LEAN-aided verification of datalog reasoning results

Author: Johannes Tantow
born 02.02.1999 in Greifswald
Matrikelnummer: 4680411

Supervisor: Prof. Dr. Markus Krötzsch

A thesis presented for the degree of
Diplom-Informatiker

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science
Institute of Theoretical Computer Science
Knowledge-Based Systems Group
01.05.2024

Contents

Acknowledgements

I want to thank my advisors Lukas Gerlach and Dr. Stephan Mennicke for introducing Lean and the problem to me and for their helpful comments during the creation of this thesis. Lukas also improved my previous graph model in chapter 5 by adding the hash map implementation.

I want to thank Prof. Markus Krötzsch for the formal supervision and his helpful comments during the writing. He proposed amongst other things example 20.

I want to thank Dr. Stefan Borgwardt for being the second reviewer of my thesis.

I want to thank Samuel Krug for proofreading my thesis and finding aspects that were not understandable if one did not know Lean.

Lean's zulipchat[1] was very helpful during the development of this thesis. It contains many useful information and its members are quick to answer questions.

Finally, I want to thank my family and friends for always supporting me during my studies.

[1]https://leanprover.zulipchat.com/

Abstract

Datalog is a logic programming language widely used in database and knowledge representation use cases. Datalog engines compute the results and are optimized to handle millions of facts. Unfortunately, formally verifying these datalog engines is out of reach due to the complexity of the implementation of these optimizations. Modern datalog engines offer however proofs why an element was derived.

We propose a checker for these proofs. The checker is implemented in Lean and formally proven to be correct against our formalization of datalog. Proofs can be given in two formats: trees and directed graphs. The input can be given as JSON files. Proofs only explain why an element is in the solution but do not tell us that a derivation is complete. Therefore we implement a certified model checker to check the completeness of a solution. We use the checker to validate the results of the datalog engine Nemo and see that validating datalog proofs is fast in most use cases. Proof graphs appear better in practice as they can reuse atoms in the proofs and are faster prepared.

Chapter 1

Introduction

One of the most important challenges in software engineering is writing bug-free software. Bugs in software may endanger the well-being of humans and property.

A recent example of the consequences of buggy software was the Horizon software used by the British Royal Mail [14]. It was supposed to be an accounting software used in the accounting of the individual post offices. Horizon wrongly calculated the balance which led to the individual subpostmasters being accused of theft or fraud. Over 900 people were sentenced to the repayment of alleged damages or prison sentences. These people still suffer financial, social or health consequences despite being innocent and multiple persons committed suicide.

The Royal Mail trusted their program too much to consider that it contained errors. But what methods exist to increase the confidence in a program? Firstly, we can prove the correctness of a program, which is done in the field of formal verification. These proofs are complicated and often very large so they need computer assistance. Due to time constraints or unfamiliarity with the methods this is only very rarely done in use cases that require specific security guarantees. An example is the verified C compiler CompCert[26].

In practice, software developers mostly employ a test-driven approach to software correctness. They design specific use cases and specify what the program is supposed to do in this case. Designing such cases is difficult and cannot cover every scenario so we can only see that some scenarios are bug-free.

An interesting development in algorithms research are certifying algorithms [28]. These algorithms do not only return a solution but also a certificate as an explanation why the returned value is the correct answer. In the case of SAT-solving, they would not only state that a formula is satisfiable but also return a model. The user can then verify on their own that the result is correct without inspecting the code's inner workings. This can also be done by programs that are called checkers. These programs are usually smaller and simpler compared to the program that calculates the solution so that they can be formally verified.

Our interest concerns computing the results of datalog programs. Datalog is a logic programming and query language that allows to write rules in the following way

$$\begin{aligned} &T(?x, ?y) \leftarrow E(?x, ?y). \\ &T(?x, ?z) \leftarrow E(?x, ?y), T(?y, ?z). \end{aligned} \tag{1.1}$$

The rules can be interpreted as formulas from first-order logic that are universally quantified and use implication. There are multiple equivalent semantics of datalog that deal with the question of what follows from a datalog program and a given set of facts also known as a database.

This task can be solved in polynomial time which allows using datalog in multiple interesting applications from data analysis[31] to data integration[19].

Modern datalog engines like Nemo[22] or Soufflé[24] can work with millions of database elements and use many optimization techniques to achieve this. Unfortunately, the correctness may suffer from this and formally verifying an engine is out of reach. Only a limited amount of optimization techniques has been verified so far[7]. They do however offer proof trees as an explanation why a certain element was derived so that we may view them as an instance of certifying algorithms. In this style, we want to implement a checker that takes these certificates and tells us whether a datalog reasoning result is correct.

We will formally verify the checker using the proof assistant Lean[17] to improve the confidence in the correctness of our checker. A proof assistant allows users to define functions and objects and prove results about the objects in machine-readable style. Therefore the proof assistant can verify that the proof is indeed correct or if not raise errors.

After introducing the basics about datalog, certifying algorithms and Lean in chapter 2, we will formalize datalog in Lean in chapter 3. After that, we provide two methods to verify a datalog reasoning result: a first method based on proof trees in chapter 4 and another method based on graphs we see as more promising in practice in chapter 5. These methods however can only tell us that all atoms present in the result are correctly derived, i.e. soundness. They do not tell us whether more facts could be derived. A method to solve this question is presented in chapter 6.

Finally, we show in chapter 7 a practical evaluation on multiple programs and databases. The code and data can be found in the corresponding Github repository.

1.1 Related Work

The approach to using certified algorithms is well documented in the literature. Multiple examples from graph theory, linear algebra or geometry are mentioned in [28]. Practical implementations can be found in the Library of Efficient Data Types and Algorithms(LEDA)[29]. These checkers are often verified in a proof assistant[2].

An alternative angle for this design principle is the de-Bruijn criterion [6]. There we consider proofs verified by a program. Since these programs may contain bugs and thus create faulty proofs, the program should offer a way to independently check the proofs instead of just returning the answer. This is done for the problem of checking proofs in propositional logic[20] or first-order logic[5] or for checking the termination of higher-order rewriting systems[32].

There is currently no formalization of datalog in Lean. Parts of the syntax and semantics of datalog are verified in Coq[8] with a focus on the fixed point semantics in order to implement the datalog reasoner directly in the proof assistant[8, 9]. These implementations are mostly slower compared to engines written in more typical programming languages and have fewer features.

Chapter 2

Preliminaries

In this section, we introduce concepts that we are going to use in future sections. We start by defining datalog, which we will formalize later. After that, we review the concept of certifying algorithms and finally, we introduce Lean.

2.1 Datalog

Datalog is a logic programming language and can be syntactically considered as a subset of Prolog. An introduction to datalog and other query languages can be found in [1], which we recall next. We assume the reader to be familar with first-order logic as introduced in e.g. [21].

In order to write datalog rules, we require the existence of three at countable, possibly infinite sets V, C and R, where V represents variables, C represents constants and R represents relation symbols. Every relation symbol r in R has an arity $ar(r) \in \mathbb{N}$. In this work we assume that zero is a natural number. We use the following conventions in this work: For R we use strings that start with a capital letter, for V strings that start with a question mark and for C any other non-empty string.

Example 1. *We consider have the variables $V = \{?x, ?y, ?z\}$ and constants $C = \{1, ..., 7\}$. The predicate symbols are the binary predicates (i.e. they have an arity of two) S and E and the nullary (i.e. arity zero) predicate Q.*

In the language of first-order logic as in [21], C and P build the signature of the language we are going to construct. We note that this fragment of first-order logic does not use function terms.

The following definitions are adapted from logic for the signature above.

A *term* is either a variable or a constant. In order to differentiate meaningfully between them, we require that $C \cap V = \emptyset$ so that no symbol is both a variable and a constant. As long as both C and V are finite, we will only have a finite number of terms. This is in contrast to general first-order logic or logic programming where function symbols can lead to an infinite amount of terms even if the signature is finite.

Example 2. *Examples of terms in our language from example 1 are $?z$, 2, 5 or $?x$.*

A counterexample would be word or A as those elements do not occur in either C or V.

An *atom* is an expression of the form $A(t_1, ..., t_n)$ for $n \in \mathbb{N}$ where A is a relation symbol ($A \in P$) and $t_1, .., t_n$ are terms. Additionally, we require that n is the arity of A, i.e. $ar(A) = n$.

Example 3. *Examples for atoms using the sets from example 1 are here: $Q(), S(?x, ?y), E(2, 3)$ or $E(?x, 7)$. In every atom, the number of terms matches the arity of the predicate symbol. Note that it is allowed to mix variables and constants in the terms of an atom.*

A first counterexample would be $Q(a)$ because the arity of the symbol Q does not match the number of terms. Another counterexample would be $S(E(2, 3), 4)$ because $E(2, 3)$ is not a term.

In the example, some atoms used only constants in their terms, while others included variables. We call an atom a *ground atom* if it is variable-free. The variables used in an atom $A(t_1 \dots t_n)$ are formally the following set:

$$Vars(A(t_1, ..., t_n)) = \{t_i \mid t_i \in V\}$$

.

Example 4. *In the previous example $Q()$ and $E(2, 3)$ are ground atoms. The first does not have any terms at all, whereas the other only uses constants.*

The other atoms all contain at least the variable $?x$ and are thus no ground atoms.

A *rule* is an expression of the form $H \leftarrow B_1, ..., B_n$ for atoms H and B_i for $n \in \mathbb{N}$. We call H the *head* of the rule and $B_1, ..., B_n$ the *body* of the rule and define the functions $head(r) = H$ and $body(r) = \{B_i \mid i \in \{1, \dots n\}\}$. *Facts* are rules where n is zero.

