

The Chase in Lean - Crafting a Formal Library for Existential Rule Research

Lukas Gerlach

Knowledge-Based Systems Group, TU Dresden, Germany
lukas.gerlach@tu-dresden.de

Abstract

The chase is a sound, complete, but possibly non-terminating algorithm for reasoning with existential rules (aka. tuple-generating dependencies), a highly expressive knowledge representation language. Although the procedure appears simple, research on theoretical properties and optimization for practical implementations has grown to a point where verifying correctness and reproducing proofs becomes challenging and intuition can sometimes be misleading. Lean is a purely functional programming language and interactive theorem prover whose community actively develops formal libraries for mathematics (Mathlib) and computer science (CSLib). In this work, we present our own endeavor of crafting a Lean framework around existential rules and the chase. We discuss design decisions concerning the nuances of chase definitions commonly found in the literature and show how these translate into Lean. To illustrate the framework’s capabilities using known results, we show that the result of a chase is a universal model and outline the formalization for proving that without so-called “alternative matches” it is even a core. Beyond existing literature, we unify sufficient chase termination conditions in the likeness of Model-Faithful Acyclicity (MFA) into a common framework while also adding support for constants in rules.

1 Introduction

The chase (Beeri and Vardi 1984) is a common reasoning algorithm in knowledge representation and reasoning. It computes a bottom-up materialization for a given knowledge base consisting of a set of existential rules (Baget et al. 2011) and a database. After an introductory example, we discuss properties of the chase that we pick as candidates for a machine-verifiable formalization, motivate the need for formal verification in this setting, and give an overview of our specific contributions made in building a Lean library.

Example 1. Consider the database $D = \{P(a, b)\}$ and the singleton rule set R :

$$P(x, y) \rightarrow S(y) \vee \exists z. P(y, z) \wedge P(z, y)$$

A chase of $\langle R, D \rangle$ is a possibly infinite tree of fact sets that starts on the root D and tries to “apply” a rule to each node to obtain its children, one for each disjunct. Depending on the variant of chase algorithm, there are different conditions for when a node receives children. For example, in the restricted (aka. standard) chase, if all rules are already satisfied in a node n , then n has no children.

In this example, the root is $\{P(a, b)\}$ and has two children $\{P(a, b), S(b)\}$ and $\{P(a, b), P(b, f_z^{1,1}(b)), P(f_z^{1,1}(b), b)\}$. Note that both of these children satisfy the single rule in R , and therefore they are the leaves of the tree.

We use Skolem terms $(f_z^{1,1}(b))$ as a naming convention for “fresh” terms introduced for existential variables (Marnette 2009). The function symbol $f_z^{1,1}$ uniquely identifies the rule id (1), the (zero-based) disjunct index (1) and the variable (z). The arguments are the terms for the variables that occur on both sides of the implication (i.e. $y \mapsto b$).

1.1 Interesting Properties of the Chase

The result of the chase, i.e. the set of all fact sets that can be built by taking the union of all fact sets along a branch of the chase tree, is a *universal model set* (Bourhis et al. 2016). In the absence of disjunctions, the result is a single fact set, which is a universal model (Deutsch, Nash, and Rummel 2008). Universal model sets are of interest for conjunctive query answering (Abiteboul, Hull, and Vianu 1995).

Beyond universality, it is desirable to create the smallest possible models, called “cores”. While the core chase (Deutsch, Nash, and Rummel 2008) is able to achieve this in theory, it does so by finding homomorphisms that allow collapsing fact sets into substructures, which is infeasible in practice. Fairly recent work coins the term “alternative match” for redundancies introduced during the chase. Avoiding alternative matches directly yields a core without the need for any additional computations (Krötzsch 2020).

With these nice properties, there is a catch: chase termination. For example, for the rule $R(x, y) \rightarrow \exists z. R(y, z)$ and the database $\{R(a, b)\}$, the chase will produce an infinite branch. In the general case, we cannot decide if the chase terminates for a given set of rules (Gogacz and Marcinkowski 2014; Grahne and Onet 2018). There is a whole line of research defining classes of rule sets with guaranteed chase termination (Fagin et al. 2005; Krötzsch and Rudolph 2011; Baget et al. 2014; Karimi, Zhang, and You 2021; Hanisch and Krötzsch 2024). We will take a closer look at Model-Faithful Acyclicity (MFA) (Cuenca Grau et al. 2013), which also gives rise to more specialized notions like Disjunctive MFA (DMFA) (Gerlach and Carral 2023b) and Restricted MFA (RMFA) (Carral, Dragoste, and Krötzsch 2017). These are some of the most general but also most technical sufficient conditions for chase termination.

1.2 The Need for Formal Verification

Especially for chase termination, intuitions often seem simple but start to dissolve when taking a closer look. Let us consider two interesting examples with subtle but crucial mistakes. (A) Claim: A particular variant of chase termination is Σ_1^0 -complete (aka. RE-complete) (Grahne and Onet 2018). This problem was shown to be Π_2^0 -hard later (Carral et al. 2022, Proposition 42) and recently proven to be Π_1^1 -complete (Carral et al. 2025). As discussed by the authors, there seems to be an intuitive semi-decision procedure. Only taking a closer look reveals that this does not work because of the “fairness” requirement of the chase. (B) Claim: Restricted Model-Faithful Cyclicity (RMFC) guarantees chase non-termination (Carral, Dragoste, and Krötzsch 2017). There is a counterexample for the original proof idea. This was discovered and fixed later when Restricted Prefix Cyclicity (RPC) was introduced as a more refined notion (Gerlach and Carral 2023a). Similar to MFA-like conditions, RMFC and RPC have surprisingly simple intuitions but extremely technical definitions, which makes catching (subtle) mistakes very difficult.

Besides theoretical research, practical reasoning systems for existential rules and other Datalog-like languages have become complex pieces of software with many additional features and performance optimizations that go far beyond the theoretical description of the chase (Nenov et al. 2015; Urbani, Jacobs, and Krötzsch 2016; Jordan, Scholz, and Subotic 2016; Ivliev et al. 2024). One of the main selling points of these systems is that they are inherently trustworthy because of the conceptual simplicity of rule-based reasoning. But with more complex optimizations, more parts need to be trusted. Formal verification can ensure a high degree of trust but fully verifying state of the art systems is extremely challenging. However, at least for (stratified) Datalog, fully verified systems have been implemented in the Rocq theorem prover (Benzaken, Contejean, and Dumbbrava 2017; Dumbbrava 2016; Bonifati, Dumbbrava, and Arias 2018). Other works also describe Datalog semantics in various theorem provers to formalize security logics (Whitehead 2007) (Rocq), verify specific Datalog programs used in program analysis (Schlichtkrull, Rydhof Hansen, and Nielson 2024) (Isabelle/HOL), or check certificates provided by a Datalog engine to validate individual reasoning results (Tantow et al. 2025) (Lean). To the best of our knowledge, none of these or other works consider existential rules though.

1.3 Our Contribution – A Lean Library

In this work, we present a formal library of theoretical results around existential rules and the chase, written in Lean (de Moura and Ullrich 2021). We are inspired and motivated by existing libraries on mathematics¹ (Baanen et al. 2026) and (other) computer science topics² (Barrett et al. 2026) that are actively developed by the Lean community. Our primary goal is to start a similar effort for theoretical research on existential rules and the chase, with the clear benefit of having machine-verifiable and extensible results.

