by VLog into account.

This chapter is divided into two parts. In Section 4.1, we concentrate on core-stratified rule sets. Section 4.2 then generalizes the strategy derived previously to arbitrary knowledge bases.

4.1 Core-Stratified Rule Sets

We begin by restricting our focus to knowledge bases where the rule set is core-stratified. In such cases it is always possible to avoid violating a restraint reliance, i.e. applying a restrained rule before the one restraining it. This results in a chase sequence that does not contain any alternative matches thereby producing a core model. We first give an overview over existing approaches. Since our end goal is to provide an effective strategy even for rule sets that are not core-stratified, we assess how well the existing strategies can be generalized. We then discuss modifications to suitable existing strategies that avoid unnecessary rule applications.

4.1.1 Existing Approaches for Computing Cores

As mentioned, Krötzsch provided a suitable strategy for applying rules that guarantees to produce core models for core-stratified rule sets [18]. The basic idea is to exhaustively apply rules in the downward closure $\rho{\downarrow^{\square}}$ of a rule ρ before applying the rule itself. We obtain a possible implementation of such a strategy by replacing the main loop in Algorithm 2.1 as follows.

```
1 while R contains rule which is applicable over I do
2 |   ρ := select(R);
3 |   if ρ↓□ contains a rule which is applicable over I then Skip ;
4 |   I := applyRule(ρ, I);
5 end
```

In every step, a rule ρ is fairly selected from the rule set R. But it is only applied if no rule from $\rho{\downarrow^{\square}}$ is applicable over the current interpretation $\mathcal{I}$. Note that this is equivalent to stating that every rule in $\rho{\downarrow^{\square}}$ is inactive, because every rule which could possibly trigger a rule in $\rho{\downarrow^{\square}}$ is itself contained in the downward closure of ρ. Ordering the application of rules in such a way ensures that no restrained rule is ever applied before the rules restraining it, which leads to a core model. But this approach cannot be generalized to arbitrary knowledge bases. To see why consider the following example.

Example 4.1.

$$a(x, y) \to \exists v, w.\, h(x, v, w) \wedge b(y, v, w) \qquad (\rho_1)$$
$$b(x, y, z) \to h(x, z, y) \wedge b(x, z, y) \qquad (\rho_2)$$

In the above rule set ρ_2 positively relies on ρ_1 but also restrains it. The downward closure of both rules is therefore $\rho_1{\downarrow^{\square}} = \rho_2{\downarrow^{\square}} = \{\rho_1, \rho_2\}$. The algorithm described above would therefore be incapable of applying either rule, leading to an infinite loop where no rule is applied. △

Hence, no algorithm of this sort is suitable as a general method for every rule set. With the transfinite chase, Gerlach offers a more flexible template for ordering rules during a restricted chase run [13]. In the core-stratified case, rules are divided into a restrained partitioning, which defines a transfinite chase sequence without alternative matches. From this, it is possible to construct an equivalent restricted chase sequence that does not contain any alternative matches as well. However, it is also possible to define a chase algorithm based on the transfinite chase itself by chaining together multiple runs of the restricted chase. The ith run starts on the (possibly infinite) result of the previous one and continuously applies rules from the rule set $R^{\leq i}$. Note that we require the union of all partitions up to R_i, because newly introduced rules from R_i might trigger rules from previous partitions. Algorithm 4.1 shows the whole procedure.

Algorithm 4.1: transfinite chase

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$, RestrainedPartitioning $R_1, \ldots, R_n$
Output: Interpretation $\mathcal{I}$

```
1  I := D;
2  i := 1;
3  Q := ∅;
4  while i ≤ n do
5  |  Q := Q ∪ R_i;
6  |  while Q contains a rule which is applicable over I do
7  |  |  ρ := select(Q);
8  |  |  I := applyRule(ρ);
9  |  end
10 |  i := i + 1;
11 end
12 return I;
```

It is worth mentioning that even though Algorithm 4.1 consists of chaining together multiple restricted chase runs, the resulting series of chase steps may not be fair. This situation occurs whenever a rule from some partition R_j is applicable, but never actually satisfied because an earlier chase run is non-terminating. However, Algorithm 4.1 is still semi-fair, which means that at least every terminating run results in a restricted chase sequence. This follows from the fact that during the run of the last chase rules are selected from the whole rule set $R^{\leq n} = R$. Algorithm 4.1 therefore only terminates once every rule from the given rule set is no longer applicable, which implies that the resulting finite sequence is fair. Gerlach also obtained an interesting result regarding the termination of the transfinite chase. Given a knowledge base $\mathcal{K}$ containing a core-stratified rule set, Algorithm 4.1 terminates if any finite restricted chase sequence for $\mathcal{K}$ exists [13]. We give a brief argument for why this is the case in the next proposition.

Proposition 4.1. *Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base with a core-stratified rule set R and $\bar{R} = R_1, \ldots, R_n$ restrained partition of R. If there exists a finite chase sequence over $\mathcal{K}$ then Algorithm 4.1 will also terminate.*

Proof. Assume that Algorithm 4.1 does not terminate and let $(\mathcal{I}_k^1), \ldots, (\mathcal{I}_k^j)$ be the chase sequences produced during each chase run. Then, $(\mathcal{I}_k^j)$ is an infinite sequence. We can extend this series of chase sequences to obtain an infinite transfinite chase sequence $(\mathcal{I}_k^1), \ldots, (\mathcal{I}_k^n)$ for $\mathcal{K}$ and $\bar{R}$. By Lemma 2.4 it does not contain any alternative matches. Lemma 2.5 then implies the existence of an equivalent infinite restricted chase sequence without alternative matches as well. But then there would exist an infinite core model in addition to a finite universal model for $\mathcal{K}$, which contradicts Lemma 2.2. □

In order to guarantee that the resulting interpretation is a core, the transfinite chase requires the provided list of rule sets to be a restrained partitioning, which presumes a core-stratified rule set. However, the transfinite chase is a well-defined algorithm for any arbitrary covering of the rule set. By restricting or relaxing the conditions imposed on the list of rule sets we can enforce additional properties or allow for arbitrary knowledge bases respectively.

4.1.2 Minimizing the Number of Parallel Chase Steps

As outlined in Section 2.4, the performance of the semi-naive evaluation algorithm may to a significant amount depend on the number of times a rule is applied during a chase run. We therefore try to delay the application of a rule as long as possible, hoping to increase the amount of matches that are satisfied at once. Unsatisfied matches for a rule ρ can potentially be produced by applying any rule that ρ positively relies on. Ideally, we should therefore wait with satisfying ρ until we are sure that every predecessor of ρ in $G^+ = G(R, \prec^+)$ is inactive, because after this point ρ can only be applied at most once. However, such a strategy is not possible if the given rule set R contains cyclic dependencies w.r.t. positive reliances, i.e. if G^+ is not acyclic. We can work around that problem by switching our focus from individual rules to strongly connected components of rules in G^+. This allows us to formulate a strategy whereby the application of rule ρ is only permitted once every rule from the predecessors of $[\rho]$ in $\hat{G}^+$ are inactive. Although determining whether or not a rule is inactive at any given moment is undecidable, we can use the strongly connected components to provide a sufficient condition. Once every rule from a given component $[\rho]$ is not applicable over the current interpretation, its rules can only be triggered by predecessors of $[\rho]$. If we determined each of them to be inactive as well then we can safely assume that no rule from $[\rho]$ will ever become applicable after this point. We can implicitly enforce this condition by using any topological sorting of $\hat{G}^+$ to provide an ordered partitioning. We then exhaustively apply all rules from each partition in the given order. This leads to an algorithm quite similar to the transfinite chase, which chains together multiple restricted chase runs that are executed on the strongly connected components of G^+.

Algorithm 4.2: Partitioned Chase

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$, topolSort $R_1, \ldots, R_n$ of $\hat{G}(R, \prec^+)$
Output: Interpretation $\mathcal{I}$

```
1 I := D;
2 i := 1;
3 while i ≤ n do
4 |   ρ := select(R_i);
5 |   applyRule(ρ);
6 |   i := i + 1;
7 end
8 return I;
```

The above algorithm differs from the transfinite chase by using the partition R_i alone as input for the ith chase run instead of the union $R^{\leq i}$ of all previous partitions. It still ensures fairness for finite runs because rules from R_i cannot cause unsatisfied matches for rules in previous partitions. In fact, this reasoning extends to any arbitrary ordered partitioning which is compatible with $\prec^+$ is the following sense.