We can generalize the $Vars$ function to rules and use this to define *ground rules.* For a rule $r = H \leftarrow B_1, ..., B_n$, we define $Vars(r)$ to be the union of the variables occuring in the head and the body:

$$Vars(r) = Vars(H) \cup \bigcup_{i \in \{1, ..., n\}} Vars(B_i)$$

and call r a ground rule if $Vars(r) = \emptyset$. This means that a rule is a ground rule if the head and every atom in the body is a ground atom.

Example 5. *Examples for rules are $E(1, 2) \leftarrow$, $Q() \leftarrow E(?x, ?y)$ or $T(?x, ?z) \leftarrow E(?x, ?y), T(?y, ?z)$.*

$E(1, 2) \leftarrow$ is both a fact and a ground rule, but not every fact is a ground rule as for example $E(?x, ?x) \leftarrow$

A *program* P is a finite set of rules.

Example 6. *We consider the program P with the following rules:*

$$\begin{aligned} &E(1,2) \leftarrow \\ &E(1,3) \leftarrow \\ &E(3,5) \leftarrow \\ &E(4,6) \leftarrow \\ &E(4,7) \leftarrow \end{aligned} \qquad \begin{aligned} Q() &\leftarrow \\ T(?x,?y) &\leftarrow E(?x,?y) \\ T(?x,?x) &\leftarrow \\ T(?x,?z) &\leftarrow T(?x,?y), T(?y,?z) \end{aligned}$$

P contains both facts and other rules.

With this, we have introduced all the needed elements for the syntax of datalog, but we have not yet considered the semantics of datalog. What is the program P supposed to express?

We have noted the similarities between datalog and first-order logic when defining the syntax of datalog. Semantically, we can view rules as elements of the universally quantified Horn-fragment of first-order logic. In the logic programming style the universal quantifiers are usually omitted and the implications are written from right to left. As a theory is a set of formulae we can view the program as a theory consisting of its rules.

Example 7. *The equivalent sentence for the fact $E(1,2) \leftarrow$ is*

$$E(1,2)$$

and for the rule $T(?x,?z) \leftarrow T(?x,?y), T(?y,?z)$ is

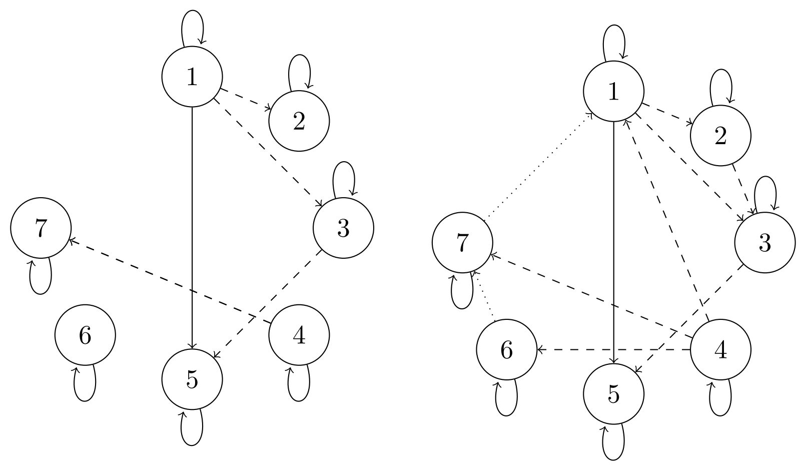
$$\forall ?x, ?y, ?z. T(?x,?y) \wedge T(?y,?z) \rightarrow T(?x,?z)$$

The semantics of a first-order theory or sentence are its models. Unfortunately, a program may have multiple models as the following example shows for the theory for P.

Example 8. *Any fact must be true in a model, but the implications allow us more freedom. As both E and T are binary predicates, we can represent them as a graph. We use dashed edges, when both E and T are present, solid edges for only T and dotted edges when only E is present.*

The first model expresses T as the reflexive transitive closure of the input E, whereas the second model expands the first model and adds more additional atoms to the previous model.

There are multiple models but we already see in the example there one model is a subset of the other. Such a model always exists in datalog and is called the least model. This model is of interest as the atoms that a true in this model are true in every model



(a) First model describing the reflexive-transitive closure

(b) Second model: Extension of (a)

Figure 2.1: Models for the program from example 6. The encoding is explained in example 8

We call a finite set of ground atoms an *interpretation*. An interpretation represents all true atoms. We want to define the model property from first-order logic for this definition and show then that there exists a least model according to the subset relation.

Recall from first-order logic that $\forall x.\phi(x)$ holds in a structure $\mathcal{A}$ if for any element $a \in \mathcal{A}$ we have that $\phi(a)$ holds. Therefore we could just create ground rules by replacing variables with all possible constants. Formally, this is defined using *groundings* or *instantiations* which are functions that map variables to constants. We can apply a grounding g to an atom by replacing every variable v in the terms by $g(v)$ and apply g to a rule by applying it to the head and every atom in the body. At the end of this, we have replaced every variable by a constant and have gained a ground rule. The ground program $\mathit{ground}(P)$ of a program P is the set of all ground rules that are the result of applying some grounding to a rule from P.

We call a ground rule r *true* in an interpretation I if whenever $\mathit{body}(r)$ is a subset of I, then also $\mathit{head}(r)$ is in I. We call an interpretation I a *model* of a program P if every rule of $\mathit{ground}(P)$ is true in I.

So now we have defined models and can define the least model as well. We still need to show that the least model exists for which the following lemma is helpful.

Lemma 1. *Let M_1, M_2 be two models of a program P. Then also $M_1 \cap M_2$ is a model of P.*

Therefore the intersection of all models is a model as well and due to the properties of the intersection it is least according to the subset relation.

In total, we call

$$\bigcap_{M \text{ is model of } P} M$$

the *least model* of P and refer to this characterization as the *model-theoretic semantics* of P.

Example 9. *In this example, the first model in of fig. 2.1 is actually the least model and the second model is just some other model.*

Therefore the semantic of this program is the reflexive-transitive closure of E if $Q()$ is given. If $Q()$ is not given then it is only the reflexive closure.

Additionally, we see that the rules for T encode general rules whereas the other rules only encode a specific input. We might want to reuse these rules for many different instances of E, but so far we also need to write a new program.

The example raises questions about the reusability of a program. Additionally, we talked in the introduction about database queries but never talked about databases.

We consider a *database* as a finite set of ground atoms similar to an interpretation. What is now the semantics of a program P and a database d? We simply add every element of d as a fact to P and reuse the previous semantics. We can also move all the facts that are ground rules in the program into the database and gain a more reusable program. An alternative model definition for a pair of P and d is therefore: An interpretation I is a model for P and d if every ground atom from d is in I and every rule from $ground(P)$ is true in I. Then lemma 1 holds again and we reuse the definition for the database and program case.

Both views are equivalent. We have shown how we can simulate a database by a program. We can simulate the program case with the program and database case by simply using an empty database.

We have now defined the semantics of datalog and shown a connection with databases. But this definition is not ideal. In order to find the semantics of a program we would need to check all interpretations and then intersect them all. This is computationally expensive. Is there a simpler way?

We can define a function that computes the least model in a more direct way. For this, we consider the case where only the program P is present.

Consider an interpretation I. We call a ground atom a an *immediate consequence* of I if there exists a ground rule r in $ground(P)$ with the head a and $body(r) \subseteq I$.

The immediate consequence operator T_P computes all immediate consequences of an interpretation.

$$T_P(I) = \{a \mid a \text{ is an immediate consequence of } I\}$$

A fixed point of a function f is an element k such that $f(k) = k$. The repeated application of T_P starting from $\emptyset$ yields a fixed point that is equal to the model-theoretic semantics of a program P.

Example 10. *We consider again the program P from example 6.*

Due to the fact $E(1,3) \leftarrow$, $E(1,3)$ is an immediate consequence of $\emptyset$. Applying T_P again, we have that $T(1,3)$ is an immediate consequence of $\{E(1,3)\}$ due to the rule $T(?x,?y) \leftarrow E(?x,?y)$ with the grounding

$$g(v) = \begin{cases} 1 & \textit{if } v = ?x \\ 3 & \textit{else} \end{cases}$$

Applying this grounding to every fact, we gain an interpretation I that contains $T(1,3)$ and $T(3,5)$. Then $T(1,5)$ is an immediate consequence of I due to the rule $T(?x,?z) \leftarrow T(?x,?y), T(?y,?z)$ and the grounding

$$g'(v) = \begin{cases} 1 & \textit{if } v = ?x \\ 3 & \textit{if} v = ?y \\ 5 & \textit{else} \end{cases}$$

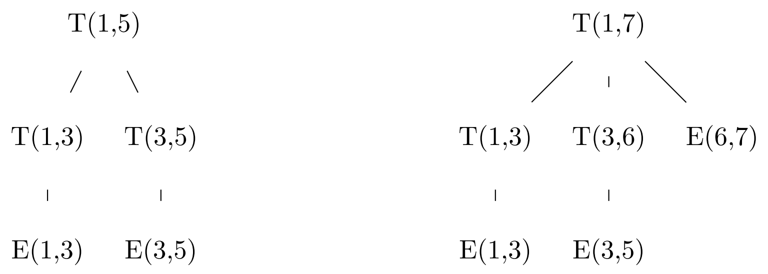
We can continue this process until we find the fixed point.

The least fixed-point of T_P is called the *fixed-point semantics* of P and is the basis for most implementations of datalog reasoners.

Our goal is to explain why an atom is in the datalog result. The third important semantics of datalog helps here.

A tree t of whose vertices are labeled with ground atoms is a valid proof tree for a ground atom a in a program P and database d if the following conditions hold:

1. The root of t is labelled by a
2. For every node n with the label a and its children l in t one of the following two conditions holds:
   (a) $a \leftarrow l_1, \ldots, l_n$ for $l_1, .., l_n$ being the labels of the elements in l is a ground rule from $ground(P)$, or
   (b) n is a leaf and a is in the database.