As a fortunate side effect, the mathematical description of the chase on existential rules also lays the foundation for the verification of specific implementations and real-world systems. For example, a (verified) reference implementation can be built on top of our project, directly in Lean.

Lean is based on dependant type theory and is a purely functional programming language where propositions are types. Proofs are in turn elements of such types. That means, the Lean kernel verifies our proofs by type checking. To prove proposition Q from proposition P , we write a function that takes an element of P and transforms it into an element of Q . The use of “tactics” allows us to write proofs in a more human readable format. Still, the proofs are meant to be explored in an interactive fashion, where a widget in the code editor shows the current hypotheses and the proof goal for the current line. In this way, one can “step” through the proof line by line. For a comprehensive view on Lean, we refer the interested reader to the official resources³.

In the upcoming sections, we present our key ideas behind crafting a Lean library for existential rules and the chase and demonstrate its capabilities by proving known properties mentioned above. We also generalize and extend some of them. Since the code is already thoroughly documented, we focus less on the specific implementation but rather on conceptual decisions and insights that do not only apply to Lean. We phrase main takeaways in *remark* environments. In the early sections we cover the following questions:

- How do we introduce “fresh” terms? (Section 2)
- When is a rule applicable? (Section 3)
- How can we view the chase as an infinite structure but still easily induce over its elements? (Section 4)

We then show the following results, again using *remarks* to highlight how earlier design decisions come into play.

- The chase result is a universal model set (Section 5).
- A chase without alternative matches yields a universal model that is a *strong core*, generalizing the original theorem, which uses *weak cores* (Section 6).
- We generalize the idea behind (D/R)MFA into a common framework with support for constants in rules and prove that chase termination is guaranteed (Section 7).

This paper is best read while exploring the formalization in a code editor with a Lean extension installed. This is probably easiest to setup in VSCode. Also, throughout the code, doc comments provide additional detail on individual results and sometimes larger parts of the formalization. Auto-generated documentation is available online at <https://monsterkrampe.github.io/Existential-Rules-in-Lean/>. All of the code (about 19000 lines in total) is available on GitHub where many symbols in this PDF directly link to.

- github.com/monsterkrampe/Basic-Lean-Datastructures
- github.com/monsterkrampe/Possibly-Infinite-Trees
- github.com/monsterkrampe/Existential-Rules-in-Lean

¹<https://github.com/leanprover-community/mathlib4>

²<https://github.com/leanprover/cslib>

³<https://lean-lang.org/learn/>

2 Terms, Atoms, and Rules

Towards the end of this section, we want to be able to formalize (disjunctive) (existential) rules. Here, we also discuss our first important design decision of using Skolem terms instead of (labelled) nulls (Remark 1). Traditionally, a rule is an expression of the form

$$\forall \vec{x}, \vec{y}. B[\vec{x}, \vec{y}] \rightarrow \bigvee_{i=1}^n \exists \vec{z}_i. H_i[\vec{y}_i, \vec{z}_i]$$

where $\vec{x}, \vec{y}, \vec{z}_1, \dots, \vec{z}_n$ are pairwise-disjoint lists of variables, $\vec{y}_1, \dots, \vec{y}_n$ are (not necessarily disjoint) lists of variables such that $\vec{y} = \bigcup_{i=1}^n \vec{y}_i$, and B (body) and $H_1, \dots, H_n$ (head) are function-free conjunctions of atoms.

In Lean, we simplify this into the following structure for `Rules`. Existential variables are identified implicitly: we treat variables as existentially quantified if they occur somewhere in head but not in body.

```
structure Rule where
  id : Nat
  body : FunctionFreeConjunction sig
  head : List (FunctionFreeConjunction sig)
  -- the List represents a disjunction
```

To formalize the basic building blocks, we start with *predicate symbols, variables, and constants*. In Lean, we capture these in a `Signature`, usually called `sig`. The signature is a structure of three arbitrary types `C`, `V`, `P` and an arity function `P → Nat`. As we saw in Example 1, we also need to be able to introduce fresh terms.

Remark 1. *Picking fresh elements from a (countably infinite) set of (labelled) nulls is a popular option. When “applying” a rule, the nulls picked for the existential variables must not occur anywhere else. The problem is that picking is a magical procedure. We know that it is safe to do since we have a countably infinite set of nulls at our disposal. But implementing a specific procedure or even only formally stating that a fresh null does not occur “before” would require us to keep some global state. Therefore, we use Skolem terms as a naming convention for nulls, which convey only local information and are “fresh by design”.*

We identify Skolem function symbols (`SkolemFS`) by a rule, a `disjunctIndex`, i.e. the index of the head disjunct in the rule, the (existential) variable that gets Skolemized, and the arity of the function symbol. The function symbols then receive the frontier of the rule as the arguments to construct a Skolem term. The *frontier* is the list of variables that occur in both head and body of the rule. This makes the Skolem terms unique. This way of Skolemizing with the frontier is also called *semi-oblivious* opposed to *oblivious* Skolemization where all body variables would be used. The semi-oblivious idea is that values used for non-frontier body variables are irrelevant to the result of the rule application.

```
structure SkolemFS (sig : Signature) where
  ruleId : Nat
  disjunctIndex : Nat
  var : sig.V
  arity : Nat
```

We define a variety of inductive types to represent different kinds of terms such as `VarOrConst`, `SkolemTerm`, and `GroundTerm`⁴. Instead of having a common term type, this more fine-grained approach limits the number of cases that we need to consider in proofs. Different kinds of atoms can now be introduced as variants of the following `GeneralizedAtom` where `T` represents an arbitrary term type. `Atoms`, `FunctionFreeAtoms`, `Facts`, and `FunctionFreeFacts` are all `GeneralizedAtoms` using `SkolemTerms`, `VarOrConsts`, `GroundTerms`, and `sig.C` as their term type, respectively.

```
structure GeneralizedAtom
  (sig : Signature) (T : Type u) where
  predicate : sig.P
  terms : List T
  arity_ok : terms.length =
    sig.arity predicate
```

A `FunctionFreeConjunction` is consequently a list of `FunctionFreeAtoms`, which gives us the essential ingredient for the `Rule` definition above. To define a `KnowledgeBase`, which is going to be the input of the chase, we still require `FactSets`, `Databases` (finite sets of function free facts), and `RuleSets`. These are readily defined with the following basic `Set` definition: `def Set (α : Type u) := α → Prop` That is, a set is a function that takes a potential element and returns whether it is a member. A `Set` is `finite` if there exists a list with the same elements.⁵ For `RuleSets` we make the technical demand that rules within the set are uniquely identified by their id, formally: if two rule ids are equal, then so are the rules.

3 Triggers and Obsolescence

As a precursor to the chase, we need to model what it formally means to “apply” a rule. The goal of an application is to satisfy the rule in the most general way. A major question to ask here is: when should a rule be applied? This will dictate which variant of the chase algorithm we model. That is, we could also ask: Do we want to model the Skolem or restricted (aka. standard) chase? (Remark 2) Before we answer this question, we cover the basic definitions of *triggers* captured in a structure called `PreTrigger`.

```
structure PreTrigger (sig : Signature) where
  rule : Rule sig
  subs : GroundSubstitution sig
```

`GroundSubstitutions` are (total) functions from `sig.V` to `GroundTerms`. A `PreTrigger` `trg` derives new facts in its *result*, i.e. `trg.mapped_head`, by instantiating the Skolemized variant of `trg.rule` with `trg.subs`. Thereby, it produces a list of `Facts` for each head disjunct.