Definition 4.1. Let R be a set of rules, $\prec \subseteq R \times R$ a binary relation on R and $R_1, \ldots, R_n$ an ordered partitioning of R. The partitioning is *compatible* with $\prec$ if $\rho_i \prec \rho_j$ for $\rho_i \in R_i$ and $\rho_j \in R_j$ implies $i \leq j$ for every $i, j \in [n]$.

Proposition 4.2. *Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base and $R_1, \ldots, R_n$ be an ordered partitioning of R. If $R_1, \ldots, R_n$ is compatible with $\prec^+$ over R then Algorithm 4.2 is a semi-fair chase algorithm.*

Proof. Let $\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n$ be a finite series of chase steps produced by Algorithm 4.2 when called on $\mathcal{K}$. Since Algorithm 4.2 uses the restricted chase itself in each partition, the produced sequence is valid. Proving fairness is equivalent to showing that no rule is applicable over $\mathcal{I}_n$. Assume for a contradiction that h is an unsatisfied match for some rule $\rho_\ell \in R_\ell$ over $\mathcal{I}_n$. Let $i_\ell \in [n]$ be such that $\mathcal{I}_{i_\ell}$ is the last step where any rule from R_ℓ is applied during the run. Hence, h is not an unsatisfied match for ρ_ℓ over $\mathcal{I}_{i_\ell}$. It cannot be a satisfied match either because in this case it would also be satisfied over $\mathcal{I}_n$. Therefore, h is not a match for ρ_ℓ over $\mathcal{I}_{i_\ell}$. But then there must exist some $i_r > i_\ell$ such that h is not a match over $\mathcal{I}_{i_r - 1}$ but is an unsatisfied match for ρ over $\mathcal{I}_{i_r}$. Let $\rho_r \in R_r$ be the rule applied to obtain $\mathcal{I}_{i_r}$. It follows that $\rho_r \prec^+ \rho_\ell$ and $r > \ell$. The provided partitioning would therefore not be compatible with $\prec^+$, which is a contradiction. □

For core-stratified rule sets, the partitioned chase as presented above is also compatible with avoiding alternative matches.

Proposition 4.3. *Let R be a core-stratified set of rules and $G^+ = G(R, \prec^+)$. Then there exists an ordered partitioning $R_1, \ldots, R_n$ which is a topological sorting of $\hat{G}^+$ that is also compatible with $\prec^\square$.*

Proof. Let $\hat{G}^+_\square = \hat{G}(R/_{\prec^+}, \hat{\prec^+} \cup \hat{\prec^\square})$ be a graph extending $\hat{G}^+$ with edges representing the $\prec^\square$-relationship. We show that $\hat{G}^+_\square$ does not contain any proper cycle and is therefore topologically sortable. Any such sorting is then a $\prec^\square$-compatible topological sorting of $\hat{G}^+$. Assume that there is a proper cycle in $\hat{G}^+_\square$ and hence

a path $[\rho_1], \ldots, [\rho_n]$ such that $[\rho_1] = [\rho_n]$. Since every rule in $[\rho]$ is reachable through $\prec^+$-edges from any other rule in $[\rho]$, there must exist a sequence of rules $\rho'_1, \ldots, \rho'_m$ with $\rho'_1 = \rho'_m$ where $\rho'_i \ (\prec^+ \cup \prec^\square) \ \rho'_{i+1}$ for every $i \in [m-1]$. This sequence must contain at least one rule ρ'_k such that $\rho'_{k-1} \prec^\square \rho'_k$. If this were not the case then $\rho'_i \prec^+ \rho'_{i+1}$ for every $i \in [m-1]$ which would imply that $[\rho_1] = \cdots = [\rho_n]$. But then $[\rho_1], \ldots, [\rho_n]$ would not be a proper cycle. Hence, we obtain that $\rho'_k \ ((\prec^+)^* \circ \prec^\square)^+ \ \rho'_k$, which means that R is not core-stratified. □

Thus, we can use any topological sorting of $\hat{G}^+_\square$ as input for the partitioned chase. In particular, such partitionings are also restrained partitionings.

Proposition 4.4. *Let R be a core-stratified rule set and $G^+ = G(R, \prec^+)$. Furthermore, let $R_1, \ldots, R_n$ be any ordered partitioning which results from a topological sort of $\hat{G}^+_\square = G(R/_{\prec^+}, \hat{\prec}^+ \cup \hat{\prec}^\square)$. Then $R_1, \ldots, R_n$ is a restrained partitioning.*

Proof. For a contradiction assume that $R_1, \ldots, R_n$ is not a restrained partitioning. Then there exist rules $\rho_\ell \in R_\ell$ and $\rho_r \in R_r$ with $\ell \leq r$ such that $\rho_r \in \rho_\ell{\downarrow^\square}$. Therefore, there is a sequence of rules $\rho'_1, \ldots, \rho'_m$ with $\rho'_1 = \rho_r$ and $\rho'_m = \rho_\ell$ where $\rho'_i \ (\prec^+ \cup \prec^\square) \ \rho'_{i+1}$ for every $i \in [m-1]$. If $R_\ell = R_r$ then there exists a path from ρ_ℓ to ρ_r in G^+. But then $\rho_\ell \in \rho_\ell{\downarrow^\square}$. If, on the other hand, $\ell < r$ there has to be a $k \in [m-1]$ such that $\rho'_k \in R_{i_k}$ and $\rho'_{k+1} \in R_{i_{k+1}}$ with $i_{k+1} < i_k$. Therefore $R_{i_k} \ (\hat{\prec}^+ \cup \hat{\prec}^\square) \ R_{i_{k+1}}$. But this violates the assumption that $R_1, \ldots, R_n$ results from the topological sort of $\hat{G}^+_\square$. □

The above statement together with the argument from Proposition 4.2 imply that the transfinite and the partitioned chase behave the same when called on topological sorts of $\hat{G}^+_\square$. From this it immediately follows that the chase sequences produced by Algorithm 4.2 are free of alternative matches. We may also apply the result from Proposition 4.1 to the partitioned chase as well. In summary, we obtain an algorithm that

- produces a core-model,
- minimizes the number of parallel chase steps and
- terminates if any terminating sequence of the restricted chase exists

when called on knowledge bases that contain a core-stratified rule set.

Example 4.2. Recall the rule set R from the introduction

$$\mathsf{leadingRole}(a, r, m) \to \mathsf{stars}(a, m) \qquad (\rho_1)$$
$$\mathsf{stars}(a, m) \land \mathsf{stars}(b, m) \to \mathsf{costar}(a, b, m) \qquad (\rho_2)$$
$$\mathsf{bigBudget}(m) \to \exists a.\, \mathsf{stars}(a, m) \land \mathsf{famous}(a) \qquad (\rho_3)$$

Here we have that ρ_2 positively relies on ρ_1 and ρ_3 while ρ_1 restrains ρ_3. Setting $R_1 \coloneqq \{\rho_1\}$, $R_2 \coloneqq \{\rho_3\}$ and $R_3 \coloneqq \{\rho_2\}$, we obtain a topological sorting of the strongly connected components in $G^+ = G(R, \prec^+)$ that is also compatible with $\prec^\square$. The 1-parallel version of the partitioned chase when called on this partitioning will therefore produce a core model and, since G^+ is acyclic, do so while only applying each rule at most once. △

4.2 Dealing with Non-Stratified Rule Sets

Thus far, the provided algorithms only cover the situation where the given set of rules is core-stratified. Although it may be impossible to produce cores if this is not the case, we might still be able to avoid certain alternative matches by recognizing restraint reliances between rules and ordering their application accordingly. On a similar note, it could at the same time be possible to accumulate matches, satisfying them with only a few parallel applications of a rule using positive reliances. The following example illustrates this point.

Example 4.3. Consider the following set of rules, which adds ρ_3 and ρ_4 to the rule set from Example 4.1.

$$a(x, y) \to \exists v, w.\, h(x, v, w) \land b(y, v, w) \qquad (\rho_1)$$
$$b(x, y, z) \to h(x, z, y) \land b(x, z, y) \qquad (\rho_2)$$
$$c(x) \to a(x, x) \qquad (\rho_3)$$
$$d(x, y, z) \to h(x, y, z) \qquad (\rho_4)$$

As before, we have $\rho_1 \prec^+ \rho_2$ and $\rho_2 \prec^\square \rho_1$. In addition, ρ_1 now positively relies on ρ_3 while ρ_4 restrains ρ_1. Even though the above rule set is not core-stratified we notice that is always possible to exhaustively apply ρ_3 and ρ_4 before ever satisfying ρ_1 or ρ_2. Exhaustively applying the former prevents splitting facts resulting from ρ_1 into multiple blocks while exhaustively applying the latter prevents any alternative match caused by applying ρ_4. △

The above example motivates the need for a procedure that is able to temporarily exclude certain rules from consideration based on their reliance relation to other rules. The partitioned chase introduced in the previous section is a chase algorithm which accomplishes this task for core-stratified rule sets. In the next section we show that by relaxing the conditions of the required ordered partitioning we can generalize the procedure to every possible rule set.