(a) A valid proof tree for $T(1,5)$ in the program in example 6

(b) An invalid tree for $T(1,7)$ in the program in example 6

Figure 2.2: Examples of valid and not valid proof trees

Example 11. *The tree in fig. 2.2a is valid. The leaves of this tree are facts and all other nodes represent ground rules from $ground(P)$ in the previous step.*

The tree in fig. 2.2b is not valid. $E(6,7)$ is neither a fact nor in the database. Additionally there is no rule in P where applying some grounding yields $T(1,7) \leftarrow T(1,3), T(3,6), E(6,7)$.

The proof-theoretic semantics of a program P and database d is the set of ground atoms that have a valid proof tree for P and d. Again this can be shown to be equal to the other definitions of semantics. We will formally prove the equality of the proof-theoretic and the model-theoretic semantics of datalog in this work.

In the rules of the program P we note a special rule of the form $T(?x, ?x) \leftarrow$. It is a fact, but not a ground rule and the only rule where this is the case. We call a rule $H \leftarrow B_1, \ldots, B_n$ *safe*, if every variable that occurs in the head also occurs in the body, i.e.

$$Vars(H) \subseteq \bigcup_{i \in \{1,..,n\}} Vars(B_i)$$

.

We call a program safe if every rule in the program is safe. Most works only consider safe programs as their ground program does not depend on the set of constants C which may not be given in practice.

Example 12. *We can transform P into the safe program P' by adding a new unary predicate N to the predicate symbols. P' is then the union of $P \setminus \{T(?x, ?x) \leftarrow\}$ and the following rules:*

$$\begin{aligned} N(?x) &\leftarrow E(?x, ?y) \\ N(?y) &\leftarrow E(?x, ?y) \\ T(?x, ?x) &\leftarrow N(?x) \end{aligned}$$

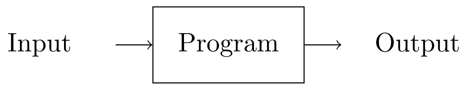


Figure 2.3: A convential algorithm

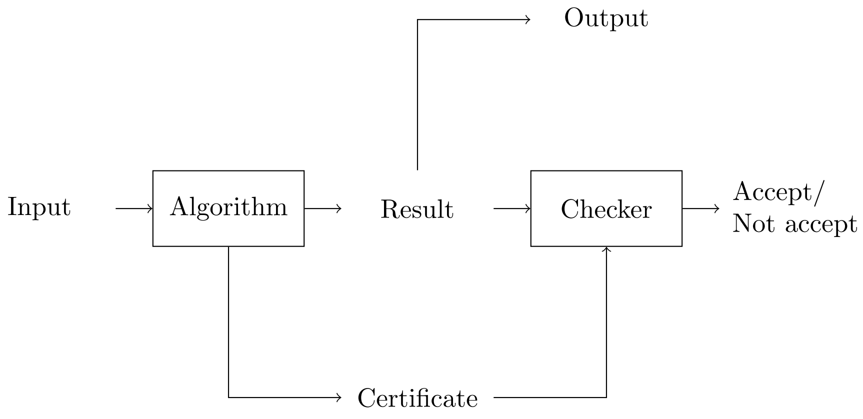


Figure 2.4: A certifying algorithm

The new predicate N (for nodes) represents any elements that occur in the E relation and is used in the body for the reflexive rule.

If some constants c are desired that do not occur in any E relation, one can also directly encode that into the database by adding $N(c)$.

What are the semantics of this newly created safe program P'? It turns out, that it is equal to the semantics for P for all original predicates, i.e. if we remove all ground atoms that use N we gain the same result. Therefore we state that any program can be transformed into an equivalent safe program.

2.2 Certifying algorithms

Most algorithms one encounters either in books like [16] or in real programs are comparable to fig. 2.3. We have an input, give this into an algorithm which is a black box for us and then we receive a result as an output. Without inspecting the implementation we have to trust that the results are correct. An alternative to this are *Certifying algorithms* depicted in fig. 2.4. They offer an explanation in addition to the result that can be checked independently by a checker or the user themselves. We follow the presentation in [28] which offers information far beyond this section as well.

Recall a definition of the class NP from complexity theory:

Definition 1 ([3])**.** *A language $L \subseteq \{0,1\}^\star$ is in NP if there exists a polynomial*

$p : \mathbb{N} \to \mathbb{N}$ and a polynomial-time Turing machine M such that forall $x \in \{0,1\}^\star$:

$$x \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|x|)} \textit{ s.t. } M \textit{ accepts } (x,u)$$

If x is in L, $u \in \{0,1\}^{p(|x|)}$ and (x,u) is accepted by M, then we call u a certificate for x

We can generalize this and require an algorithm not only to give us an output x but also a certificate u as a reason why x is the correct output. Then we can either check by ourselves that x is a correct solution according to u or use another program that checks this. As verifying a solution is never harder than computing it, the checker is usually simpler so that we either see directly that the checker is correct or we can formally verify it.

The formal framework of [28] defines certifying algorithms in the following way. We consider the set X of input values and the set Y of output values of a function f. A predicate $\phi : X \to \{true, false\}$ states a precondition for the inputs and another predicate the postcondition $\psi : X \times Y \to \{true, false\}$, ϕ allows the algorithm to only accept part of the input space and $\psi(x,y)$ typically expresses that y is a valid output for the input x. In cases where x is not a valid input, we use the new symbol $\perp$ to denote that the algorithm does not return anything and use instead of Y the following set of output values $Y^\perp = Y \cup \{\perp\}$. The certificate or *witness* from a set W and its correctness is expressed by the predicate $\mathcal{W} : X \times Y^\perp \times W$ with the following properties:

1. **Strong witness property**: Consider a triple (x,y,w) that satisfies the witness predicate $\mathcal{W}$. If $y = \perp$, i.e. the input is not valid, we want w to be a proof of this fact. Otherwise, we have that $y \in Y$ and we want w to be a proof that the postcondition is satisfied. In total, we have the following requirements:

$$\forall x, y, w.\ (y = \perp \wedge \mathcal{W}(x,y,z) \to \neg\phi(x)) \wedge (y \in Y \wedge \mathcal{W}(x,y,z) \to \psi(x,y))$$

2. **Simplicity**: The strong witness property has a simple proof

3. **Checkability**: It is possible to check efficiently if $\mathcal{W}(x,y,w)$ holds for a triple (x,y,w)

A *(strongly) certifying algorithm* is an algorithm that stops on all inputs $x \in X$ and returns a tuple $\langle y, w \rangle$ such that $\mathcal{W}(x,y,w)$ holds.

This is illustrated in the following examples. We start with an example from graph theory.

Example 13 ([28])**.** *Consider the problem of deciding whether a graph $G = (V;E)$ is bipartite, i.e. there exists a partition of V into V_1, V_2 with $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$, such that for all edges $e \in E$ we have that $e \cap V_1 \neq \emptyset$ and $e \cap V_2 \neq \emptyset$, so that all edges are only between vertices that are in the different partitions.*

The set of inputs X is the set of strings over $\{0,1\}$ and $\phi(x)$ holds whenever x encodes a graph. The postcondition is the following: If $y = true$, then x encodes a bipartite graph. If $y = false$, then x does not encode a bipartite graph.

Adopting the well-known algorithm for checking whether a bipartite graph, we can construct the following certifying algorithm.

1. *Check if x encodes a binary graph. If not return $\perp$ and a witness that describes the problem in the encoding.*

2. *Explore the graph in depth-first search and color vertices along the path by alternating two colors. If we want to color a vertex by a color c and it already has the color c', we continue exploring another path if $c = c'$, or stop if $c \neq c'$ and return $false$. If $c \neq c'$ is the case, we have found a cycle of odd length in our current path and return this cycle as a witness.*

3. *If we reach this step, then all vertices are colored in a way that no neighboring vertices have the same color. Then we return true and the two sets of colored vertices as the witness.*

We know from graph theory that a graph is bipartite iff it has no cycle of odd length.

For any value of y a checker can efficiently check if the returned reason does indeed hold for x.

The above example illustrates the concept well for a decision problem, but we want to verify whether a set of ground atoms is the semantics of a datalog program and database, i.e. a function problem. The next example illustrates a certifying algorithm for a function problem in number theory.

Example 14 ([28])**.** *Let X be the set of pairs of natural numbers and Y the set of natural numbers. We want to compute the greatest common divisor, gcd, for two numbers a and b that are not equal to zero, i.e. the largest $g \in \mathbb{N}$ that is a divisor of both a and b. The precondition $\phi(\langle a, b\rangle)$ is that both a and b are different from zero and the post-condition is that $\psi(\langle a, b\rangle, g)$ shall be true if g is the greatest common divisor of a and b.*

The normal way of computing this involves the Euclidean algorithm, but this is not a certifying algorithm. The extended Euclidean algorithm however returns in addition to the greatest common divisor g two integers s and t such that $g = gcd(a, b) = as + bt$. In [28], the back direction is proven, i.e. that if g is a divisor of a and b and $g = as + bt$ holds for two integers s and t, then g is also the greatest common divisor of a and b.

Therefore a certifying algorithm to compute the gcd is the extended Euclidean algorithm, that returns as y $gcd(a, b)$ and as the certificate the pair $\langle s, t\rangle$.

A checker then only has to check that $g = as + bt$ holds and that g divides both a and b.