In the context of any chase variant, a `PreTrigger` needs to be `loaded` for a given `FactSet F`. That is, `trg.mapped_body`, which results from applying `trg.subs` to `trg.rule.body`, needs to be contained in `F`. Depending on the chase variant, a trigger can be *obsolete*

⁴The definition for `GroundTerms` is a bit more technical but abstracted away to look like an inductive type from the outside.

⁵List is inductively defined and therefore each list is finite.

in different ways. The two variants we want to consider here are *SkolemObsolescence* (from the Skolem chase) and *RestrictedObsolescence* (from the restricted chase). *SkolemObsolescence* marks a trigger trg as obsolete for a *FactSet* F , if the result for one head disjunct is already contained in F . *RestrictedObsolescence* additionally checks if such a containment can be achieved by allowing a different mapping of the existential variables in trg.rule.head .

Example 2. *Reconsidering Example 1, we observe that the root node with fact $P(a, b)$ yields two fact sets through the result of the *PreTrigger* trg with the single *Rule* in R and a *GroundSubstitution* that maps x to a and y to b . The trigger trg is *loaded* for $\{P(a, b)\}$ since the instantiated body of trg.rule is $P(a, b)$. Additionally, trg is obsolete for neither *SkolemObsolescence* nor *RestrictedObsolescence* since trg.rule.head is not satisfied by the single fact $P(a, b)$. On the other hand, the *PreTrigger* trg' mapping x to b and y to $f_z^{1,1}(b)$ is *loaded* but obsolete with respect to *RestrictedObsolescence* for the fact set $\{P(a, b), P(b, f_z^{1,1}(b)), P(f_z^{1,1}(b), b)\}$ since its head is satisfied by the facts $P(f_z^{1,1}(b), b)$ and $P(b, f_z^{1,1}(b))$. For *SkolemObsolescence*, trg' is not obsolete for this fact set but the trigger mapping x to a and y to b would be obsolete for the fact set $\{P(a, b), S(b)\}$.*

Despite the obvious differences of the specific conditions, their behavior has certain commonalities that we may extract into a more general class of conditions.

Remark 2. *Instead of focusing only on Skolem or restricted obsolescence (and therefore the respective chase), we introduce a generalizing *ObsolescenceCondition* that only demands four essential properties. (1) Subset monotonicity: if a trigger is obsolete for a fact set, then it is obsolete on all supersets. (2) If a trigger is obsolete, it is satisfied (under first-order logic semantics). (3) If the trigger result already occurs in the given fact set, then it is obsolete for the generic condition. (4) If two triggers are equivalent (*equiv*), then one is obsolete if and only if the other is obsolete. Note that the most liberal condition that still fulfills (3) is *SkolemObsolescence* and the most restrictive condition that still fulfills (2) is *RestrictedObsolescence*.*

Now, a *Trigger* is a *PreTrigger* with a fixed *ObsolescenceCondition*. We say that a *Trigger* is *active* (for a fact set) if it is *loaded* and not obsolete with respect to its obsolescence condition. Finally, an *RTrigger* is subtype of *Trigger* enforcing that the trigger’s rule stems from a given rule set. For example, in a chase for a given knowledge base K , it only makes sense to consider triggers that feature rules from K and it is convenient for us to capture this property directly in the trigger type instead of demanding it explicitly in all places.

Let us briefly mention two other popular chase variants that we do not yet capture by generalizing obsolescence: the oblivious chase and the core chase. Both differ in ways other than obsolescence. To model the oblivious chase, the Skolem terms would need to take all body variables as arguments (not only trg.rule.frontier).

SkolemObsolescence then gives the oblivious behavior. The core chase uses restricted obsolescence but on top of computing trigger results, the resulting fact sets need to be condensed to their “core” during the chase. In the future, we plan to introduce additional generalizations to be able to capture all four chase variants in a common framework.

4 The Chase as a Coinductive Structure

As briefly outlined in Example 1, the intuitive idea behind the chase is to satisfy all rules in a given *KnowledgeBase* by iteratively and exhaustively adding the results of active triggers to the previously produced *FactSets* based on the initial *Database*. This procedure yields a possibly infinite tree of finite degree. In the special case of *deterministic* rules, where each rule has exactly one disjunct, this tree only consists of a single branch. An important design decision will be to take a coinductive view on the chase (Remark 3).

4.1 Chase Derivations and Branches

Works that are not concerned with disjunctions usually model the chase as an infinite sequence of fact sets. Even in the disjunctive setting, we can characterize individual branches of a chase tree independently of the tree structure. For simplicity, we will mainly discuss this view. Lifting the ideas to a whole tree structure is not much harder. Let us define such a chase branch on paper first, similar to definitions one would usually find in the literature.

Definition 1. *For a knowledge base $\langle R, D \rangle$, a chase branch is a sequence of fact sets $F_0 = D, F_1, \dots$ such that*

- *for each $i \geq 0$, F_{i+1} results from F_i by adding the result of a disjunct of a trigger that is active for F_i , and*
- *for each trigger λ , there is some F_i such that, for each $j \geq i$, λ is not active for F_j .*

Let us reflect on what an infinite sequence really is and, more importantly, what it is *not*. An infinite sequence is *not* an inductive type. Elements of inductive types (like lists) are always finite. Think of the natural numbers: a number is either zero or the successor of a number. While there are infinitely many natural numbers, each natural number is finite. The elements of an inductive type are the elements of the least fixed point of everything that can be built from the constructors. What we want instead is the greatest fixed point, i.e. coinductively defined lists. To model this, we build a small framework around what we call *PossiblyInfiniteList*.⁶ This includes basic access functions such as *head* and *tail*, a suffix relation ($<:+$) and a recursion/induction principle to show properties for elements of the list (*mem_rec*).

Based on this insight, we also want to treat the chase branch as a coinductive data structure. But this does not work with Definition 1. We currently force chase branches to start on a database, which means that (most) subbranches are no proper chase branches, since their first fact set may already contain Skolem terms. Therefore, we define a *chase derivation* where we lift this restriction. The other properties are the same as in Definition 1.

⁶Mathlib provides very similar machinery in `Stream’.Seq`.

```

structure ChaseDerivation (obs : ObsolescenceCondition sig) (rules : RuleSet sig) where
  branch : PossiblyInfiniteList (ChaseNode obs rules)
  -- The derivation needs to contain an initial fact set.
  isSome_head : branch.head.isSome
  -- Each fact set is produced by a trigger.
  triggers_exist : ∀ n : Nat, ∀ before ∈ (branch.drop n).head,
    let after := (branch.drop n).tail.head
    (exists.trigger.opt.fs obs rules before after)
  -- The used triggers are active.
  triggers_active : ∀ n : Nat, ∀ before ∈ (branch.drop n).head,
    ∀ after ∈ (branch.drop n).tail.head,
    ∃ orig ∈ after.origin, orig.fst.val.active before.facts
  -- Every trigger is eventually inactive.
  fairness : ∀ trg : (RTrigger obs rules), ∃ i : Nat,
    (∃ node ∈ (branch.drop i).head, ¬ trg.val.active node.facts)
    ∧ (∀ j : Nat, ∀ node2 ∈ (branch.drop i).tail.get? j, ¬ trg.val.active node2.facts)

```

Listing 1: Definition of a `ChaseDerivation` based on `PossiblyInfiniteList` and four conditions.