4.2.1 The Generalized Partitioned Chase

Extending Algorithm 4.2 to arbitrary rule sets requires us to lift the condition that the provided partitioning has to be a restrained partitioning. In order to achieve an improvement compared to just running the restricted chase itself, a meaningful restriction on the acceptable partitionings has to be made. We first observe, that any suitable ordered partitioning $R_1, \ldots, R_n$ of a rule set R must have the following properties:

1. The ordered partitioning must be compatible with $\prec^+$
2. The ordered partitioning must be compatible with $\prec^\square$
3. If R is core-stratified, then $R_1, \ldots, R_n$ is a restrained partitioning.

Violating the first property makes it possible for a rule to trigger another one from a previous partition. But since no partition is ever revisited during the run of the algorithm this would lead to an unfair chase sequence even if the algorithm terminates. Not satisfying the second property may force the chase run to produce alternative matches. The last property simply ensures that the algorithm produces a core in the well-behaved case of core-stratified rule sets. It also allows us to take advantage of the termination result in Proposition 4.1.

Besides these hard constraints on on the allowed partitionings, we also prefer each of the partitions to be as small as possible. We expect that this leads to chase runs that exclude more alternative matches and reduce the amount of parallel chase steps. Based on these restrictions, we obtain a suitable ordered partitioning by dividing the rule set R into strongly connected components in the rule graph containing positive and restraint reliances. Formally, this corresponds to any topological sorting of the graph $\hat{G}^\star = \hat{G}(R, \prec^+ \cup \prec^\square)$. By definition, any such ordering is compatible with $\prec^+$ and $\prec^\square$. In fact, since strongly connected components cannot be split into multiple partitions without violating the first or the second condition they are the smallest possible partitions as well.

Example 4.4. In the rule set from Example 4.3, the following ordered partitioning results from a topological sort of $\hat{G}^*$: $R_1 = \{\rho_3\}$, $R_2 = \{\rho_4\}$ and $R_3 = \{\rho_1, \rho_2\}$. Any split of R_3 results in an ordered partition that is either incompatible with $\prec^+$ or with $\prec^\square$. △

As the following proposition shows, topological sortings of the graph $\hat{G}^\star$ correspond to $\prec^\square$-compatible topological sortings of $\hat{G}^+$ if the provided rule set is core-stratified. It follows from Proposition 4.4 that these are restrained partitionings as well.

Proposition 4.5. *Let R be a core-stratified rule set and $R_1, \ldots, R_n$ a topological sorting of $\hat{G}^\star = \hat{G}(R, \prec^+ \cup \prec^\square)$. Then $R_i \in R/_{\prec^+}$ for all $i \in [n]$.*

Proof. Assume that there exists a partition R_i and rules $\rho_1, \rho_2 \in R_i$ such that $\rho_1 \prec^\square \rho_2$. The rule ρ_1 must be reachable from ρ_2 in $G(R, \prec^+ \cup \prec^\square)$, since ρ_1 and ρ_2 are contained within the same partition. But then $\rho_2 \in \rho_2{\downarrow^\square}$, which would imply that R is not core-stratified. □

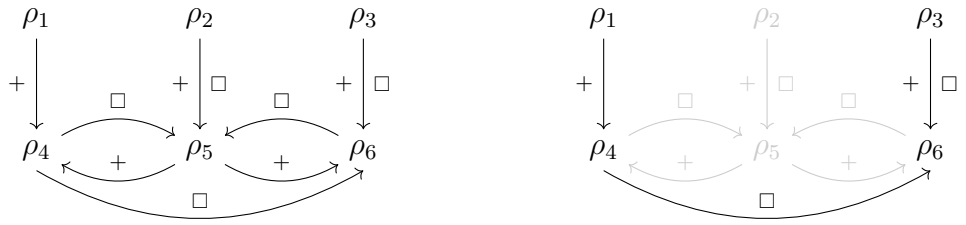
Based on these considerations we give a generalized version of the partitioned chase, which is shown in Algorithm 4.3, that accepts any knowledge base as input. Note that this is a semi-fair chase algorithm by Proposition 4.2.

Algorithm 4.3: Generalized Partitioned Chase

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$, topolSort $R_1, \ldots, R_n$ of $\hat{G}(R, \prec^+ \cup \prec^\square)$
Output: Interpretation $\mathcal{I}$

```
1 I := D;
2 i := 1;
3 while i ≤ n do
4 |   ρ := select(R_i);
5 |   applyRule(ρ);
6 |   i := i + 1;
7 end
8 return I;
```

However, there are still ways in which we can improve our current approach. One limitation of the above algorithm is that its partitionings are computed from the whole rule set at the start of its execution. But this ignores the fact that rules may become inactive in the middle of a chase run. Each dependency based on a reliance originating from such an inactive rule can safely be disregarded after this point. In certain cases this may lead to a situation where strongly connected



(a) Graph including the whole rule set **(b)** Graph excluding inactive rules

Figure 4.1: Visualization of the rule set from Example 4.5.

components split into smaller groups. In the best case scenario, the set of the remaining potentially active rules may become core-stratified even if the original rule set was not. We illustrate this point in the following example.

Example 4.5. Consider a knowledge base $\mathcal{K} = \langle R, \mathcal{D} \rangle$ consisting of the database $\mathcal{D} = \{e_1(1,2,3), e_3(1,2,2)\}$ and the rule set R

$$e_i(x,y,z) \to a_i(x,y,z),\ i \in \{1,2,3\} \tag{ρ_i}$$
$$a_1(x,y,z) \to a_1(x,y,y) \tag{ρ_4}$$
$$a_2(x,y,z) \to \exists v, w.\, a_1(x,v,w) \wedge a_3(x,v,w) \tag{ρ_5}$$
$$a_3(x,y,z) \to \exists v.\, a_1(x,v,v) \wedge a_3(x,v,v) \tag{ρ_6}$$

The reliance relationships of R are visualized in Figure 4.1. After applying ρ_1 and ρ_3 we obtain $\mathcal{I} = \mathcal{D} \cup \{a_1(1,2,3), a_3(1,2,2)\}$. This makes ρ_4 and ρ_6 applicable over $\mathcal{I}$. But since ρ_4, ρ_5 and ρ_6 form a strongly connected component, Algorithm 4.3 may select ρ_6 before ρ_4, which leads to an alternative match after satisfying ρ_4. Because ρ_2 is not applicable over $\mathcal{D}$ but the only rule which can trigger ρ_5, we know that the latter is inactive. Removing ρ_5 from consideration splits the component $\{\rho_4, \rho_5, \rho_6\}$ into the "potentially active" groups $\{\rho_4\}$ and $\{\rho_6\}$. This leaves us with only one remaining restraint relation that has to be taken into account, namely $\rho_4 \prec^{\square} \rho_6$. △

An algorithm that ignores inactive rules during its run can therefore reliably prevent alternative matches in cases like in the above example. The next section provides such a procedure.

4.2.2 The Dynamic Partitioned Chase

The goal of this section is to derive an algorithm that reacts appropriately to situations where certain rules become inactive during the chase run. For this purpose, we use the partitioned chase as a starting point but now recalculate the ordered partitioning every time we assess a rule to be inactive. Since every partition might contain multiple $\prec^+$-groups, we can no longer use the applicability of $[\rho]$ over the current interpretation alone to determine whether said group is inactive as we did in the core-stratified case. Instead, we now need to explicitly keep track of what $\prec^+$-groups still contain potentially active rules. Only after every predecessor of a group in $\hat{G}(R, \prec^+)$ has been determined to be inactive is it safe to remove it from consideration, provided the group does not contain any unsatisfied rules. The following algorithm implements this approach.