For decision problems, we require an explanation when the input is in a language as well as an explanation when the input is not in the language. A common misconception is that therefore certifying algorithms are only possible for problems that are known to be in $NP \cap coNP$, which is presumed to not include the interesting NP-complete problems. This is not the case as we do not require the certificate to be polynomial in the size of the input. Indeed, SAT-solvers that decide the NP-complete satisfiability problem of propositional logic offer certificates in the DRAT proof format as a model or a proof of unsatisfiability[33].

In the datalog case, many modern reasoners such as Nemo or Soufflé already offer certificates for atoms in the form of the previously introduced proof trees. We therefore focus on implementing a checker for datalog.

In [2] an approach to develop checkers for certifying algorithms is outlined. There the checker is implemented in C because already the certifying algorithm itself is written in C++, so that both versions are similar. In order to verify the correctness of the checker they employ the proof assistant Isabelle. The C-code is exported by a tool into Isabelle so that the verification can take place. In Isabelle then the problem is defined and it is proven that the export does fulfill the desired properties.

In this workflow, we have to trust the correctness of the hardware, the C compiler, the tool that exports C code to Isabelle and Isabelle itself. Additionally, we have to trust that we have correctly defined and specified the problem in Isabelle.

In this work, we modify this workflow a bit. We use a different proof assistant, Lean, instead of Isabelle. Lean is not only a proof assistant but also a functional programming language. Therefore we can implement the checker directly in Lean and do not have to export a model of our code into the proof assistant. Our trust base is therefore smaller, as we no longer need an additional compiler for the language the checker is written in nor a tool to export this into the proof assistant.

2.3 Lean

In this section, we introduce Lean, a programming language and proof assistant. As of writing, the current version is Lean 4. A more in-depth introduction can be found in [23] or [11].

Lean's core is a small trusted kernel[18], that captures the most important functionalities and can be extended by the user. Since version 4 Lean is implemented in Lean itself[17]. The most important libraries for Lean that we will use are the standard library Std4[13] which contains important data structures and tactics and mathlib4[12], which contains definitions, tactics and theorem from diverse areas of mathematics and computer science.

Lean can encode many mathematical results. The foundations of mathematics are often built on top of set theory (e.g. [21]) but most proof assistants instead use *type theory*. Type theory was introduced by Bertrand Russel as an alternative foundation during the foundational crisis of mathematics.

Any element, a *term*, has a *type* in type theory. This is denoted by `term:type`.

Example 15. *Different terms and their types are displayed below:*

```
42:nat
true:bool
sort([1,2,3]): List (nat)
nat: Type
Type: Type₁
```

`nat` is a natural number, `true` is a boolean, the result of the application of `sort` to `[1,2,3]` has the type `List nat`. All previous types have a type as well, `Type`. `Type` itself also has a type `Type`$_1$. This forms an infinite sequence of type universes, that are non-cumulative, i.e. no term has multiple types.

In contrast to set theory, functions are a direct element of type theory and do not need a complicated encoding. For any types `A` and `B`, there exists the type of functions from A to B, `A → B`.

We can define functions in Lean by the keyword `def` similar to other functional programming languages, such as the id function for natural numbers. In the parentheses, we denote an argument `x` with its type. Due to currying a function can have multiple arguments. After the colon, we denote the returned type of the function, which is here again a natural number. Therefore we have the following type $\mathbb{N} \to \mathbb{N}$, as the function takes a natural number as an input and returns a natural number as well. After the `:=` we denote the natural number this function returns.

```
def id (x: ℕ): ℕ:= x
```

Everything we have seen so far holds in general in type theory. Lean itself is based however on a specific type theory, the Calculus of Inductive Constructions (CIC), a dependent type theory[17, 15].

The id function we defined above is correct but only works for natural numbers. We would have to implement another id function for booleans, string or any other type. This is rather cumbersome because we never actually use any properties of the specific type. We can generalize the id function to help us with this problem.

```
def id (A: Type)(a: A): A:= a
```

This id function has now two arguments. The first argument is the type we want to use it on, the second argument is an element of this type. `id ℕ` works as previously and has the type $\mathbb{N} \to \mathbb{N}$. It seems less clear which type id itself has because this depends on the first argument. Such types are called *dependent types* and we would denote this specific type as `Π (A:Type) A → A`, known as a *Pi type*. Intuitively, a Pi type is a function that returns a type for any input value.

In practice, this is often written in the following way:

```
def id {A: Type}(a: A): A:= a
```

Here the type A is an implicit argument, which is denoted by the braces, and Lean can fill it in for the user based on the input a. Therefore the user can write just `id 42` instead of `id ℕ 42`.

So far we have discussed what types are and how we can define functions that are important for the programming aspect of Lean, but this does not tell us much about the use as a proof assistant. Due to the Curry-Howard correspondence a proof of a statement can be equivalently seen as a function that has the type of the statement. The type of statements is known in Lean as `Prop`. There exist functions like `Or: Prop → Prop → Prop` and the dependent types form the quantifiers. We view the Pi-type $\Pi(a : A), \beta(a)$ as a function mapping a to the true statements of $\beta(a)$, which is equivalent to the universal quantor. Prop therefore has elements such as $\forall(x : \mathbb{N})$`, 0 ≤ x` or `∃ (x y: Bool), x = y ∨ ¬ x = y`.

Every statement defines its own type, whose elements are the proofs for it. In contrast to other types, all elements of these types are equivalent which is known as *proof irrelevance*, so that elements of these types only serve as a witness for their truth.

Proofs can be constructed in Lean in two ways. We start by writing `theorem` (or lemma, proposition etc. in mathlib) and denote the statement as we did for a function. The main difference is that the type is what we want to prove and that the function can not be executed. An example of a statement and its proof in Lean is shown below:

```
theorem testTheorem {A: Type}: ∀ (x y z: A), x = y → id y = z →
    z = x := by
  intro x y z h1 h2
  unfold id at h2
  rw [h1]
  apply Eq.symm
  exact h2
```

The proof is done in tactic mode. We open the tactic mode with the keyword `by`. At the start our context only contains that A is a type and the goal is $\forall(x\ y\ z\ A), x = y \to id\ y = z \to z = x$, which is compactly written as

$$\{A : Type\} \vdash \forall(x\ y\ z : A), x = y \to id\ y = z \to z = x$$

We apply tactics to change the goal or a hypothesis in the context. These may introduce new goals or fulfill old goals. A proof is finished, when all goals are proven. We explain the proof now in more detail.

1. We start with the `intro` tactic. This tactic moves elements that are universally quantified or the start of an implication from the goal into the context. Alternatively, one can use `revert` to move a hypothesis from the context back to the goal to prove a stronger statement.

   At the end, the state is:

$$\{A : Type\}(x\ y\ z : A)(h1 : x = y)(h2 : id\ y = z) \vdash z = x$$

2. After that we use the `unfold` tactic at the hypothesis h2. This tactic replaces `id` with its definition in h2, which now looks like this: $(h2 : y = z)$

3. We use now the tactic `rw`, i.e. rewrite, that replaces in the goal x by y due to the hypothesis $h1$ so that the goal is now $z = y$.

4. The goal is now almost the same as $h2$, but the order is wrong. Therefore we use the apply tactic with

   `Eq.symm :=` $\forall\{A : Type\}(ab : A), a = b \to b = a$.

   Apply tries to match the statements consequent with our goal and opens new goals for the antecedents of this statement.

   Our current proof state is then this:

$$\{A : Type\}(x\ y\ z : A)(h1 : x = y)(h2 : y = z) \vdash y = z$$

5. Our goal is now exactly the same as the hypothesis $h2$. The `exact` tactic uses $h2$ to complete the goal.

Other important tactics not seen here are `by_contra` which allows a proof by contradiction, `by_cases` which allows a case distinction or `constructor` that splits an and statement into two goals. Lean offers also methods that try to prove goals themselves like `simp` that uses lemmas marked with `@simp` or `tauto`, that can recognize some tautologies. The set of tactics is however not fixed and the user can introduce new tactics to simplify the reasoning process.

The majority of formal proofs in this work are done in tactics mode, but we see above that describing them in natural language is quite verbose. Therefore we will only describe the idea we followed in the following sections. Any result is also linked to the code in the later sections and can be directly accessed via the hyperlink to see the formalization instead.

These proofs are done backwards, i.e. starting from the goal towards the assumptions. There are also forward proofs that start from the assumptions by building hypotheses until we reach the goal or by composing theorems like functions. While this is possible, it was not done in this work. A good source for this is either [23] or [4].

We can also use `def` to define new types. Sets of type `A` are in Lean implemented as functions from `A` to `Prop`. Functions to `Prop` are in general not computable so that the membership in a set is also not computable in contrast to membership in lists.

```
def Set (α : Type) := α → Prop
```

This introduces a new type. If we want to just use functions that accept certain elements of type `Set`, using `def` would create a new type for which we would have to define again the set operations ∪ or ∩. In this case, one can use the `abbrev` command which provides a simple alias.

```
abbrev NatSet := Set ℕ
```

The CIC and Lean allow the creation of types by induction. We can create our own implementation of the natural numbers with the `inductive` keyword. After that, we list the constructors. These can be constant constructors such as `zero` or function constructors like `succ`. An inductive type may have many constructors of both types or even none as the empty type.

```
inductive myNat: Type
| zero: myNat
| succ: myNat → myNat
```

Our definition states that an element of `myNat` is either `zero` or the result of applying `succ` to some other element of `myNat`. Any element of `myNat` is only the result of one of these constructors and all elements of `myNat` are distinct.

A typical mathematical definition for this would be: myNat is the smallest set S of elements that contains *zero* and whenever an element a is in S, then also *succ* a is in S.