Remark 3. We treat a `ChaseDerivation` as a coinductive data structure. From our point of view, this resonates well with proofs about the chase usually carried out on paper only that the formal basics are rarely explicitly discussed. For example, when showing properties of fact sets in the chase via induction, it might not always be clear what we induce over. We could use the index within the chase but it is more convenient to consider only the following two cases: the fact set is the initial one or it results from a trigger application. This induction principle is something we can define using our coinductive framework (see `mem_rec`).

Our framework around `ChaseDerivations` offers many convenience features inspired by, but also going beyond, the underlying `PossiblyInfiniteList`.

- Accessor functions such as `head`, `next`, and `tail`.
- A suffix relation on chase derivations (`IsSuffix`, where `cd2 <:+ cd` reads as: “`cd2` is a subderivation of `cd`”).
- A recursion/induction principle to show properties over all members of the chase derivation (`mem_rec`).
- A total order on derivation members (`predecessor`, where `node ≤ node2` reads as: “`node` occurs in the chase before `node2`”). A strict version (`<`) also exists.

Listing 1 shows the full `ChaseDerivation` definition⁷. Note the similarities to Definition 1. The associated conditions are still expressed in the vocabulary of the underlying `PossiblyInfiniteList`, which leaves them quite convoluted. With the new vocabulary, we show theorems that express the conditions in a more accessible way. For example, we restate fairness in `fairness'` and `fairness_prec` in terms of suffixes and the predecessor relation, respectively. One can see from the definition that the `ChaseDerivation` is generic over an `ObsolescenceCondition`, which was introduced in the last section. Furthermore, maybe unexpectedly,

⁷In reality, the `ChaseDerivation` extends an even more general `ChaseDerivationSkeleton`. Listing 1 shows the combined structure.

the derivation is not based on a possibly infinite list of `FactSets` but `ChaseNodes` instead. A `ChaseNode` stores a `FactSet` and an optional `RTrigger`. So each chase step remembers not only *what* was derived but also *how*. A `ChaseBranch` as in Definition 1 is a `ChaseDerivation` for a `KnowledgeBase` where the first fact set is forced to be the `Database`.

Let us now discuss some important properties about `ChaseDerivations` and `ChaseBranches`. We already mentioned that we define a total order on the `ChaseNodes` of the derivation. But so far it might not be obvious that this is possible. Could the same `ChaseNode` not occur multiple times in the same derivation? Fortunately, no! The essential theorem for this shows that the `head` can never also be a member of the `tail`. Assuming for a contradiction that this was the case, the copy of head was introduced by a trigger, but then the trigger cannot be active anymore since all the derived facts already occur in the original head. We rely on property (3) from Remark 2 here.

Furthermore, we mentioned in Remark 1 that Skolem terms are “fresh by design”. We now want to prove this along a chase branch. What we show is slightly different but entails the desired property: if a Skolem term for a trigger λ occurs in a `ChaseNode`, then the whole result of λ is already in this chase node. This means if a Skolem term is already present, then every trigger that would introduce it (again) is already obsolete, which ensures freshness. Note that this is not quite the case for derivations instead of branches, since the term in question could occur in the first fact set instead of resulting from a trigger.

The most important property of a chase branch is that its result, defined as the union of all fact sets, is a model (`result_models_kb`). Without disjunctions, one obtains a stronger result showing that the result of a chase branch on deterministic rules is a *universal model*, meaning that it is one of the most general ones (`detCBResultUniversallyModelsKb`). Our proof for this depends on a corresponding result on chase trees, which we discuss at length in Section 5.

Example 3. Reconsidering Example 1 and our insights from Example 2, we can construct a finite *ChaseBranch* for *RestrictedObsolescence* that contains exactly $P(a, b), P(f_z^{1,1}(b), b), P(b, f_z^{1,1}(b))$ in its result. We can also construct an infinite *ChaseBranch* for *SkolemObsolescence* that derives an infinite “chain” of P relations including their inverses except for $P(b, a)$. Note that both of these results are models of the input knowledge base. Furthermore, there is a *ChaseBranch* producing only $P(a, b), S(b)$ for both obsolescence conditions, which also models the knowledge base.

4.2 Tree Derivations and Chase Trees

TreeDerivations and *ChaseDerivations* are similar. We mainly exchange the *PossiblyInfiniteList* for a *FiniteDegreeTree*. We provide a similar coinductive framework to access the root, the (immediate) child nodes and child trees, as well as a subtree relation and a recursion/induction principle for members. The only drawback we have is the predecessor relation. Instead of defining the relation directly on the *ChaseNodes*, we now need to take their addresses in the tree into account to obtain a proper order. This is because the same node can now occur multiple times in the tree, just not along the same branch. The address is an easy way to tell the different occurrences apart. The predecessor relation is then merely the prefix relation of the addresses. Just like the *ChaseDerivation*, the *TreeDerivation* does not enforce starting on a *Database*. In a similar way that *ChaseBranches* are defined for *ChaseDerivation*, we also define *ChaseTrees* based on *TreeDerivations*.

Each branch in tree of the *TreeDerivation* is shown to be a proper *ChaseDerivation* and the set of branches for the *TreeDerivation* consists of all *ChaseDerivation* that correspond to a branch in the tree of the *TreeDerivation*. We can now define the *TreeDerivation* result as follows.

```
def result (td : TreeDerivation obs rules) :  
  Set (FactSet sig) :=  
  td.branches.map ChaseDerivation.result
```

For the special case of a *ChaseTree*, it is easy to show that every fact set in the result is a model of the underlying knowledge base since we already know that the result of each *ChaseBranch* is a model (see *result_models_kb*).

Example 4. Based on Example 3, we can see that there is a (single) *ChaseTree* for *RestrictedObsolescence* and the respective knowledge base. This tree has a root node with $P(a, b)$ and two children adding $S(b)$ or $P(f_z^{1,1}(b), b), P(b, f_z^{1,1}(b))$, respectively. The result therefore is the set with the two *FactSets*: $\{P(a, b), S(b)\}$ and $\{P(a, b), P(f_z^{1,1}(b), b), P(b, f_z^{1,1}(b))\}$. We already realized earlier that both are models of the knowledge base.

The next section focuses entirely on showing a universality property for the *ChaseTree* result.

5 Chase Trees yield Universal Model Sets

The universality property of a *ChaseTree* is as follows. A set of models $\mathcal{U}$ is a *universal model set* if for each model M , there is a model $U \in \mathcal{U}$ such that there is a homomorphism from U to M . So we aim to show that the *ChaseTree* result is a universal model set.

```
theorem chaseTreeResultIsUniversal  
  (ct : ChaseTree obs kb) :  
  ∀ (m : FactSet sig), m.modelsKb kb →  
  ∃ (fs : FactSet sig)  
    (h : GroundTermMapping sig),  
    fs ∈ ct.result ∧ h.isHomomorphism fs m
```

Example 5. Recall the result from the *ChaseTree* discussed in Example 4, that is the set $\mathcal{U}$ with the two fact sets $\{P(a, b), S(b)\}$ and $\{P(a, b), P(f_z^{1,1}(b), b), P(b, f_z^{1,1}(b))\}$. We want to argue intuitively that $\mathcal{U}$ is universal. Every model for the input knowledge base needs to contain $P(a, b)$ and satisfy the single rule from Example 1. But for this to be true, it needs to contain $S(b)$ or it needs to contain $P(b, n), P(n, b)$ for some term n . For any such model M , we find a *GroundTermMapping* h , which is the identity on all terms except that it maps $f_z^{1,1}(b)$ to n . Thereby, we can pick either the first or second element from $\mathcal{U}$ (call this U) and witness that h is a homomorphism from U to M .