Algorithm 4.4: dynamic partitioned chase

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$
Output: Interpretation $\mathcal{I}_\infty$

Function `updateActive(`*Group* $[\rho]$`)`**:**
  if *pred*$_{\hat{G}^+}([\rho]) \cap A = \emptyset$ *and no rule in* $[\rho]$ *is applicable* **then**
    $A \coloneqq A \setminus \{[\rho]\}$;
    change $\coloneqq$ true;
    foreach $[\rho'] \in$ *succ*$_{\hat{G}^+}([\rho])$ **do**
      `updateActive(`$[\rho']$`)`;
    end
  end
$G^+ \coloneqq G(R, \prec^+)$;
$\mathcal{I} \coloneqq \mathcal{D}$;
$A \coloneqq R/_{\prec^+}$;
change $\coloneqq$ true;
foreach $[\rho] \in R/_{\prec^+}$ *with pred*$_{\hat{G}^+}([\rho]) = \emptyset$ **do** `updateActive(`$[\rho]$`)` ;
while $A \neq \emptyset$ **do**
  if *change* **then**
    $R_A \coloneqq \bigcup A$;
    $Q_1, \ldots, Q_n \coloneqq$ topological sort of $\hat{G}(R_A, \prec^+ \cup \prec^\square)$;
    change $\coloneqq$ false;
  end
  $\rho \coloneqq$ select(Q_1);
  $\mathcal{I} \coloneqq$ applyRule(ρ);
  foreach $[\rho] \in A$ *with pred*$_{\hat{G}^+}([\rho]) = \emptyset$ **do** `updateActive(`$[\rho]$`)` ;
end
return $\mathcal{I}$;

Algorithm 4.4 receives as input any arbitrary knowledge base $\mathcal{K} = \langle R, \mathcal{D} \rangle$. At the start, we set G^+ to be a graph which records the positive reliance relationship between the rules. The interpretation $\mathcal{I}$ is assigned to the database $\mathcal{D}$. In addition, we maintain a set A of potentially active $\prec^+$-groups, which is initialized to $R/_{\prec^+}$ in Line 11. It is updated via the function `updateActive`. This function determines if a given group of rules $[\rho]$ can safely be considered as inactive. This is done by checking whether any of its members are applicable over the current interpretation and whether the given group has any potentially active predecessors. If this is not the case then $[\rho]$ is removed from A. Here, we also indicate that the partitioning should be recomputed by setting the global variable `change` to true. Since removing $[\rho]$ may prove that some of its successors are also inactive, we call `updateActive` on each of them as well. Before entering the main loop, `updateActive` is called on every $\prec^+$-group that does not have any predecessors. This will eliminate rules that are verifiably inactive over $\mathcal{D}$. In every iteration of the main loop, we first check whether there occurred any changes in the set of potentially active rules. If this is the case, we compute a new topological sorting of the rule graph that contains positive and restraint relations, but with nodes restricted only to potentially active rules. We then select the next rule ρ from the minimal partition and apply it over the current interpretation. After each step, we update the set A by calling `updateActive` for every potentially active $\prec^+$-group with no predecessors.

As with the other procedures provided thus far, we prove that it is a semi-fair chase algorithm.

Proposition 4.6. *Let $\mathcal{K} = \langle R, \mathcal{D} \rangle$ be a knowledge base and $\mathcal{I}_\infty$ the result of a finite run of Algorithm 4.4 on $\mathcal{K}$ Then there is no rule in R which is applicable over $\mathcal{I}_\infty$.*

Proof. We will prove this statement by showing that any $\prec^+$-group which is not present in the set of potentially active rules A is indeed inactive over the current interpretation. Since Algorithm 4.4 only stops once A is empty, no rule can be applicable over the returned result.

Assume for a contradiction that $\rho' \in [\rho]$ is a rule that is active over $\mathcal{I}$ and R. Then, there is a chase sequence $(\mathcal{I}_k) = \mathcal{I}_0, \mathcal{I}_1, \ldots$ with $\mathcal{I}_0 = \mathcal{I}$ such that ρ' was applied over $\mathcal{I}_k$ with the extended match h. Assume w.l.o.g. that ρ' is the first rule from $[\rho]$ applied in $(\mathcal{I}_k)$. Since $[\rho]$ was removed during `updateActive`, no member of $[\rho]$ is applicable over $\mathcal{I}$. Hence, h is not a match for ρ over $\mathcal{I}$ and it follows that there is some rule ρ'' applied over $\mathcal{I}_{k'}$ with $k' < k$ such that $\rho'' \prec^+ \rho'$. Because we assumed

that ρ' was the first rule applied from $[\rho]$, we have that $\rho'' \notin [\rho]$ and therefore that $\rho'' \in R \setminus \bigcup A$, since ρ'' is a predecessor of ρ' in $G(R, \prec^+)$. But then ρ'' could not have been inactive over $\mathcal{I}$ and R. □

The dynamic partitioned chase as described in Algorithm 4.4 requires that the applicability of every rule is known at all times in order to determine whether a group of rules should be considered inactive. However, such information might not be available in implementations of the chase procedure, as is the case in VLog. As a practical alternative, we can use positive reliances to provide a necessary condition for when a rule is applicable, which can be easily computed. The adjusted version of the algorithm for the 1-parallel chase used by VLog is shown in Algorithm A.1. The main difference to the original procedure is that it now maintains a set T of *triggered* rules. A rule is added to T, if it positively relies on a rule that was applied in the previous step while deriving new facts. Rules contained in T may therefore be thought of as potentially applicable rules. Since the 1-parallel application of a rule satisfies every one of its matches, we remove a rule from T immediately after it is applied. We Initialize T with every rule that does not have any predecessors in G^+. Note every rule that is not contained in T is not applicable over the current interpretation. We may therefore replace the check for applicability in the function `updateActive` with testing membership in T.

4.2.3 Order Within Strongly Connected Components

With the partitioned chase algorithms presented previously we obtain a high-level ordering of the given rule set, whereby some rules are exhaustively applied before others. But this leaves open the question of what strategy should be employed within groups of rules. We find that within strongly connected components, the optimization goals given at the start of the chapter cannot be met at the same time. We demonstrate this on the rule set from Example 4.1.

Example 4.6. Recall the following rule set R consisting of two rules which form a strongly connected component.

$$a(x, y) \to \exists v, w.\, h(x, v, w) \wedge b(y, v, w) \qquad (\rho_1)$$

$$b(x, y, z) \to h(x, z, y) \wedge b(x, z, y) \qquad (\rho_2)$$

Say we start with a database $\mathcal{D} = \{a(1, 1), a(2, 3), b(1, 2, 3)\}$. In this case, both ρ_1 and ρ_2 are applicable. However, satisfying either violates one of our optimization

goals. Starting with ρ_1, we obtain $\mathcal{I} = \mathcal{D} \cup \{h(1, n_v, n_w), b(1, n_v, n_w), h(2, m_v, m_w), b(3, m_v, m_w)\}$. Here, we are able to satisfy ρ_2 with a single parallel application, which only leads to one additional predicate-block for b and h. But at the same time we introduce the alternative match $\{x \mapsto 1, y \mapsto 1, v \mapsto 3, w \mapsto 2\}$. If, on the other hand, we begin with ρ_2, we avoid said alternative match but have to apply ρ_1 one additional time, splitting the facts belonging to b and h across more predicate-blocks. △

Hence, no generic strategy can be given. Which order of applying rules leads to a better performance in practice depends heavily on given knowledge base. Still, we give two alternatives to merely selecting the rule from the strongly connected component at random:

1. Positive First, which prioritizes rules from $\prec^+$-groups that have no active predecessor
2. Unrestrained First, which prioritizes rules that are not restrained by potentially active rules

The first strategy aims for a minimal amount of parallel rule applications by delaying the application of rules from $\prec^+$-groups that have a potentially active predecessor within its strongly connected component. This approach would be preferable in situations where the main performance bottleneck consists of frequent consolidation steps or joining of EDB-predicates. However, this might also indirectly reduce the number of alternative matches in the dynamic version of the partitioned chase. Once every rule from the prioritized group ceases to be applicable, it can be removed from consideration. This may remove certain dependencies, splitting the strongly connected component into multiple partitions sooner than would otherwise be the case. We implement this strategy by replacing the call to `select` the partitioned chase with a call to the following function.

```
1 Function selectPositiveFirst(RuleSet Q):
2     M_1, ..., M_n := topological sort of Ĝ(Q, ≺⁺);
3     return select(M_1);
```

The second strategy discussed here is a generalization of the Datalog-First chase. Since datalog rules do not contain any existential variables, there cannot be an alternative match w.r.t. to such a rule. Therefore, datalog rules cannot be restrained. But their products might still satisfy other rules, which in turn may pre-

vent alternative matches that would otherwise be caused by them. However, this is not unique to datalog rules since we can employ the same reasoning for any rule which is not restrained by rules that are potentially active. This leads to a strategy that is useful even in groups that do not contain any datalog rule.