We can define for inductive types functions recursively like the add function. Here we reuse the inductive schema. If m is zero, then $n + m$ is simply n. If m instead is the successor of some other myNat m', then we return the successor of `add n m'`.

```
def myNat.add (n m: myNat): myNat :=
match m with
| zero => n
| succ m' => succ (add n m')
```

If we want to prove a statement about elements on an inductive type, we can use the `induction` tactic to create proofs similar to those by (structural) induction in natural language proofs. If no induction hypothesis is needed one can also use the `cases` tactic that creates a goal for every constructor. We illustrate the induction tactic by proving that adding to zero any myNat n yields n.

```
theorem myNat.add_zero (n: myNat): myNat.add zero n = n := by
    induction n with
    | zero =>
        unfold add
        rfl
    | succ n' ih =>
        unfold add
        rw [ih]
```

We start with the induction tactic to perform an induction over n. Each constructor is listed separately and after the arrow, we start the proof for this constructor. In the zero case, it is enough to unfold the definition of add to get `zero=zero` and use `rfl` which proves these equalities.

In the succ case, we have two values: the myNat element n' we use in the constructor of succ and also the induction hypothesis`ih: add zero n' = n'`.

We can unfold the definition of add again and use `rw` to replace `add zero n'` by `n'` in the goal `succ (add zero n') = succ n'` due to the induction hypothesis `ih`. The `rw` tactic applies `rfl` at the end automatically to finish the proof.

After defining the natural numbers inductively, we can also define the even numbers by induction. We again have the zero constructor and a succ constructor that now shall represent the next natural number.

```
inductive evenNat: Type
| zero: evenNat
| evenSucc: evenNat → evenNat
```

While this works, we see no relation between the elements of myNat and evenNat. We can of course define a function that maps elements of evenNat to myNat and use this to transform any even number to a natural number.

```
def evenNatToMyNat (e: evenNat) :=
    match e with
    | zero => myNat.zero
    | succ e' => myNat.succ (myNat.succ (evennatToMyNat e'))
```

This approach works but is a bit cumbersome. We always need to type this ourselves and if we forget it Lean raises a type error. There exists however the `class Coe`. A class describes an abstract property e.g. of a type and can be implemented by a specific type via `instance`.

The class `Coe` is defined (in a simplified version) as follows. It takes two types and a function that maps from one type to the other. We implement `COe` for `evenNat` and `myNat` with `evenNatToMyNat`. Then we can use elements of `evenNat` in functions defined for `myNat` and Lean will automatically convert them using `evenNatToMyNat`.

```
class Coe (α : Type) (β : Type) where
    coe : α → β

instance coeEvenNatMyNat: Coe evenNat myNat := ⟨evenNatToMyNat⟩
```

An alternative way of defining types are structures that are comparable to classes in object-oriented languages. Structures have fields that have a type and we start listing them one by one after the `where` statement. Structures are like inductive types with a single constructor. Therefore they are not allowed to be recursive, i.e. no structure A may have an element of type A, because there would be no base case. We can build an element of the type of the structure by passing the elements in braces and can use the name of the field to access an element.

```
structure player where
    (name:String)
    (numGoals: ℕ)
    (active: Bool)
```

```
def player1: player := {
        name := "Test",
        numGoals := 10,
        active := true
    }

    def player1Goals: ℕ := player1.numGoals
```

Returning to our example of myNat, we can define some more functions. In the previous section, we talked about the Euclidean algorithm. We can try to implement it ourselves in Lean. Firstly, we would require the modulo function for that. We will only declare the function for now to not lose focus and use the keyword `sorry` in the value. This closes in proofs any goal or can be used for incomplete functions, but will throw an error if executed.

```
def myNat.mod (n m: myNat): myNat := sorry
```

The Euclidean algorithm can then be implemented in the following way:

```
def myNat.Euclid (a b: myNat): myNat :=
if b = zero
then a
else Euclid b (myNat.mod a b)
```

Lean will not accept this function and instead raise an error in the line of the if that it failed to synthesize an instance of `DecidableEq` for myNat. DecidableEq allows us to use = between elements of a type in a function. This is not a trivial requirement as the equality of real numbers is undecidable[30]. We need to provide a function to Lean that can be used to decide whether two elements of myNat are equal. During the definition of inductive types, we already noted, that all elements of an inductive type are different, which allows us to do this. Luckily, Lean can even do this alone if we tell Lean to do this using the `deriving` keyword. It can even derive instances for multiple classes. Another important class would be the Inhabited class which states that there exists some default element of this type.

```
inductive myNat: Type
| zero: myNat
| succ: myNat → myNat
deriving DecidableEq, Inhabited
```

Now the error will disappear, but a new one will be introduced. Lean does not recognize the termination of `myNat.Euclid`. We have already defined recursive functions such as `myNat.add` before and we did not encounter this problem. In `myNat.add` we however used the inductive schema and only did the recursion on n in the case of *succ* n. This allowed Lean to conclude that we only call the function on smaller elements so that the function will terminate.

To show termination, one is required to define some *well-founded relation*, i.e. a relation that does not have any infinite descending chains. Such a relation would be the less-than relation of the natural numbers so it is often convenient to define some size measure of the elements of a type. Such a function exists automatically for any inductive type and is called `sizeOf`. We can provide this and are only left to prove that `sizeOf (mod a b) < sizeOf b`, which we will leave open here.

```
def myNat.Euclid (a b: myNat): myNat :=
  if b = zero
  then a
  else Euclid b (myNat.mod a b)
termination_by sizeOf b
decreasing_by
  simp_wf
  sorry
```

An alternative to this termination proof would be to mark this function using the `partial` keyword, which denotes that this function does not always terminate. This is however often not beneficial as we no longer can use the `unfold` tactic to gain the definition of this function in a proof.

We want to take a look at another example of an inductive type, that we will often use namely lists. A list (here List' as List is already a standard definition) is either the empty list or the result of combining a list with a new element and we can define a get function

```
inductive List' (A: Type): Type
| nil : List' A
| cons : A → List' A → List' A
deriving DecidableEq

def getFirstElementOrDefault (A: Type) [Inhabited A] (l: List'
    A): A :=
match l with
| nil => Inhabited.default
| cons hd _ => hd
```

We see in this definition a new type of argument. We want to return an element of the list. If the list is empty, this proves to be difficult. In that case, we return the default element that exists because A is an instance of `Inhabited`. Such a type-class parameter is given in brackets and Lean will look for the instance that shows the implementation of the type class for this type when using this function.

This may not always be the desired behaviour and maybe we want to return nothing if the list is empty. In this case, we can use the `Option` Type. It has two constructors, one contains the element and the other symbolizes that there is nothing. Here α has the type `Type u` that symbolizes it works for types from any type universe. Then we can return none if the list is empty and else return the same value as before.

```
inductive Option (α : Type u) where
```

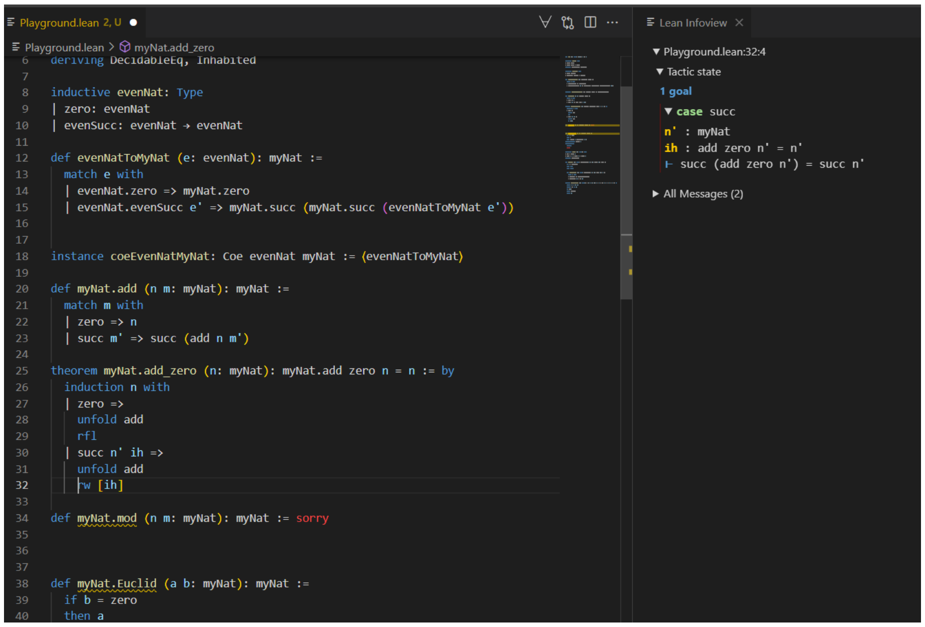
```
| none : Option α
| some (val : α) : Option α

def getFirstElement (A: Type) (l: List’ A): Option A :=
match l with
| nil => Option.none
| cons hd _ => Option.some hd
```

We will also often use the Exception type. This uses two types ϵ and α. The type ϵ is the error we report and α are the elements we return if no error occurs.

```
inductive Except (ε : Type u) (α : Type v) where
error : ε → Except ε α
ok    : α → Except ε α

f getFirstElement (A: Type) (l: List’ A): Except String A :=
match l with
| nil => Except.error "Called getFirstElement on emptyList"
| cons hd _ => Except.ok hd
```



```
deriving DecidableEq, Inhabited

inductive evenNat: Type
| zero: evenNat
| evenSucc: evenNat → evenNat

def evenNatToMyNat (e: evenNat): myNat :=
  match e with
  | evenNat.zero => myNat.zero
  | evenNat.evenSucc e' => myNat.succ (myNat.succ (evenNatToMyNat e'))


instance coeEvenNatMyNat: Coe evenNat myNat := ⟨evenNatToMyNat⟩

def myNat.add (n m: myNat): myNat :=
  match m with
  | zero => n
  | succ m' => succ (add n m')

theorem myNat.add_zero (n: myNat): myNat.add zero n = n := by
  induction n with
  | zero =>
    unfold add
    rfl
  | succ n' ih =>
    unfold add
    rw [ih]

def myNat.mod (n m: myNat): myNat := sorry



def myNat.Euclid (a b: myNat): myNat :=
  if b = zero
  then a
```

```
▼ Playground.lean:32:4
 ▼ Tactic state
  1 goal
   ▼ case succ
   n' : myNat
   ih : add zero n' = n'
   ⊢ succ (add zero n') = succ n'
▶ All Messages (2)
```

Figure 2.5: An example of Lean in practice. On the left side is the code seen and on the right side the context in a proof with the current goal is displayed

Chapter 3

Formalization of Datalog

In the previous section, we introduced datalog. Our goal is to check whether ground atoms are the result of correct datalog derivations. A correctness proof requires us to define what correct derivations are which we solve by formalizing the syntax and semantics of datalog in Lean. As of writing, there is to the best of our knowledge no formalization of datalog in Lean yet.