A proof on paper might just say that for a given M , the corresponding U and the homomorphism can be constructed inductively along the chase. While this is an essential ingredient, this does not conclude the proof. The final U only results from taking the union of the constructed sets and the homomorphisms also still need to be combined into a single function. And even then, we still need to prove that the combined U indeed occurs in the result of the chase tree, which is to specify the infinite branch that the constructed fact sets belong to. And of course, we also need to prove that the combined homomorphism agrees with the inductively constructed ones on their corresponding fact sets. All of this is not hard to believe and therefore likely omitted on paper. Still the formal arguments are not trivial and yet again involve some coinductive principles.

Remark 4. To be able to construct an infinite branch in a tree from an inductive construction, we make use of the *generate_branch* function, which is part of our *FiniteDegreeTree* framework. This function is corecursive and accepts a “generator” function as input that always creates a new tree element from a previous one. Given that this generator adheres to certain properties, we can directly conclude that the *PossiblyInfiniteList* produced by the *generate_branch* function is indeed a branch in the *FiniteDegreeTree*. We also lift this to *TreeDerivations* in *generate_subderivation*.

The generator function (*hom_step*) takes an *InductiveHomomorphismResult* as input and output, which consists of a node in the chase tree and a *GroundTermMapping* that is a homomorphism from the node into the target model M . We show the type signatures in Listing 2. The application of *generate_subderivation* can be found in the proof

```
abbrev InductiveHomomorphismResult (ct : ChaseTree obs kb) (m : FactSet sig) :=
  { pair : ct.NodeWithAddress × (GroundTermMapping sig) //
    pair.snd.isHomomorphism pair.fst.node.facts m }

noncomputable def hom_step (m_is_model : m.modelsKb kb)
  (prev_res : InductiveHomomorphismResult ct m) : Option (InductiveHomomorphismResult ct m)
```

Listing 2: Signature of Inductive Homomorphism Definition for the Universality Proof.

of `chaseTreeResultIsUniversal`. After setting up an initial `InductiveHomomorphismResult`, we essentially call `generate_subderivation` passing `(hom_step ct m m_is_model)` as the generator to obtain the full branch that yields U as its result. Using `generate_subderivation_mem_branches`, we get the proof that the generated branch indeed occurs in the branches of the chase tree. We also define a combined version of all homomorphisms (called `global_h`) and show all necessary properties, which involves quite a few lines of code, but this is conceptually less interesting. Thus, we have now successfully shown in Lean that every `ChaseTree` for any `ObsolescenceCondition` indeed procudes a universal model set, which, as mentioned earlier, also yields the special case: when restricting to deterministic rules, the result of every chase branch for any `ObsolescenceCondition` is a universal model.

6 Alternative Matches and Cores

In this section, we discuss that, going beyond universality, if we can avoid *alternative matches* in a `ChaseBranch` on deterministic rules, then the result is a core. In fact, we have formalized all results from Section 3 in (Krötzsch 2020). We even generalize the presented results as we again show the results generically for any `ObsolescenceCondition` but more importantly, since we consider two different notions of cores. On infinite structures, different definitions of cores are possible. For clarity, we refer to the definition used in (Krötzsch 2020) as a “weak core”, which states that a `FactSet` is a *weak core* if every endomorphism is an embedding (strong and injective). A `FactSet` is a *strong core* if every endomorphism is an automorphism (strong, injective, and surjective). On finite structures the notions of weak and strong core coincide. The main goal of this section is to describe the formalization of the following result (which includes a corresponding result for weak cores).

```
theorem result_isStrongCore_of_noAltMatch
  {cb : ChaseBranch obs kb}
  (det : kb.isDeterministic) :
  ¬ cb.has_alt_match →
  cb.result.isStrongCore
```

Since homomorphisms are `GroundTermMappings` and therefore total functions, defined on all terms, our definitions of `strong` and `injective_for_domain_set` take an argument that specifies the relevant part of the domain. For example, for the definition of `strong`, one might intuitively state that the mapping of every fact not in a given fact set `fs` should again not be in `fs`. This demand is too

strong. Often, we have no information on how a homomorphism handles terms that are outside of `fs.terms` and there might very well be facts outside of `fs` that end up in `fs` under the homomorphism. As long as such a fact features at least one term outside of `fs.terms`, this is unproblematic. The requirement for strong homomorphisms shall only ensure that previously existing but unconnected terms do not get connected via a predicate by the homomorphism.

```
def strong (h : GroundTermMapping sig)
  (domain : Set (GroundTerm sig))
  (A B : FactSet sig) : Prop :=
  ∀ (e : Fact sig),
  (∀ t, t ∈ e.terms → t ∈ domain) →
  ¬ e ∈ A → ¬ (h.applyFact e) ∈ B
```

The idea for alternative matches is to witness redundancies in a `ChaseBranch`. Intuitively, a term t that is newly introduced by a trigger is redundant if we can find a homomorphism from the result of the trigger into another part of the chase such that t does not occur in the image. The existence of an alternative match for a trigger witnesses that at least one newly introduced term is redundant in the context of a given fact set. We present the formal definition in Listing 3. We say that a `ChaseBranch` has an alternative match if some trigger in the branch has an alternative match in the context of the result of the branch. This definition does not yet make the assumption of deterministic rules. We only demand this on theorems where it is really necessary.

Example 6. Consider the database $D = \{P(a, b)\}$ and the singleton rule set R :

$$P(x, y) \rightarrow \exists z. P(y, z) \wedge P(y, x)$$

The rule applied to $P(a, b)$ produces $P(b, f_z^{1,1}(b))$, $P(b, a)$ and thereby a redundancy in itself ($f_z^{1,1}(b) \mapsto a$). Indeed, this mapping witnesses an alternative match. Still, $\{P(a, b), P(b, a)\}$ is a universal model and also a core. Only we cannot obtain it with the restricted or Skolem chase.

For the main goal, we first prove that the chase result is a weak core in the absence of alternative matches in `result_isWeakCore_of_noAltMatch`. We follow along the original proof (Krötzsch 2020, Theorem 2). We assume for a contradiction that some endomorphism on the `ChaseBranch` result is not strong or not injective. However, since the branch does not have alternative matches, we can inductively modify this endomorphism to be the identity on each step in `ChaseBranch`. At the same time, we can show alongside the same induction that this construction preserves that the endomorphism is not strong or not injective, respectively, which eventually leads to a contradiction.

```
def isAlternativeMatch (h_alt : GroundTermMapping sig) (trg : PreTrigger sig)
  (disj_index : Fin trg.mapped.head.length) (fs : FactSet sig) : Prop :=
  (h_alt.isHomomorphism trg.mapped.head[disj_index.val].toSet fs) ∧
  (∀ t, t ∈ trg.rule.frontier.map trg.subs -> h_alt t = t) ∧
  (∃ t, (t ∈ trg.fresh.terms.for_head_disjunct disj_index.val ...) ∧
    (¬ t ∈ (trg.fresh.terms.for_head_disjunct disj_index.val ...).map h_alt))
```

Listing 3: Definition of Alternative Match.