Example 4.7. Here, we analyze the rule set given in Example 2.3 in more detail.

$$\mathsf{book}(x) \to \exists v, i.\, \mathsf{writtenBy}(x, v, i) \wedge \mathsf{author}(v) \qquad (\rho_1)$$

$$\mathsf{author}(x) \to \exists w, i.\, \mathsf{authorOf}(x, w, i) \wedge \mathsf{book}(w) \qquad (\rho_2)$$

$$\mathsf{authorOf}(x, y, j) \to \exists i.\, \mathsf{writtenBy}(y, x, i) \qquad (\rho_3)$$

$$\mathsf{writtenBy}(x, y, j) \to \exists i.\, \mathsf{authorOf}(y, x, i) \qquad (\rho_4)$$

The following reliance relationships hold between the rules of the above rule set: $\rho_1 \prec^+ \rho_2 \prec^+ \rho_1$ and $\rho_1 \prec^+ \rho_4 \prec^\square \rho_2 \prec^+ \rho_3 \prec^\square \rho_1$. It follows immediately that the rule set is not core-stratified and that every rule is contained in a single strongly connected component. As was the case in Example 2.2, repeatedly applying the rules ρ_1 and ρ_2 leads to an infinite cycle, which keeps producing new nulls and thereby alternative matches. But since ρ_3 and ρ_4 are not restrained by any rule, an early application of those rules can never lead to additional alternative matches, which allows us to prioritize them over ρ_1 and ρ_2. △

The strategy is implemented by using the following function to select the next rule.

```
1 Function selectUnrestraintFirst(RuleSet Q):
2     Q_R := {ρ ∈ Q | ∃ρ_r ∈ ⋃ A. ρ_r ≺□ ρ};
3     Q_U := Q \ Q_R;
4     if Q_U ≠ ∅ then
5         return select(Q_U);
6     end
7     else
8         return select(Q_R);
9     end
```

4.2.4 Termination

Given a knowledge base $\mathcal{K}$ that contains a core-stratified rule set, Proposition 4.1 ensures that every version of the partitioned chase terminates if any finite restricted chase sequence for $\mathcal{K}$ exists in the first place. This statement does not hold for arbitrary knowledge bases. Unfortunately, it is even the case that our strategy might exclude terminating runs entirely.

Example 4.8. Consider the following set of rules R.

$$b(x) \to \exists v, w.\, h(x, v, w) \wedge f(x) \wedge c(w) \qquad (\rho_1)$$
$$c(x) \to f(x) \qquad (\rho_2)$$
$$h(x, y, z) \to p(y, z) \qquad (\rho_3)$$
$$p(y, z) \to \exists v.\, p(z, v) \qquad (\rho_4)$$

Any application of ρ_4 leads to an infinite cycle that continuously produces new p-atoms. Take, for instance, the database $\mathcal{D} = \{b(1), c(2), h(1, 2, 2)\}$. Satisfying ρ_1 introduces the atom $h(1, n_1, n_2)$, which results in $p(n_1, n_2)$ after applying ρ_3. After this point, the chase is forced to produce infinitely many atoms of the form $p(n_2, n_3)$, $p(n_3, n_4), \ldots$ by endlessly applying ρ_4. However, we could have prevented the application of ρ_1 by prioritizing ρ_2. This would have lead to a finite universal model $\mathcal{I} = \mathcal{D} \cup \{f(1), p(2, 2)\}$. Crucially, ρ_2 positively relies on ρ_1 but does not restrain any of the other rules nor do any of them positively rely on ρ_2. Any topological sorting of $\hat{G}(R, \prec^+ \cup \prec^\square)$ would therefore prioritize ρ_1 over ρ_2 thereby forcing a non-terminating chase sequence. △

The above example also highlights that the generalized versions of the partitioned chase may exclude chase runs without alternative matches. We propose that a possible solution for this problem is to take a different type of reliance, which we call *blocking*, into consideration when deriving the application order.

Definition 4.2. A rule $\rho_1 = \varphi_1 \to \psi_1$ *blocks* a rule $\rho_2 = \varphi_2 \to \psi_2$, written $\rho_1 \prec^\dagger \rho_2$, if there exist interpretations $\mathcal{I}_a \subseteq \mathcal{I}_b$ such that

1. $\mathcal{I}_b$ was obtained by applying ρ_1 over $\mathcal{I}_a$ with the extended match h_1',
2. there is an unsatisfied match h_2 for ρ_2 over $\mathcal{I}_a$,
3. h_2 is a satisfied match for ρ_2 over $\mathcal{I}_b$,
4. there is no bijective substitution $\nu\colon \mathsf{vars}_\exists(\psi_2) \to h_1'(\mathsf{vars}_\exists(\psi_1))$ where we have $\psi_2 \nu h_{2\forall} = \psi_1 h_1'$.

Intuitively, a rule blocks another if the application of the first rule may satisfy a match of the second rule, given that they are both applicable at the same time. The last condition ensures that ρ_2 is not blocked by ρ_1 simply because ρ_1 derives essentially the same facts ρ_2 would have derived if it was applied first instead. In this case, there would be no reason to prefer the application of the ρ_1 over ρ_2. It also gets rid of the trivial case where a rule would block itself because applying it satisfies the corresponding match.

The concept of blocking and restraining a rule are closely related because the presence of an alternative match indicates that the prior application of the rule causing it would have satisfied the restraining rule. It is therefore not surprising that both reliances correspond in the majority of instances. But it is still the case that neither relation subsumes the other.

Example 4.9. Example 4.8 showed a case where $\rho_2 \prec^{\dagger} \rho_1$ but $\rho_2 \not\prec^{\square} \rho_1$. In the same set of rules ρ_3 and ρ_4 are an example of two rules where both relations correspond. The following pair of rules demonstrates a situation where $\rho_6 \prec^{\square} \rho_5$ but $\rho_6 \not\prec^{\dagger} \rho_5$.

$$a(x, y, z) \to \exists v_1, v_2.\, a(x, v_1, v_2) \wedge b(x) \qquad (\rho_5)$$

$$b(x) \to \exists w_1, w_2, w_3.\, a(x, w_1, w_2) \wedge c(w_3) \qquad (\rho_6)$$

△

Computing and integrating blocking reliances into the partitioned chase works analogously to positive and restraint reliances. To test whether a blocking reliance is present between two rules we can use similar algorithms as presented in Chapter 3. Integrating blocking into the generalized partition chase is then simply a matter of including the relationship into the topologically sorted rule graph. We conjecture that such an approach could prevent that terminating runs are excluded.

Since none of our tested rule sets suffered from non-termination we omit details regarding the implementation of blocking reliances and leave further theoretical considerations to future works.

Chapter 5

Evaluation

To evaluate our strategy, we implemented the dynamic partitioned chase described in the previous chapter as well as the algorithms that compute the reliances from Chapter 3 into the VLog reasoning engine. In all the experiments that were performed, we also compare the two strategies of selecting rules within strongly connected components outlined in Section 4.2.3. The implementation is based on a fork of VLog[1] from June 5, 2021 and is available online[2]. Since we only alter the order in which rules are applied, integrating our approach only requires surface-level changes to the existing architecture. We will therefore largely omit details regarding the implementation, but mention a few of the relevant aspects, for example additional optimizations not described in Chapter 3 and Chapter 4, while discussing the results of the experiments.

Section 5.1 describes the hardware on which the experiments have been conducted and introduces all of the knowledge bases used as benchmarks. We divide the actual evaluation of the results into three parts. Section 5.2 compares the time of materializing the knowledge bases achieved by the presented rule orders to VLogs original strategy. In Section 5.3 we focus on the ability of every strategy to compute cores and to avoid alternative matches. Lastly, we evaluate the performance of the algorithms for computing potential reliances between rules in Section 5.4. In particular, we are intrested in whether or not the amount of resources invested into the computation of reliances exceeds the improvement gained by employing a strategy based on them.

[1] `https://github.com/karmaresearch/vlog`
[2] `https://github.com/aannleax/vlog`

5.1 Experimental Setup

The experiments were conducted on a computer with a 2.7 GHz Intel Core i5 processor, 8 GB of RAM and a 2 TB HDD on Windows 10 (Build 19043). All of the software, this includes VLog and its dependencies, were compiled from source code using the MSVC compiler version 19.28.29335 for x64.