There does exist a formalization of datalog in Coq[8] which includes the syntax of datalog and the fixed point semantics of datalog with a datalog engine that is proven to be correct. Similarly, we will also formalize the syntax of datalog. After that, the paths will diverge as we are interested in the proof-theoretic semantics of datalog to check proof trees. Additionally, we will also formalize the model-theoretic semantics of datalog for completeness arguments and to have extra security that our formalization is correct by proving that both semantics are equivalent.

3.1 Syntax

We recall from the preliminaries that an atom is of the form $A(t_1, \dots, t_n)$ for sets of relation symbols, variables and constants R, V and C and an arity function $ar : P \to \mathbb{N}$. If we were to directly formalize this an atom would be defined in the following way, where we use types instead of sets.

```
def atom (C: Type) (P: Type) (V: Type) (ar: P → ℕ): Type :=
    sorry
```

Such a definition is rather verbose with already four arguments. Anything like the semantics that will use atoms will require even more arguments. To have a more compact representation we reuse the definition of a signature that shrinks the number of arguments to just one. We use types instead of sets as this is more natural in type theory. If we were to use sets we would have to decide already which type these sets should have which seems unclear for now. This allows us to instead define later the types we want to use. Even if this is not an issue and we decide to use for relation symbols a set R of type `Set A`. Then any relation symbol r also has the type A but R might not contain every element of the type A, so that we additionally require a proof that r is in R. Such a membership is not required when using a type for the relation symbols as a term which would be the relation symbol r has always exactly one type.

The formalization showed us that the requirements of countability of the sets R, C and V were not required. Any result we wanted to prove holds already in this general case. Therefore we forego of modelling this assumption.

A `signature` is then a structure that has a type for constants, variables[1] and relation symbols and an arity function.

```
structure signature where
  (constants: Type)
  (vars: Type)
  (relationSymbols: Type)
  (relationArity: relationSymbols → ℕ)
```

In the following, unless denoted otherwise, we will always use a fixed signature τ and assume that all types have instances for `DecidableEq` and `Hashable` classes to use Lean's automatic derivation for later uses in the program.

Another requirement was that the sets of constants and variables are distinct. This proved again to be unnecessary in our formalization because we define terms as an inductive type with a constructor for the constant and the variable case. Therefore constants and variables in an atom will always be distinct.

```
inductive term (τ: signature): Type
| constant : τ.constants → term τ
| variableDL : τ.vars → term τ
deriving DecidableEq, Hashable
```

For an atom we have fields for the symbol, the list of terms and a proof that the length of the list of terms matches the arity of the symbol.

```
structure atom where
  (symbol: τ.relationSymbols)
  (atom_terms: List (term τ ))
  (term_length: atom_terms.length = τ.relationArity symbol)
deriving DecidableEq, Hashable
```

Two structures of the same type are equal iff all their fields are equal. Due to proof irrelevance, we gain the following expected criteria for the equality of atoms.

Lemma 2 (`atomEquality`). *For all τ-atoms a_1, a_2, we have $a_1 = a_2$ iff their `symbols` and `atom_terms` are equal.*

Rules and programs can be modeled straightforwardly. Here we use lists to have the body in a fixed order because this simplifies later algorithms.

[1]We use vars instead because variable(s) is a keyword in Lean

```
structure rule where
  (head: atom τ)
  (body: List (atom τ))
deriving DecidableEq

abbrev program := Finset (rule τ)
```

Next, we want to define the ground versions of our previous definitions. Groundings are simply the functions from variables to constants.

```
def grounding (τ: signature):= τ.vars → τ.constants
```

We see multiple ways to define ground atoms. Firstly, we can define them like we defined atoms but use constants instead of terms.

```
structure groundAtom (τ: signature) where
  symbol: τ.relationSymbols
  atom_terms: List (τ.constants)
  term_length: atom_terms.length = τ.relationArity symbol
deriving DecidableEq, Hashable
```

Secondly, we can define ground atoms as a special type of atom by constructing a new structure consisting of an atom and a proof that for all terms in the atom exists some constant that is equal to it.

```
structure groundAtom (τ: signature) where
    atom: atom τ
    ground: ∀ (t: term τ), t ∈ atom.atom_terms → ∃ (c: τ
    .constants), t = term.constant c
```

The second variant allows us an easy conversion from ground atoms to atoms by simply returning the atom element. Also, we can convert atoms easily to ground atoms as soon as we have the proof. These conversions have to be written by hand in the first variant.

The first variant on the other hand allows us to define functions that create ground atoms more directly. We can take a grounding and just map the term list using the grounding without having to provide a proof that all terms of the atom are then equal to some constant. In the second variant, we have to first define the function on the atom level and then have to prove that this operation does not create any variables. This may sound like extra security in case we mess things up, but when defining the terms as a list of constants the type checker of the kernel does the check for us.

The number of conversions is rather limited whereas an easier way to define functions may be useful more often. Therefore we chose the first variant.

We start by defining these conversions that we now have to do manually. We can convert a ground atom to an atom by mapping every constant to a term via `term.constant`. The map operation does not change the length of a list of atoms so that the term length property is preserved(`listMapPreservesTermLength`).

```
def groundAtom.toAtom (ga: groundAtom τ): atom τ:= {
    symbol:=ga.symbol,
    atom_terms:= List.map term.constant
    ga.atom_terms,term_length:= listMapPreservesTermLength ga
}
```

For later proofs, it is useful to know that this is an embedding of the ground atoms into the atoms, i.e. that if two ground atoms are different then the result of their toAtom functions is also different.

For this, we need the following result (listMapInjectiveEquality): If two lists l_1, l_2 are equal then mapping both lists by the same function f results in the same list l. If the function is injective the back direction holds as well, i.e. if mapping l_1 and l_2 using f yields the same list l then also l_1 and l_2 are equal, which we can prove by induction.

Lemma 3 (groundAtomToAtomEquality)**.** *Let a_1, a_2 be two ground atoms. Then a_1 is equal to a_2 iff the result of `groundAtom.toAtom` of both is equal.*

Proof. If $a_1 = a_2$, then also the result of applying `toAtom` to both of them is the same.

For the back direction, we know that the results of applying `toAtom` to a_1 and a_2 are equal and want to show that a_1 is equal to a_2. We use a similar lemma as lemma 2 for ground atoms(groundAtomEquality). Therefore we have to show that their symbols and terms are equal. As `toAtom` does not change the symbol, the first part follows. For the second part, we have to show the term lists of a_1 and a_2 are equal. We know that mapping every constant in the atom terms of a_1 and a_2 via the `term.constant` function yields the same list. Any inductive constructor such as `term.constant` is injective [2] and hence we can use listMapInjectiveEquality to see that the term lists of a_1 and a_2 are equal. □

We can therefore employ `groundAtom.toAtom` safely as the type coercion from ground atom to atom.

In order to convert atoms to ground atoms, it would be enough to use a proposition that says that all elements are constants. In later uses such as the definition of safety for rules, it will be beneficial to have a function that computes all variables that occur in an atom. If this returns the empty set, this is the required proof to convert an atom into a ground atom.

We can define the variables occurring in a term as a finite set.

```
def termVariables: term τ → Finset τ.vars
| (term.constant _) => ∅
| (term.variableDL v) => {v}
```

The variables occurring in an atom are then the union of the term variables of its terms. This can be recursively expressed using `List.foldl`. We start with the empty set and return the union with this set and the term variables of every term occurring in the atom by recursively taking the union of the previous result and the term variables of the element at the start of the list.