Remark 5. For a key lemma, we again utilize a corecursive construction similar to Section 5 although a bit simpler. Whenever there is a homomorphism h from a fact set F in a *ChaseBranch* to its result, we need to be able to extend h into an endomorphism on the result that agrees with the original homomorphisms on F . We formalize this in *hom_for_node_extendable_to_result* with an inductive construction of a homomorphism (without constructing a tree branch now). We use the *generate* function of *PossiblyInfiniteLists* to obtain an infinite list of homomorphisms that we combine into one.

To prove *result_isStrongCore_of_noAltMatch*, we still need to show that every endomorphism on the chase result U is surjective. But this is close to the alternative match definition. Suppose for a contradiction that some endomorphism h on U is not surjective. Then, there is a first term t that is never mapped to itself by h^i for any $i \geq 1$ (indicating that h is repeated i times). The term t originates from some trigger λ . Each term s that was introduced earlier is therefore mapped to itself by h^j for some $j \geq 1$. By picking a large enough common multiple k of these j , (1) the mapping h^k is still an endomorphism on U and therefore also a homomorphism from the result of λ into U , (2) all frontier terms (which occurred before λ) are mapped to themselves, and (3) t does not occur in the image of the fresh terms of the trigger after applying h^k . This shows all three conditions from the alternative match definition and therefore yields the desired contradiction.

7 A Framework for MFA-like Conditions

Model-Faithful Acyclicity (MFA) (Cuenca Grau et al. 2013), Disjunctive MFA (DMFA) (Gerlach and Carral 2023b), and Restricted MFA (RMFA) (Carral, Dragoste, and Krötzsch 2017) are sufficient conditions for chase termination (aka. acyclicity notions). So if a rule set fulfills the condition, then every chase on it terminates. Termination formally means that the underlying *PossiblyInfiniteList* is finite, in case of a *ChaseBranch*, or that all branches are finite, in case of a *ChaseTree*. A *KnowledgeBase* terminates if all of its *ChaseTrees* terminate and a *RuleSet* terminates if all *KnowledgeBases* featuring this rule set terminate. We generalize MFA, DMFA, and RMFA into a common framework for any *ObsolescenceCondition* and (as per our usual definition) we allow rules to contain constants, which is not considered in the original definitions of DMFA and RMFA. For constants, not many changes are necessary but the challenge is to make the *right changes* in the *right places*.

We highlight these in **bold**. Thanks to Lean, we can easily spot how the changes affect our proofs. We should also mention that the original definitions for DMFA and RMFA are optimized to work only with chase trees that prioritize the application of Datalog rules. We drop this optimization in our formalization to support arbitrary chase trees.

The central idea of all 3 acyclicity notions is to compute a chase-like procedure starting on a special database based on a given rule set rs . We call the resulting fact set *mfaSet* and show that rs .*terminates* if *mfaSet* is finite (see *terminates_of_mfaSet_finite*). The idea for the proof is that every trigger used in any *ChaseBranch* has a corresponding trigger application in the *mfaSet*.

The *mfaSet* is based on the *mfaKb*, which can be obtained solely from rs . The rule set in *mfaKb* is exactly rs . The database of *mfaKb* (aka. *criticalInstance*) contains facts for each predicate in rs and all possible combinations of **constants from rs** and a special constant $\star$.

The *parallelDeterminizedChase* is a chase-like procedure used to compute *mfaSet* from *mfaKb*. It treats disjunctions as conjunctions and applies all active triggers at once. It thereby yields an *InfiniteList* of *FactSets*. Indeed, this list is always infinite, since when no triggers are active anymore, then the last fact set is just repeated forever. The goal is that *mfaSet* is a gross overestimation of all fact sets that may occur in any *ChaseBranch* of *mfaKb*. At the same time, *mfaKb* generalizes all *KnowledgeBases* featuring rs . So the *mfaSet* really overapproximates the result of any *ChaseBranch* of any such knowledge base. The proof of *terminates_of_mfaSet_finite* comes down to showing that every *ChaseBranch* for every *KnowledgeBase* featuring rs can be embedded into *mfaSet*. This is done by mapping all constants to $\star$ **except the ones occurring in the rule set, which are just mapped to themselves**. We prove this via induction using *mem_rec*.

Remark 6. The realization of Remark 2, namely that we can generalize Skolem and restricted obsolescence into a common *ObsolescenceCondition*, resonates well with DMFA and RMFA and immediately allows us to generalize these notions accordingly. Not only that but also MFA can be expressed in a similar fashion.

The difference between MFA, DMFA, and RMFA comes down to which triggers we consider to be active in the *parallelDeterminizedChase*. We use an *MfaObsolescenceCondition* (MOC) to express this, which is an alias for *LaxObsolescenceCondition*. These enforce fewer conditions than our usual *ObsolescenceCondition* (OC). We then make

our main result `terminates_of_mfaSet_finite` generic over an MOC m and an OC o and demand that m `blocks_obs` o . The idea behind `blocks_obs` is that whenever a trigger λ from the `parallelDeterminizedChase` is obsolete according to m , then all *corresponding* triggers λ' from any real `ChaseBranch` on any knowledge base are obsolete according to o . Corresponding means that replacing all constants in λ' by $\star$ **except the ones from the rule set** yields λ . Or in other words: λ' is simulated by λ in the `parallelDeterminizedChase`.

MFA uses `DeterministicSkolemObsolescence` (DSO) for its MOC, which marks a trigger as obsolete if all of its results are already present. For DMFA and RMFA (and everything in between), we use `BlockingObsolescence` (BO), which depends on a given OC. For both DSO and BO, we need to show that they have the necessary `blocks_obs` property. DSO has the `blocks_obs` property for every OC and BO has this property exactly for the OC that it receives as an argument. So DMFA and RMFA correspond exactly to BO with `SkolemObsolescence` and `RestrictedObsolescence` respectively, but any other OC could be used just as well.

The definition of `BlockingObsolescence` is extremely involved and beyond what we can cover in this writeup. To at least give an intuition, the idea for both DMFA and RMFA is to limit the number of triggers in the computation of the `mfaSet` by checking which triggers λ are necessarily obsolete as soon as they are loaded. The basis for this check is a backtracking of triggers that need to be applied in order for λ to become loaded. This roughly works as follows. When λ is loaded, its instantiated body needs to be present. But not only that. Each Skolem term stems from a unique rule head disjunct. Consequently a suitable trigger must have been applied before. In this way, one can backtrack a set of facts F that are necessarily involved in the target trigger λ becoming loaded. If λ is obsolete with respect to this F , then we call λ “blocked” since it can under no circumstances be applied in a chase. This is exactly when `BlockingObsolescence` marks λ as obsolete.⁸ Checking whether λ is obsolete with respect to F uses the OC that BO received as an argument. The difference in the OC is also precisely the what sets DMFA and RMFA apart.