The knowledge bases we consider here mostly consist of the ones used by Benedikt et al. for benchmarking chase algorithms [5]. This includes:

- LUBM, which is an established ontology benchmark in the Semantic Web Community [15]
- STB-128 and ONTOLOGY-256, which are scenarios developed for data integration [2]
- DEEP, which was created as a stress test for reasoning engines by using rules able causing it to generate a large amount of facts [5]
- DOCTORS, which is a scenario used by Geerts et al. [12]

Furthermore, we consider REACTOME[3] and UNIPROT[4], which were used as part of the evaluation of VLog by Carral et al. [9]. These are real-world ontologies containing information about molecule interactions and protein sequences respectively. They were obtained by using data sampling algorithm based on random walks [19]. We also include UOMB[5], which is an extension of LUBM using a more complex ontology. In addition to the ones mentioned so far, we create our own knowledge base named CYCLE to highlight a specific inefficiency present in the original rule order used by VLog. It consists of the following rule set.

$$\mathsf{edb}_a(x, y) \to a(x, y) \qquad (\rho_1)$$

$$\mathsf{edb}_b(x, y) \to b(x, y) \qquad (\rho_2)$$

$$a(x, y) \wedge a(y, z) \to a(x, z) \qquad (\rho_3)$$

$$b(x, y) \wedge b(y, z) \to b(x, z) \qquad (\rho_4)$$

$$\mathsf{edb}_c(x, y, z) \wedge \mathsf{edb}_d(y, z, x) \wedge a(x, y) \wedge b(y, z) \to r(x, y, z) \qquad (\rho_5)$$

$$\mathsf{edb}_c(x, y, z) \wedge \mathsf{edb}_d(y, z, x) \wedge b(x, y) \wedge a(y, z) \to r(x, y, z) \qquad (\rho_6)$$

[3] `https://www.reactome.org/`
[4] `https://www.uniprot.org/`
[5] `https://www.cs.ox.ac.uk/isg/tools/UOBMGenerator/`

Dataset	Parameters
CYCLE	$N_c = 500$ and $N_t = 10$M
DEEP	Number of target-TGDs: 200
DOCTORS	Number of source-facts: 1M
LUBM	Number of universities: 100
ONTOLOGY	—
REACTOME	Sample size: 80%
STB	—
UNIPROT	Sample size: 10%
UOMB	Number of universities: 40

Table 5.1: Parameters used for each dataset (if available)

In the above rule set we have a $\prec^+$-cycle between the rule ρ_3 and ρ_4. Also, ρ_5 and ρ_6 positively rely on ρ_3 and ρ_4. Note that the last two rules contain bodies that require a relatively expensive join operation to compute. The original strategy used by VLog selects rules in the order they are given, wrapping around when reaching the last rule. On the given rule set, this would lead to the last two rules being executed as often as the cycle produces new facts. In particular, the join between the tables representing the edb-predicates edb_c and edb_d has to be performed each time. The partitioned chase strategy groups ρ_3 and ρ_4 into one partition and only moves on once said rules are inactive. This guarantees that ρ_5 and ρ_6 are only applied once. The databases are produced as follows. The facts corresponding to the predicates edb_a and edb_b form a chain of constants up to a given maximum length. So for example, the database includes the facts $\text{edb}_a(1, 2), \text{edb}_a(2, 3), \ldots, \text{edb}_a(N_c - 1, N_c)$. The tables corresponding to the predicates edb_c and edb_b are populated at random from the available constants until they contain a specified number N_t of entries.

All of the knowledge bases used can be obtained online[6]. Some of them are available in different versions, allowing for a variable amount of input facts or rules. The parameters were selected in such a way as to not exceed the limitations of the hardware used, especially in regard to the size of its main memory. They are shown in Table 5.1.

[6] `https://github.com/karmaresearch/Chasing-VLog`

Dataset	VLog		PosFst		UnresFst	
	Time [s]	IDB-Avg.	Time [s]	IDB-Avg.	Time [s]	IDB-Avg.
Cycle	70.94	8.00	17.26	4.50	17.11	4.5
DEEP	8.52	14.92	3.20	1.00	4.59	5.43
DOCTORS	3.33	—	3.27	—	3.29	—
LUBM	2.85	0.79	2.80	0.60	2.77	0.61
ONTOLOGY	2.95	1.80	2.92	0.50	2.95	0.50
REACTOME	2.83	0.78	2.83	0.51	2.80	0.61
STB	0.77	1.03	0.77	0.28	0.77	0.28
UNIPROT	2.18	0.76	2.02	0.65	2.02	0.64
UOMB	6.34	1.67	5.92	0.91	5.81	0.93

Table 5.2: Time of a full materialization of the given dataset in seconds and the average amount of applied IDB-rules

5.2 Time Measurements

Table 5.2 summarizes the results of measuring the run times of a full materialization of each of the chosen knowledge bases using the original rule order by VLog as well as the dynamic partitioned chase in both of its variants, namely Positive First (PosFst) and Unrestrained First (UnresFst). Each experiment represents the best value of three consecutive runs. The measurements only include the time for the materialization of the knowledge base and leave out the time needed to load the input facts into main memory. Here, we also exclude the time used for computing the reliance relations between the rules. We consider them separately in Section 5.4. Since we hypothesize that the number of parallel rule applications may affect run times, we also provide the average number of times an IDB-rule, i.e. a rule that contains at least one IDB-predicate in the body, has been applied during the chase. We excluded EDB-rules from this consideration because every one of them has to applied exactly once independently of the chosen rule order.

Overall, we can state that our approach in both of its variants offers a strict improvement over the original rule application strategy employed by VLog. In every instance, we were able to reduce or at least maintain the run times achieved by VLog. Also, we observe that we successfully lowered the number of rule applications in every experiment.

This effect is most pronounced in the knowledge base DEEP, where every IDB-rule was applied almost 15 times on average by VLog. With the strategy that prioritizes positive groups, we manage to apply every rule only once, which is possible

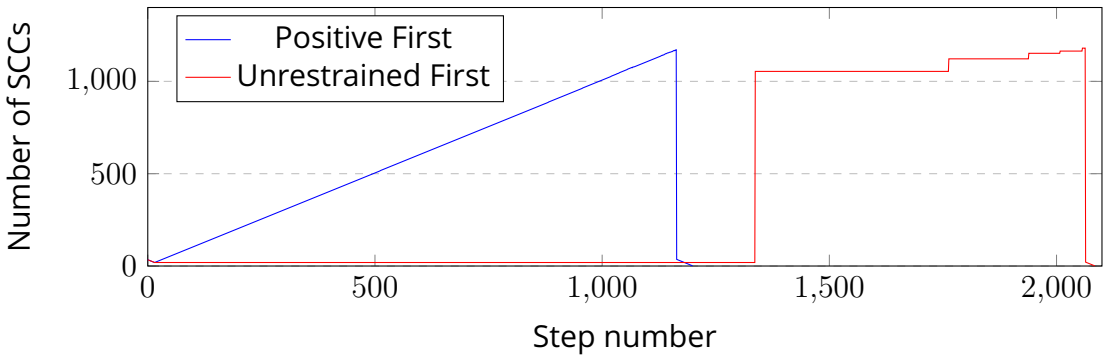


Figure 5.1: Number of strongly connected components in potentially active rules at each step of running the chase on DEEP

because the rule graph does not contain any $\prec^+$-cycles. This avoids unnecessary splitting of the facts into multiple predicate-blocks, which prevents costly consolidation operations. We therefore see a significant improvement in the run time of 63% compared to the original approach. Prioritizing unrestrained rules still reduces the amount of applications by a factor of 2.7. This leads to a time reduction of 49% over the original strategy.

The difference in behavior of the Positive First and Unrestrained First approach in the DEEP dataset is visualized in Figure 5.1. Here we show the number of the strongly connected components in the potentially active rule set, which are available at every step in the chase run. As we can see, we start out with all of the rules being contained within a small number of groups. Prioritizing $\prec^+$-groups without active predecessors splits them into several smaller components after each rule application. The number of strongly connected components therefore steadily increases until step 1163, where every rule is in its own group. At this point all of them immediately prove to be inactive and number of components collapses. In contrast, the Unrestrained First strategy spends a majority of its run time applying rules in a small number of large strongly connected components. Within those, we cycle through the available rules similar to VLog, which allows for a rule to be applied multiple times even in an acyclic $\prec^+$-graph. Only at step 1337 does the large group break into several smaller components. This happens five more times before each of the remaining rules is proven to be inactive and the number of groups collapses.