[2]This can be shown with the `injection` tactic

```
def List.foldl_union {A B: Type} [DecidableEq B]
(f: A → Finset B) (init: Finset B) (l: List A): Finset B :=
 List.foldl (fun x y => x ∪ f y) init l

def atomVariables (a: atom τ) : Finset τ.vars :=
List.foldl_union termVariables ∅ a.atom_terms
```

If the term variables are empty, we know that the term must be a constant. We can convert a term to a constant with a function that takes a term and a proof that its term variables are empty. In the variable case, which is impossible to occur given the arguments as we have a proof that there are no term variables, we use `False.elim` which uses that anything follows from a false proposition to generate an object of the right type. This can never be returned and is only there for completeness.

```
def termWithoutVariablesToConstant (t: term τ) (h: termVariables
    t = ∅): τ.constants :=
  match t with
  | term.constant c => c
  | term.variableDL v =>
      have h': False := by
        unfold termVariables at h
        simp at h
      False.elim h'
```

We use this function to convert atoms with no variables to ground atoms. If the atom variables of some atom a are empty, then also the term variables of every term t in a are empty (`atomVariablesEmptyIffAllTermVariablesEmpty`). Using this we can call `termWithoutVariablesToConstant` on every term in the atom. We require a proof that an element is in the list, which is given by `List.attach` together with the element itself.

```
def atomWithoutVariablesToGroundAtom (a: atom τ) (h:
    atomVariables a = ∅): groundAtom τ :=
{
  symbol:= a.symbol,
  atom_terms := List.map (fun ⟨x, _h⟩ =>
    termWithoutVariablesToConstant x (Iff.mp
    (atomVariablesEmptyIffAllTermVariablesEmpty a) h x _h))
    a.atom_terms.attach,
  term_length :=
    by
      simp
      apply a.term_length
}
```

After defining the conversions we define ground rules similar to rules.

```
structure groundRule (τ: signature) where
```

```
  head: groundAtom τ
  body: List (groundAtom τ)
deriving DecidableEq
```

We can apply a grounding to a term by replacing a variable by its grounding result and keeping the constant.

```
def applyGroundingTerm (g: grounding τ) (t: term τ): term τ :=
  match t with
  | term.constant c => term.constant c
  | term.variableDL v => term.constant (g v)
```

Using this function we apply groundings also to atoms and rules.

```
def atomGrounding (g: grounding τ) (a: atom τ): groundAtom τ := {
symbol := a.symbol,
atom_terms := List.map (applyGroundingTerm'  g) a.atom_terms,
term_length := applyGroundingTerm'PreservesLength  g a
}

def ruleGrounding (r: rule τ) (g:grounding τ): groundRule τ := {
    head:=atomGrounding g r.head,
    body:= List.map (atomGrounding g) r.body
}
```

The ground program of a program P is the set of all ground rules that are the result of the application of a grounding to a rule from P.

```
def groundProgram (P: program τ) :=
{r: groundRule τ | ∃ (r': rule τ) (g: grounding τ), r' ∈ P ∧ r =
     ruleGrounding r' g}
```

3.2 Semantics

After finishing the definition of the syntax, we start formalizing the semantics. We mentioned in the preliminaries that the semantics can be defined with or without a database. We decided to formalize the semantics with the database as this is more general. It is simpler to pass an empty database into the checker than to write every fact from the database into the rule file.

In this section, we do not want to deviate too much from the path by implementing databases in a complicated way. For now, a database is simply something that has a contains function that returns true if an element is in the database. This class can be implemented in multiple ways in the algorithm later.

```
class database (τ: signature) where
  (contains: groundAtom τ → Bool)
```

We call a set of ground atoms an interpretation as in the preliminaries.

```
abbrev interpretation (τ: signature) := Set (groundAtom τ)
```

We start by defining the proof-theoretic semantics as proof trees are our certificates in the checker so this is the most important semantic for us.

We require the notion of a tree to formalize the proof-theoretic semantics. In mathlib exists a definition of a tree[3], but this is despite its name only a binary tree. With Mathlib's trees, we could only represent rules in the proof tree whose body has at most length two. While any datalog program can be transformed into an equivalent program that only contains rules whose body has a length of at most two, this is not desirable as rules in practice are often longer. We could require the user to just transform the rules into this form, but this seems to hinder the acceptance of this tool and may mask errors that only occur in optimizations for longer rules.

Therefore we start by defining a tree as an element t with as a vertex a and a list of subtrees l. We call a the root of t and the root of the trees in l the children of t. A leaf is an element whose subtree list is empty.

```
inductive tree (A: Type)
| node: A → List (tree A) → tree A

def root: tree A → A
| tree.node a _ => a

def children: tree A → List A
| tree.node _ l => List.map root l
```

An important measure for trees is the height defined as the longest path from the root to a leaf. An alternative recursive definition is that it is the maximum of the height of all subtrees plus one, which we can implement with the listMax function.

```
def listMax {A: Type} (f: A → ℕ): List A → ℕ
| [] => 0
| (hd::tl) => if f hd > listMax f tl then f hd else listMax f tl

def height (t:tree A): ℕ :=
  match t with
  | tree.node a l => 1 + listMax (fun ⟨x, _h⟩ => height x)
    l.attach
termination_by sizeOf t
decreasing_by
    simp_wf
    apply Nat.lt_trans (m:= sizeOf l)
    apply List.sizeOf_lt_of_mem _h
    simp
```

[3]see at https://leanprover-community.github.io/mathlib4_docs/Mathlib/Data/Tree.html

Our implementation looks a bit different compared to the text. This is because we are required to prove the termination of this function. We call the height function recursively, but with a list function so that we no longer follow the inductive schema directly. The well-founded relation will be the less-than relation between the sizeOf results of the trees. Therefore we have to prove that sizeOf x < 1 + sizeOf a + sizeOf l holds to show the termination. The term x is one of the subtrees we call height on and by transitivity, it suffices to prove that sizeOf x < sizeOf l holds. The size of any member of a list must be smaller than the size of the list itself by the implementation of the sizeOf function. The List.attach function that takes a list l and maps any element a from l to an element of { a // a ∈ l}, so a pair of the original element and a proof that it is a member of l. This allows us to complete the proof.

We call a tree t_2 a member of a tree t_1 if it occurs in the subtrees of t_1 and can prove that any member has a smaller height than the original tree (heightOfMemberIsSmaller) because the result of f for some member a of l is less-equal to the value of listMax f l(listMax_le_f_member).

With this, we finish the general results about trees and can return to formalizing the proof-theoretic semantics. A proof tree is a tree whose vertices are ground atoms.

```
abbrev proofTree (τ: signature):= tree (groundAtom τ)
```

Some proof trees will not represent valid derivations by our definition. The next step is to define a predicate that shows the validity of a tree similar to our definition in the preliminaries.

Again, we want to design our checker with the best compatibility in mind. In some papers (e.g. [10]) the leaves of a valid proof tree have to be database elements. We leave this open to allow also facts from the program to serve as leaves for those programs that do not come with a database.

We express the validness of a proof tree with the root a and the list of subtrees l with respect to a program P and a database d by a disjunction with two disjuncts. We call the first the *rule case*. There we require a rule r and a grounding g, such that r is in P, the rule grounding of r with g yields the ground rule we gain from the root and its children. Additionally, all subtrees must be valid, which we again express via List.attach for the termination proof. The second case is the *database case*. There a must be a leaf and contained in the database. This allows facts from the program as the leaves of a tree as they fulfill the rule case.

```
def isValid (P: program τ) (d: database τ) (t: proofTree τ):
    Prop :=
  match t with
  | tree.node a l => ( ∃(r: rule τ) (g:grounding τ), r ∈ P ∧
    ruleGrounding r g = {head:= a, body:= (List.map root l)} ∧
    l.attach.Forall (fun ⟨x, _h⟩ => isValid P d x))
  ∨ (l = [] ∧ d.contains a)
```

```
termination_by sizeOf t
decreasing_by
  simp_wf
  decreasing_trivial
```

We see from this definition that any element that is contained in the database has a simple proof tree. The tree is just the element as the root without any subtrees (databaseElementsHaveValidProofTree).

We do not consider input or export relations so that the database elements are always part of the semantics. This also simplifies the definition for isValid in contrast if we only want elements that are the result of some rule to be in our semantics.

The proof-theoretic semantics with respect to a program P and database d is the set of ground atoms that are the root of a valid proof tree with respect to P and d. We avoided earlier defining the types of a signature as a finite type so that we cannot expect a finite set here. We will manage to prove the same results and do not have to prove that there are only finitely many ground atoms.

```
def proofTheoreticSemantics (P: program τ) (d: database τ):
    interpretation τ:=
{a: groundAtom τ | ∃ (t: proofTree τ),root t = a ∧ isValid P d t}
```

The proof-theoretic semantics provides us with proof trees a good explanation why an element is part of the solution. We only have to verify that the proof tree is correct. This however tells us only that we have found a subset of the solution. It tells us not whether there are more derivations possible. Passing an empty list of proof trees would always be accepted but this is in general not the complete solution. The other semantics are better equipped to deal with this problem. They describe the solution as the least element of some set. If we verify that the result is in this set (i.e. the set of models or the set of fixed points), then our result is a superset of the solution. If both criteria hold, then we have exactly the solution.

Formalizing another semantics also strengthens our formalization because we might spot some wrong assumptions we made in the definitions that way. If we make an error in the definition of one semantic we may spot it when trying to prove that both semantics are equal.

Both the model-theoretic and the fixed point semantics are defined to be the least object of something. We decided to formalize the model-theoretic semantics because we can directly give the model. For the fixed-point semantics, we would first need to prove that the fixed point even exists and may need some theorems about fixed points, but such a formalization (in Coq) can be found in [8].

We start by formalizing the criteria for a rule being true. An interpretation is a set whereas a body is a list. So that we can compare them, we define the groundRuleBodySet of ground rule r as the conversion of the body to a finite set. This operation preserves the members so that a ground atom is in the groundRuleBodySet of r iff it is in the body of r.