While the backtracking idea is still intuitive, in truth this whole procedure involves introducing fresh constants for terms introduced in the backtracking and renaming constants in triggers apart. Fresh constants are necessary because our Skolem terms do not indicate the mapping of non-frontier body variables. Picking fresh constants introduces a problem similar to Remark 1 and in this case indeed involves passing around a list of constants that have already been used. Renaming constants in triggers apart is necessary since the BO check considers triggers from the `parallelDeterminizedChase`, which would only feature $\star$ **and the rule set constants**

in all positions, but the check still needs to generalize all possible triggers that end up mapping to λ . Renaming again leads to the introduction of fresh constants. Specifically the handling of fresh constants is what makes the machinery around `BlockingObsolescence` involved. Defining `BlockingObsolescence` and proving the `blocks_obs` property involves a couple of thousand lines of code. For more explanations and examples for some of the subtleties, we refer the interested reader mainly to the original papers (Gerlach and Carral 2023b; Carral, Dragoste, and Krötzsch 2017).

There is still one point that we have not discussed. How do we know if `mfaSet` is finite? The acyclicity notions should be decidable after all and not run into an infinite computation. All notions handle this in the same way: the computation of the `parallelDeterminizedChase` stops as soon as we witness a *cyclic term*, e.g. $f(g(f(c), d))$ but not $g(f(c), f(d))$. Since the function symbols are Skolem functions, a cyclic term hints at a rule being applied on its own result. This does not necessarily endanger termination but it could. In this case, we cannot tell if `mfaSet` is finite. If the procedure terminates on its own without seeing a cyclic term, then `mfaSet` is obviously finite. Since there are only finitely many non-cyclic terms, the procedure always halts. This is how we define `isMfa` (again generically over an MOC). The main correctness result is `terminates_of_isMfa`, which we also instantiate specifically for DSO and BO afterwards. Convincingly presenting this argument to the Lean kernel involves theorems around finiteness of sets containing different syntactic entities. For example, we require a result showing that a `FactSet` must be finite if it only features finitely many different predicates and only finitely many different terms. Also, we show that a functional term of a large enough depth is necessarily cyclic if there are only finitely many different function symbols (see `cyclic_of_depth_too_big`).

Example 7. We realized in Example 3 that there is an infinite `ChaseBranch` for `SkolemObsolescence`. Consequently, the rule set should not be (D)MFA. Indeed, the `mfaSet` is infinite and contains $P(\star, \star), S(\star), P(f_z^{1,1}(\star), \star), P(\star, f_z^{1,1}(\star)), P(f_z^{1,1}(f_z^{1,1}(\star)), f_z^{1,1}(\star)), P(f_z^{1,1}(\star), f_z^{1,1}(f_z^{1,1}(\star)))$. Thus, we witness a cyclic term $f_z^{1,1}(f_z^{1,1}(\star))$ and conclude that the rule set is not (D)MFA. However, the rule set is still RMFA.

8 Concluding Remarks

We provide a formalization of disjunctive existential rules and a generalized chase procedure in Lean also modelling possibly infinite trees as a by-product. We demonstrate the usability by proving and extending known results from existential rule research: (1) the chase result is a universal model set, (2) a chase without alternative matches yields a universal model that is a *strong* core, and (3) we prove correctness of a novel framework built around MFA-like acyclicity notions with full support for constants in rules.

The possibilities for extending this library are just as broad as the realm of existential rule research. On our roadmap, addressing issue (B) from the introduction is one

⁸The Datalog optimization in the original definitions of DMFA and RMFA computes the Datalog-closure on the backtracked facts.

of our main upcoming goals. The considerations around DMFA and RMFA feel like a mere precursor to this as non-termination conditions such as DMFC (Gerlach and Carral 2023b) and RPC (Gerlach and Carral 2023a) only get more complicated. Furthermore, formalizing the core chase is already work in progress. Specifically, we aim to show that the core chase terminates if and only if a finite universal model exists (Deutsch, Nash, and Rummel 2008, Theorem 7). Related to cores, extending the formalization around alternative matches to cover more parts of the introducing paper is also well within our scope (Krötzsch 2020). Long term, we hope to add the result related to issue (A) and decidability results for chase termination on existential rule fragments (Leclère et al. 2019; Gogacz, Marcinkowski, and Pieris 2023; Gerlach et al. 2025), which require more foundational work on computability theory.

Beyond theoretical results, it would be interesting to have a formally verified reference implementation of the chase or the MFA-like conditions directly in Lean. Right now, we only describe the chase structures mathematically without specific procedures for how they can be computed. However, this would be possible with a significant but manageable amount of work. Lean is in principle able to compile computable definitions directly into executable code. The biggest issue here likely will be that the chase is not guaranteed to terminate. Lean usually expects recursion to be well-founded. If this is not obvious, Lean also expects a termination proof. There are ways to define possibly non-terminating functions though, but they come at a cost. Definitions can be marked as `partial`, which still allows to execute them but we essentially lose the ability to prove properties about them. What seems more interesting is the rather recent `partial_fixpoint` feature. It allows to mark certain functions, e.g. tail-recursive ones, as possibly non-terminating. Compared to the `partial` keyword, we retain the possibility of proving properties about the function.

On a meta level, we hope our humble contribution to be a spark that ignites interest within the community to formalize more definitions and results from the field of knowledge representation and reasoning. We believe that a formal library of definitions and proofs not only ensures reproducibility but also encourages development of novel, fully verified theorems. And, as we demonstrate, inspires new generalizations. We realized in our own formalization efforts that the process can be tedious and time-consuming at times but pays off and even becomes enjoyable in the long run.

Acknowledgments

I want to thank my colleague Dr. Stephan Mennicke for proofreading and providing feedback on a draft of this paper.

This work was partly supported by DFG (German Research Foundation) in project 389792660 (TRR 248, Center for Perspicuous Systems) and in the CeTI Cluster of Excellence; by the Bundesministerium für Forschung, Technologie und Raumfahrt (BMFTR) in the Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI); and by BMFTR and DAAD (German Academic Exchange Service) in project 57616814 (SECAI, School of Embedded and Composite AI).

AI Declaration

The authors have employed Generative AI tools only for proofreading (spellchecking and improvement suggestions).