Another major performance improvement was achieved in the knowledge base CYCLE. In this case we successfully reduced the number of expensive joins performed by the reasoner. This resulted in time savings of 76% compared to VLog

Dataset	Strat.	Facts	PosFst			UnresFst		
			Sel.	New	Delta	Sel.	New	Delta
CYCLE	✓	0.4M	0	0	0	0	0	0
DEEP	✗	0.9M	1148	1015	+19K	2060	2032	-0.5K
DOCTORS	✓	0.8M	0	0	0	0	0	0
LUBM	✗	18.7M	3	0	0	1	0	0
ONTOLOGY	✗	5.7M	123	64	0	64	64	0
REACTOME	✓	11.4M	0	0	0	0	0	0
STB	✓	1.9M	0	0	0	0	0	0
UNIPROT	✓	24.2M	0	0	0	0	0	0
UOMB	✗	18.6M	21	7	+35K	8	6	-12K

Table 5.3: Comparison of the selection strategies regarding the number of derived facts and selection of restrained rules

for both strategies. Positive First and Unrestrained First coinside since the rule set does not contain any non-datalog rules.

In all other knowledge bases either no change in the materialization times was observed, or in the case of UOBM and UNIPROT only a minor improvement of 7%-8% was achieved. This is despite the fact that the number of rule applications was reduced by a factor of 3.6 and 3.7 for the knowledge bases ONTOLOGY and STB respectively. In every experiment except for CYCLE and DEEP the average number of IDB-rule applications using the original strategy did not exceed 1.8. In the case of LUBM, REACTOME and UNIPROT the average was even below 1.0. Based on this, we conjecture that the facts were not sufficiently fragmented accross multiple predicate-blocks as to trigger expensive consolidation steps or to otherwise significantly affect the time needed for joining or duplicate-elimination.

5.3 Cores and Alternative Matches

In this section, we evaluate the partitioned chase strategy based on its ability to avoid alternative matches and to reduce the amount of redundant facts. In Table 5.3 we show which of the chosen knowledge bases contain core-stratified rule sets as well as the number of IDB-facts produced by a full materialization using VLogs original implementation. For each variant of the partitioned chase, Table 5.3 also records the number of rules that were selected during the run but were restrained by another potentially active rule. We further track how many of those restrained rules actually lead to new derivations and the difference of the number of derived facts compared to the original strategy used by VLog.

As we can see, five of the nine knowledge bases considered in our experiments are core-stratified. In all of those cases, no rule that is restrained by a potentially active rules has been applied in both variants, hence guaranteeing that the produced model is a core. However, this did not lead to any change in the size of the produced model compared with VLog.

In the case of LUBM, the rule set of which is not core-stratified, both strategies were still able to produce a core model. Although restrained rules were selected by both variants, none of those lead to new derivations. But again, we observe no change in the number of derivations.

In the rule set of DEEP almost all of the rules restrain themselves. Therefore, the number of restrained rules selected is merely indicative of the total number of times any rule has been applied. The Unrestrained First strategy required 2084 steps to finish while Positive First terminated in only 1199 steps. We therefore see a corresponding amount of selected resrained rules in each variant.

Using the Unrestrained First strategy also did not lead to any significant advantage over using Positive First in the knowledge bases ONTOLOGY and UOMB. Although Unrestrained First is less likely to select a restrained rule during its run over both knowledge bases than Positive First, the number restrained rules which were applied and lead to new derivations still remains the same for ONTOLOGY and is only reduced by one in the case of UOMB.

In all of the experiments, the number of derived facts either did not change or only by an insignificant margin compared to the total amount of derivations. We attribute this to three possible factors. For one, a significant number of restraint reliances originate from datalog rules as shown in Table A.1. The rule set of REACTOME, for instance, does not contain any restraint relation between two non-datalog rules. But since VLog prioritizes the application of datalog rules, violations of restraints might be naturally rare. Secondly, the order of rules given in the input file might play a role in the effectiveness of the chosen strategy. VLog cycles through the rules in the order they are given. If a restrained rule often appears later in the file than the rule restraining it, such a strategy may prevent alternative matches without explicitly calculating them. This indeed holds true for most of the considered knowledge bases as seen in Table A.1. As an example take the knowledge base ONTOLOGY. Although most of the restraint relations are between non-datalog rules, a vast majority of the them follows the order as given in the input. Combined with the fact that in most experiments, each rule was

Dataset	Naive		Optimized		
	Calls	Time [ms]	Calls	Time [ms]	Frac. [%]
CYCLE	72	4	16	1	0.0
DEEP	2880000	3336	70806	244	3.6
DOCTORS	50	3	8	1	0.0
LUBM	36992	35	335	6	0.1
ONTOLOGY	559682	486	1527	23	0.5
REACTOME	722402	579	2504	28	0.8
STB	79202	72	255	8	0.6
UNIPROT	563952	440	671	31	0.9
UOMB	362952	274	1463	32	0.4

Table 5.4: Impact of computing the reliance relationship

only applied at most once or twice on average this again might lead to a situation where most of the restraint reliances are satisfied indirectly. Lastly, it might also be the case that alternative matches are simply rare in the considered knowledge bases independently of the chosen order. Recall that the violation of a restraint reliance only presents a necessary condition but is not sufficient for alternative matches to actually occur.

5.4 Computing the Reliance Relationship

Table 5.4 summarizes the cost of calculating the existence of a reliance relationship between the rules. In theory, the corresponding algorithms would have to be called for every pair of rules in the given rule set. However, we observe that a rule can only positively rely on another, if the second rule contains a predicate in its head which is also present in the body of the first rule. Similarly, a rule may only restrain another, if there exists a predicate which occurs in the head of both rules such that the corresponding atom contains an existential variable in the restrained rule. As an optimization, we record which predicates are present in the body or head of which rules via a hash map. Based on that, we only call the reliance algorithms on rule pairs that meet the condition laid out above. Although the reliance algorithms would have rejected non-unifiable rules early, this technique stills prevents the construction of costly auxiliary data structures. To show the impact of this optimization rule pairs in this manner, Table 5.4 distinguishes between the naive method, which calculates the relationships between every rule pair and the optimized method, which filters rule pairs as described above. We

record the total number of calls to both reliance algorithms and their overall run times for both methods. For reference, the last column shows the percentage of the time needed for computing the reliances using the filtered approach compared to the time it takes VLog to materialize the full knowledge base.

In every one of the considered knowledge bases filtering the rule pairs saved a majority of calls to the reliance algorithms from being performed. Except for the small rule sets in CYCLE and DOCTORS at least 95% of the calls were prevented. For these larger rule sets, this resulted in time savings between 84% and 96%.

Calculating the reliances of DEEP was by far the most time consuming compared to all other knowledge bases. In this case, the computation took almost 250 ms, which corresponds to about 3.6% of the time for materializing the full knowledge base by VLog. However, this is offset by the fact that we improved the run time by 49% or 63% depending on the variant using the result of this computation. In every other experiment, the total time taken for computing the reliance relationship does not exceed 1% compared to performing a full materialization.

We therefore conclude that computing the existence of reliances between rules is feasible in practice even for larger rule sets and does not greatly impact the overall run time. But it should be noted that the considered rule sets mostly consisted of simple rules, which only contained a small number of body and head atoms and rarely used a predicate multiple times. This severely limits the amount of unifiers that need to be tested. Hence, the result might not apply to knowledge bases containing a large amount of complex rules.

Chapter 6

Conclusion

6.1 Summary

In this thesis we presented strategies for selecting rules during a restricted chase run. We evaluated their effectiveness by implementing them into the rule reasoning engine VLog.

The proposed strategies are based on syntactic relations between rules called reliances. These indicate whether the application of one rule may trigger the application of another, or whether satisfying two rules in the wrong other might introduce redundancies into the chase result. We showed a method of testing whether such a relationship holds between two given rules. It is based on computing representative interpretations, which verify that the corresponding situation is possible in principle. Furthermore, we proved the correctness of the presented algorithms. Soundness was shown by confirming the validity of the constructed interpretations while completeness followed from the fact that they were constructed from the most flexible unifier between the rules.

Given these syntactic relations, we devised methods for selecting rules that follow the order implied by the reliances as best as possible. We began by considering the core-stratified case. Here, it is always possible to pick a rule order which does not introduce any redundancies while at the same time optimizing for the semi-naive evaluation strategy employed by VLog. This resulted in an algorithm that first divides the rule set into an ordered partitioning and then strings together multiple restricted chase sequences executed on each partition. We extended this strategy to arbitrary rule sets by relaxing the condition imposed on

the allowed partitionings and disregarding inactive rule during a chase run. In addition, we considered potential ways of selecting rules within the partitions. The Positive First strategy tries to minimize the number of rule applications, while Unrestrained First is a generalization of Datalog-First and aims to avoid deriving redundant facts.