References

- Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison-Wesley.
- Baanan, A.; Ballard, M. R.; Commelin, J.; Chen, B. G.-g.; Rothgang, M.; and Testa, D. 2026. Growing mathlib: maintenance of a large scale mathematical library. In de Paiva, V., and Koepke, P., eds., *Intelligent Computer Mathematics*, 51–70. Cham: Springer Nature Switzerland.
- Baget, J.; Leclère, M.; Mugnier, M.; and Salvat, E. 2011. On rules with existential variables: Walking the decidability line. *Artif. Intell.* 175(9-10):1620–1654.
- Baget, J.; Garreau, F.; Mugnier, M.; and Rocher, S. 2014. Extending acyclicity notions for existential rules. In Schaub, T.; Friedrich, G.; and O’Sullivan, B., eds., *ECAI 2014 - 21st European Conference on Artificial Intelligence*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 39–44. IOS Press.
- Barrett, C.; Chaudhuri, S.; Montesi, F.; Grundy, J.; Kohli, P.; de Moura, L.; Rademaker, A.; and Yingchareonthawornchai, S. 2026. Cslib: The lean computer science library.
- Beeri, C., and Vardi, M. Y. 1984. A proof procedure for data dependencies. *J. ACM* 31(4):718–741.
- Benzaken, V.; Contejean, E.; and Dumbrava, S. 2017. Certifying standard and stratified datalog inference engines in ssreflect. In Ayala-Rincón, M., and Muñoz, C. A., eds., *Interactive Theorem Proving - 8th International Conference, ITP*, volume 10499 of *Lecture Notes in Computer Science*, 171–188. Springer.
- Bonifati, A.; Dumbrava, S.; and Arias, E. J. G. 2018. Certified graph view maintenance with regular datalog. *Theory Pract. Log. Program.* 18(3-4):372–389.
- Bourhis, P.; Manna, M.; Morak, M.; and Pieris, A. 2016. Guarded-based disjunctive tuple-generating dependencies. *ACM Trans. Database Syst.* 41(4):27:1–27:45.
- Carral, D.; Larroque, L.; Mugnier, M.-L.; and Thomazo, M. 2022. Normalisations of Existential Rules: Not so Innocuous! In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, 102–111.
- Carral, D.; Gerlach, L.; Larroque, L.; and Thomazo, M. 2025. Restricted chase termination: You want more than fairness. *Proc. ACM Manag. Data* 3(2 (PODS)).
- Carral, D.; Dragoste, I.; and Krötzsch, M. 2017. Restricted chase (non)termination for existential rules with disjunctions. In Sierra, C., ed., *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017*, 922–928. ijcai.org.
- Cuenca Grau, B.; Horrocks, I.; Krötzsch, M.; Kupke, C.; Magka, D.; Motik, B.; and Wang, Z. 2013. Acyclicity notions for existential rules and their application to query answering in ontologies. *J. Artif. Intell. Res.* 47:741–808.

- de Moura, L., and Ullrich, S. 2021. The lean 4 theorem prover and programming language. In Platzer, A., and Sutcliffe, G., eds., *CADE 28 - 28th International Conference on Automated Deduction*, volume 12699 of *Lecture Notes in Computer Science*, 625–635. Springer.
- Deutsch, A.; Nash, A.; and Rammel, J. B. 2008. The chase revisited. In Lenzerini, M., and Lembo, D., eds., *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008*, 149–158. ACM.
- Dumbrava, S. 2016. *Formalisation en Coq de Bases de Données Relationnelles et Déductives -et Mécanisation de Datalog. (A Coq Formalization of Relational and Deductive Databases -and a Mechanizations of Datalog)*. Ph.D. Dissertation, University of Paris-Saclay, France.
- Fagin, R.; Kolaitis, P. G.; Miller, R. J.; and Popa, L. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336(1):89–124.
- Gerlach, L., and Carral, D. 2023a. Do repeat yourself: Understanding sufficient conditions for restricted chase non-termination. In Marquis, P.; Son, T. C.; and Kern-Isberner, G., eds., *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023*, 301–310.
- Gerlach, L., and Carral, D. 2023b. General acyclicity and cyclicity notions for the disjunctive skolem chase. In Williams, B.; Chen, Y.; and Neville, J., eds., *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023*, 6372–6379. AAAI Press.
- Gerlach, L.; Larroque, L.; Marcinkowski, J.; and Ostropolski-Nalewaja, P. 2025. About the Multi-Head Linear Restricted Chase Termination. In *Proceedings of the 22nd International Conference on Principles of Knowledge Representation and Reasoning*, 346–355.
- Gogacz, T., and Marcinkowski, J. 2014. All-instances termination of chase is undecidable. In Esparza, J.; Fraigniaud, P.; Husfeldt, T.; and Koutsoupias, E., eds., *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014*, volume 8573 of *Lecture Notes in Computer Science*, 293–304. Springer.
- Gogacz, T.; Marcinkowski, J.; and Pieris, A. 2023. Uniform restricted chase termination. *SIAM J. Comput.* 52(3):641–683.
- Grahne, G., and Onet, A. 2018. Anatomy of the chase. *Fundam. Informaticae* 157(3):221–270.
- Hanisch, P., and Krötzsch, M. 2024. Chase termination beyond polynomial time. *Proc. ACM Manag. Data* 2(2):93.
- Ivliev, A.; Gerlach, L.; Meusel, S.; Steinberg, J.; and Krötzsch, M. 2024. Nemo: Your friendly and versatile rule reasoning toolkit. In Marquis, P.; Ortiz, M.; and Pagnucco, M., eds., *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024*.
- Jordan, H.; Scholz, B.; and Subotic, P. 2016. Soufflé: On synthesis of program analyzers. In Chaudhuri, S., and Farzan, A., eds., *Computer Aided Verification - 28th International Conference, CAV 2016*, volume 9780 of *Lecture Notes in Computer Science*, 422–430. Springer.
- Karimi, A.; Zhang, H.; and You, J. 2021. Restricted chase termination for existential rules: A hierarchical approach and experimentation. *Theory Pract. Log. Program.* 21(1):4–50.
- Krötzsch, M., and Rudolph, S. 2011. Extending decidable existential rules by joining acyclicity and guardedness. In Walsh, T., ed., *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 963–968. IJCAI/AAAI.
- Krötzsch, M. 2020. Computing cores for existential rules with the standard chase and ASP. In Calvanese, D.; Erdem, E.; and Thielscher, M., eds., *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020*, 603–613.
- Leclère, M.; Mugnier, M.-L.; Thomazo, M.; and Ulliana, F. 2019. A Single Approach to Decide Chase Termination on Linear Existential Rules. In Barcelo, P., and Calautti, M., eds., *22nd International Conference on Database Theory (ICDT 2019)*, volume 127 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 18:1–18:19. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- Marnette, B. 2009. Generalized schema-mappings: from termination to tractability. In Paredaens, J., and Su, J., eds., *Proceedings of the Twenty-Eighth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2009*, 13–22. ACM.
- Nenov, Y.; Piro, R.; Motik, B.; Horrocks, I.; Wu, Z.; and Banerjee, J. 2015. Rdfbox: A highly-scalable RDF store. In Arenas, M.; Corcho, Ó.; Simperl, E.; Strohmaier, M.; d’Aquin, M.; Srinivas, K.; Groth, P.; Dumontier, M.; Heflin, J.; Thirunarayan, K.; and Staab, S., eds., *The Semantic Web - ISWC 2015 - 14th International Semantic Web Conference*, volume 9367 of *Lecture Notes in Computer Science*, 3–20. Springer.
- Schlichtkrull, A.; Rydhof Hansen, R.; and Nielson, F. 2024. Isabelle-verified correctness of datalog programs for program analysis. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing (SAC 2024)*, 1731–1734. ACM.
- Tantow, J.; Gerlach, L.; Mennicke, S.; and Krötzsch, M. 2025. Verifying datalog reasoning with Lean. In Forster, Y., and Keller, C., eds., *Proceedings of the 16th International Conference on Interactive Theorem Proving (ITP’25)*, volume 352 of *LIPIcs*. Dagstuhl Publishing.
- Urbani, J.; Jacobs, C. J. H.; and Krötzsch, M. 2016. Column-oriented datalog materialization for large knowledge graphs. In Schuurmans, D., and Wellman, M. P., eds., *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 258–264. AAAI Press.
- Whitehead, N. 2007. A certified distributed security logic for authorizing code. In Altenkirch, T., and McBride, C., eds., *Types for Proofs and Programs*, 253–268. Springer.