We implemented our approach in both of its variants into VLog and evaluated its effectiveness on several benchmark and real-world knowledge bases. We saw a significant improvement in the run time of performing a full materialization in two of the nine tested cases. On every other knowledge base run times remained the same or improved by only a small margin. Furthermore, we did not manage to show any advantage of using our approach to remove redundant facts. However, our implementation demonstrates that calculating the reliance relationship is feasible in practice, taking up only a small portion of the computational resources.

6.2 Future Work

In this section we address open questions and directions for future work as well as some limitations of our current results.

One problem left unsolved is to provide a complete algorithm for restraint reliances. In its current state, certain cases of self-restrains cannot be detected. In this work, we also altered the original definition of the restraining relationship provided by Krötzsch. The impact of this change on the theoretical guarantees possible as well as its practical implications need further consideration. This also brings about questions of how to implement the detection of the original restraint reliances, since the algorithms shown so far seem not to be quite sufficient. Moreover, the characterization of both types of reliances assumes the restricted chase even though we use them in an environment where the 1-parallel chase is applied. Here, the benefits of adjusting the notions to this case need to be weighted against potential overhead such a change would cause.

As shown in Section 4.2.4, our approach to selecting rules might exclude terminating runs. We proposed a new type of reliance, called blocking, as a way to address this issue. It is an open question of how blocking relates to restraint reliances and whether or not including it into consideration when selecting rules is of any practical benefit.

Computing the reliances could be performed in a reasonable time frame in every of our experiments. However, further examples, in particular of knowledge bases containing more complex rules, are needed to assess whether this holds true in general. We would also like to repeat the experiments on better hardware which is able to handle more input facts. We suspect that this would lead to rules being applied more often by VLog, which may increase the impact of the suggested strategies.

Early experiments also revealed that the order of applying rules within strongly connected may have a huge impact on the overall run time even on rule sets which only contain datalog rules. Such cases cannot be resolved with our techniques. We would therefore like to investigate what other properties of rule sets can be used for optimizing their application order.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. 1995.

[2] Patricia C. Arocena et al. "The IBench Integration Metadata Generator". In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 108–119. ISSN: 2150-8097.

[3] Jean-François Baget et al. "On rules with existential variables: Walking the decidability line". In: *Artificial Intelligence* 175.9 (2011), pp. 1620–1654.

[4] C. Beeri and M. Y. Vardi. "The implication problem for data dependencies". In: *Automata, Languages and Programming*. Ed. by Shimon Even and Oded Kariv. Springer Berlin Heidelberg, 1981, pp. 73–85.

[5] Michael Benedikt et al. "Benchmarking the Chase". In: *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 37–52. ISBN: 9781450341981.

[6] Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. "A general Datalog-based framework for tractable query answering over ontologies". In: *Journal of Web Semantics* 14 (2012), pp. 57–83.

[7] David Carral, Irina Dragoste, and Markus Krötzsch. "Restricted chase (non)termination for existential rules with disjunctions". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*. Ed. by Carles Sierra. International Joint Conferences on Artificial Intelligence, 2017, pp. 922–928.

[8] David Carral et al. "Preserving Constraints with the Stable Chase". In: *Proceedings of the 21st International Conference on Database Theory (ICDT 2018)*. Ed. by Benny Kimelfeld and Yael Amsterdamer. Vol. 98. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Mar. 2018, 12:1–12:19.

[9] David Carral et al. “VLog: A Rule Engine for Knowledge Graphs”. In: *The Semantic Web – ISWC 2019*. Cham: Springer International Publishing, 2019, pp. 19–35. ISBN: 978-3-030-30796-7.

[10] Alin Deutsch, Alan Nash, and Jeff Remmel. “The Chase Revisited”. In: *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 149–158.

[11] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. “Data exchange: getting to the core”. In: *ACM Trans. Database Syst.* 30 (2003), pp. 174–210.

[12] Floris Geerts et al. “Mapping and cleaning”. In: *In ICDE*. 2014, pp. 232–243.

[13] Lukas Gerlach. *Chase-Based Computation of Cores for Existential Rules*. Sept. 2021.

[14] Gösta Grahne and Adrian Onet. “Anatomy of the Chase”. In: *Fundam. Informaticae* 157 (2018), pp. 221–270.

[15] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. “LUBM: A benchmark for OWL knowledge base systems”. In: *Journal of Web Semantics* 3.2 (2005). Selcted Papers from the International Semantic Web Conference, 2004, pp. 158–182.

[16] Pavol Hell and Jaroslav Nešetřil. “The core of a graph”. In: *Discrete Mathematics* 109.1 (1992), pp. 117–126.

[17] Kryštof Hoder and Andrei Voronkov. “Comparing Unification Algorithms in First-Order Theorem Proving”. In: *KI 2009: Advances in Artificial Intelligence*. Ed. by Bärbel Mertsching, Marcus Hund, and Zaheer Aziz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 435–443.

[18] Markus Krötzsch. “Computing Cores for Existential Rules with the Standard Chase and ASP”. In: *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020)*. Ed. by Diego Calvanese, Esra Erdem, and Michael Thielscher. IJCAI, 2020, pp. 603–613.

[19] Jure Leskovec and Christos Faloutsos. “Sampling from Large Graphs”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 631–636. ISBN: 1595933395.

[20] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. “Testing Implications of Data Dependencies”. In: *ACM Trans. Database Syst.* 4.4 (Dec. 1979), pp. 455–469.

[21] Thomas Neumann and Gerhard Weikum. “The RDF3X engine for scalable management of RDF data”. In: *The Vldb Journal - VLDB* 19 (Feb. 2010), pp. 91–113. DOI: 10.1007/s00778-009-0165-y.

[22] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41.

[23] Jacopo Urbani, Ceriel Jacobs, and Markus Krötzsch. “Column-Oriented Datalog Materialization for Large Knowledge Graphs”. In: *Proceedings of the 30th AAAI Conference on Artificial Intelligence*. Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 258–264.

[24] Jacopo Urbani et al. “Efficient Model Construction for Horn Logic with VLog”. In: *Proceedings of the 8th International Joint Conference on Automated Reasoning (IJCAR 2018)*. Ed. by Didier Galmiche, Stephan Schulz, and Roberto Sebastiani. Vol. 10900. LNCS. Springer, 2018, pp. 680–688.

Appendix

Algorithm A.1: dynamic partitioned chase (1-parallel version)

Input: KnowledgeBase $\mathcal{K} = \langle R, \mathcal{D} \rangle$
Output: Interpretation $\mathcal{I}_\infty$

Function `updateActive(`*Group* $[\rho]$`)`**:**
  if *pred*$_{\hat{G}^+}([\rho]) \cap A = \emptyset$ *and* $[\rho] \cap T = \emptyset$ **then**
    $A \coloneqq A \setminus \{[\rho]\}$;
    change $\coloneqq$ true;
    foreach $[\rho'] \in$ *succ*$_{\hat{G}^+}([\rho])$ **do**
      `updateActive(`$[\rho']$`)`;
    end
  end
$G^+ \coloneqq G(R, \prec^+)$;
$\mathcal{I} \coloneqq \mathcal{D}$;
$A \coloneqq R/_{\prec^+}$;
$T \coloneqq \{\rho \in R \mid \text{pred}_{G^+}(\rho) = \emptyset\}$;
change $\coloneqq$ true;
while $A \neq \emptyset$ **do**
  if *change* **then**
    $R_A \coloneqq \bigcup A$;
    $Q_1, \ldots, Q_n \coloneqq$ topological sort of $\hat{G}(R_A, \prec^+ \cup \prec^\square)$;
    change $\coloneqq$ false;
  end
  $\rho \coloneqq$ select(Q_1);
  $\mathcal{I} \coloneqq$ apply-1-parallel(ρ);
  $T \coloneqq T \setminus \{\rho\}$;
  foreach $[\rho] \in A$ *with pred*$_{\hat{G}^+}([\rho]) = \emptyset$ **do** `updateActive(`$[\rho]$`)` ;
  if *new facts were derived during this step* **then**
    foreach $\rho' \in$ *succ*$_{G^+}(\rho)$ **do** $T \coloneqq T \cup \{\rho'\}$;
  end
end
return $\mathcal{I}$;

Dataset	Restrained By		Appeared Earlier	
	Datalog	Non-Datalog	yes	no
CYCLE	0	0	0	0
DEEP	0	11297	5730	4742
DOCTORS	0	1	1	0
LUBM	37	0	24	13
ONTOLOGY	103	291	313	17
REACTOME	16	0	16	0
STB	12	41	50	3
UNIPROT	31	3	29	5
UOMB	131	6	91	46

Table A.1: Shows how often a rule has been restrained by a datalog/non-datalog rule and how many times the restraining rule appeared earlier in the input file