(groundRuleBodySet_iff_groundRuleBody)

```
def groundRuleBodySet (r: groundRule τ): Finset (groundAtom τ) :=
     List.toFinset r.body
```

This allows us to define the criteria for a rule being true in an interpretation i in a natural way. Whenever all elements of the body are in the interpretation, then the head must be a member of the interpretation as well.

```
def ruleTrue (r: groundRule τ) (i: interpretation τ): Prop :=
(groundRuleBodySet r).toSet ⊆ i → r.head ∈ i
```

An interpretation is a model, if all rules from the ground rule are fulfilled. Additionally, it also needs to contain the elements from the database so that we will be able to prove the equivalence of the semantics later because all elements in the database are already in the proof-theoretic semantics.

```
def model (P: program τ) (d: database τ) (i: interpretation τ) :
    Prop :=
(∀ (r: groundRule τ), r ∈ groundProgram P → ruleTrue r i)
∧ ∀ (a: groundAtom τ), d.contains a → a ∈ i
```

Now we are equipped with the necessary tools to formalize the model-theoretic semantics. We defined the semantics in the preliminaries as the intersection of all models, i.e.

$$\bigcap_{M \text{ is model of } P} M$$

This operation is called `Set.iInter` in Lean. We would need to transform the set of interpretations to an indexed family to use it. This would require us to define a type that serves as the index and a function mapping every index to a model.

```
def iInter (s : ι → Set α) : Set α
```

We can instead define it more directly. Sets are equal whenever they have the same members by the principle of extensionality. Therefore we also try to find a formula that is true for an element whenever it is in $\bigcap_{X:\phi(X)} X$ and can use the set comprehension to obtain the model-theoretic semantics. We know that an element is in $X \cap Y$ whenever it is in both X and Y. We can generalize this to conclude that an element is in $\bigcap_{X:\phi(X)} X$ if it is in all sets X that satisfy ϕ. Therefore the model-theoretic semantics are the atoms that are in all models.

```
def modelTheoreticSemantics (P: program τ) (d: database τ):
    interpretation τ :=
{a: groundAtom τ | ∀ (i: interpretation τ), model P d i → a ∈ i}
```

We now have to prove that this is actually the least model. We start by showing that it is a subset of every model. This follows from the definitions.

Lemma 4 (`leastModel`)**.** *Let P be a program and d be a database. For all models M of P and d, the model-theoretic semantics of P and d is a subset of M*

Proving that our definition is a model takes a bit more work.

Lemma 5 (`modelTheoreticSemanticsIsModel`)**.** *For all programs P and databases d the model-theoretic semantics of P and d is a model.*

Proof. Let M be the model-theoretic semantics of P and d. We have to show that it fulfills both model criteria. We start by showing that all rules are true in M. We assume for a contradiction that there is a rule r that is not true, i.e. that the body set is a subset of M, but the head of r is not in M. Then there must be a model M' of P and d such that the head of r is not in M'. By lemma 4 we know that all elements of M must be in the model of M'. Therefore the body of r is a subset of M' and since M' is a model the head of r must be in M' which violates our assumption that the head of r is not in M'.

Now we show that all elements in d are in M as well. The proof works in the same way as before but is simpler. Suppose that this does not hold. Then there is a database element a which is not in M. By the definition of M there must exist a model M' such that a is not in M'. But any model must contain all database elements which yields the contradiction and finishes the proof. □

The remainder of this chapter is spent proving the equivalence of both semantics, i.e. the following theorem.

```
theorem SemanticsEquivalence (P: program τ) (d: database τ):
proofTheoreticSemantics P d = modelTheoreticSemantics P d
```

By the anti-symmetry of the subset operation, it suffices to show that both semantics are a subset of each other.

Our first goal is to show that the proof-theoretic semantics is a subset of the model-theoretic semantics for any fixed program P and database d. We actually prove a stronger statement by showing that all elements in the proof-theoretic semantics are in any model. By lemma 5 the model-theoretic semantics are a model so that the claim follows.

Lemma 6 (`proofTreeAtomsInEveryModel`)**.** *Let P be a program and d be a database. Let a be a ground atom in the proof-theoretic semantics of P and d. Then we have $a \in M$ for all models M of P and d.*

Proof. An element is in the proof-theoretic semantics whenever it is the root of a valid proof tree t. We prove this by strong induction on the height of t for all trees t and ground atoms a.

There are two cases when t is valid. If it is valid and in the rule case, then there exists a rule r and grounding g such that r is in P and the grounding of r, which we call r', has the head a and the body l. All elements of l are the root of valid proof trees as well and by the definition of the height function have a smaller height. By the induction hypothesis therefore all elements of l are in

M. Since r' is the result of applying a grounding to a rule from P, r' must be true in M. Therefore a is also in M.

In the database case, the root is an element of the database. Any element of the database must be in any model by definition. □

For the back direction, it suffices to show that the proof-theoretic semantics are a model as well by lemma 4.

Lemma 7 (`proofTheoreticSemanticsIsModel`)**.** *Let P be a program and d be a database. Then the proof-theoretic semantics of P and d is a model for P and d.*

Proof. As any element from the database d has a valid proof tree, the database is contained in the proof-theoretic semantics.

We need to prove that all rules from $\mathit{ground}(P)$ are true in the proof-theoretic semantics. Let r be such a rule and suppose that every element of $\mathit{body}(r)$ is in the proof-theoretic semantics. Therefore there exists a list of trees $t_1, \dots, t_n$ such that all trees are valid and the list $\mathit{root}(t_1), \dots, \mathit{root}(t_n)$ equals the body of r. Then we can build a new tree with the root $\mathit{head}(r)$ and the children $t_1 \dots t_n$. This tree is valid because all subtrees are valid and the root and its children are the instance of a ground rule of P. Therefore also $\mathit{head}(r)$ is in the proof-theoretic semantics and r is true. □

We expand a bit on the proof above. The lemma we use to get the valid proof tree for $\mathit{head}(r)$ is called `createProofTreeForRule`. We know that all elements in the body have valid proof trees and need to extract a list of these proof trees. We show in the lemma `getTree_Helper` a more general property from which this follows.

Lemma 8 (`getTree_Helper`)**.** *Let A and B be types. Let l be a list of type A and S be a finite set of type A. Let f be a function from B to A and valid a property of elements of type B. If all elements from l are in S and for any element a in S there exists an element b such that $f(b) = a$ and b is valid, then there exists a list l' such that mapping l' with f yields l and all elements of l' fulfill the $\mathit{isValid}$ predicate.*

Before proving the statement, we explain how we want to use this. A is supposed to be the ground atoms and B is supposed to be the proof trees. The function $f : B \to A$ is the root function, l is the body of the rule r and S is the finite set of the body elements. We use the set as we will use induction on l so that l will not always be the body of a rule in the proof. If we want to show that the head is in the proof-theoretic semantics, we already know that every atom a in the body has a valid proof tree whose root is a. Then we get the list of proof trees we can use as the children for the rule creation.

Proof of lemma 8. We prove this by structural induction on l. If l is empty then we can simply use the empty list for l' that fulfills the desired properties.

If l has the shape $\mathit{hd} :: \mathit{tl}$, then we can get a hd_b element for hd since hd in S. By the properties of S, hd_b is valid and is mapped to a via f. Any element in tl is still in S so that the induction hypothesis provides us a list tl' that maps via f to tl with only valid members. Then the list $hd_b :: tl'$ is the witness for l'. $\square$

This concludes the back direction and we finish the proof of the equivalence of the semantics and conclude this chapter.

Theorem 1 (`SemanticsEquivalence`)**.** *For any program P and database d the model-theoretic and the proof-theoretic semantics of P and d coincide.*

Chapter 4

Validating proof trees

After introducing the problem and modeling datalog in Lean, we now describe the algorithm to verify a solution partially. We focus in this section on the soundness by verifying a given proof tree. In the previous section, we introduced the following characterization for valid proof trees:

```
def isValid(P: program τ) (d: database τ) (t: proofTree τ):
Prop :=
match t with
| proofTree.node a l =>
( ∃(r: rule τ) (g:grounding τ),
    r ∈ P ∧
    ruleGrounding r g = groundRuleFromAtoms a (List.map root l)
    ∧ l.attach.All₂ (fun ⟨x, _h⟩ => isValid P d x))
∨ (l = [] ∧ d.contains a)
```

This is a disjunction so we need procedures to decide for both disjuncts if they are true. The second disjunct consists of a database check and a check of list emptiness. Both can be done straightforwardly.

The first part is more interesting. Since we use in the rule case existential quantifiers, this is not computable and we have to implement a function checking whether the rule case holds. While we can simply iterate over the program to check for the existence of a rule r, the number of groundings might be exponential or even infinite. Instead of using groundings, we want to use substitutions that we introduce in the next section.

4.1 Substitutions

A grounding is a function from variables to constants. This means we always need to specify for every variable a constant that it is mapped to. This was good in the definitions to ensure that we always get a ground atom, but raises problems in the unification case as the following example demonstrates.

Example 16. *Suppose we want to know whether we can unify the list of terms* $l_1 = [?x, ?y]$ *with the list of constants* $l_2 = [a, b]$*, i.e. we look for a function* $f : V \to C$ *such that mapping every element of* l_1 *with* f *yields* l_2*.*

We start by trying to unify x *and* a*. If we use groundings, we might use this function* $g = x \mapsto a, y \mapsto a, z \mapsto a$*.*

Now we want to use this result and match another term y *with the constant* b*. The variable* y *is already mapped to a different constant, but we cannot say whether this is due to a previous matching process or simply because we needed to define a value for every input.*

One solution might be to use a special constant symbol that we map variables to that are really unmapped for now. Instead, we want to use substitutions that were already introduced in [8]. A substitution is a partial mapping from variables to constants. We implement this by mapping to an Option of constant.

```
def substitution (τ: signature):= τ.vars → Option (τ.constants)
```

This allows us to only specify what is necessary. We call the substitution that maps any variable to none the empty substitution.

If we apply a substitution to a term and this term is a constant, we do not change the term. If the term is a variable and the substitution does not map the variable to none we replace the variable with the result of the substitution.

```
def applySubstitutionTerm (s: substitution τ) (t: term τ): term τ
    :=
match t with
| term.constant c => term.constant c
| term.variableDL v =>
    if p: Option.isSome (s v)
    then term.constant (Option.get (s v) p)
    else term.variableDL v
```

Similar to groundings we can extend this to atoms and rules.
(applySubstitutionAtom, applySubstitutionRule)

As we no longer replace all variables with constants, this only results in atoms instead of ground atoms.

The main result we want to prove is the following.

```
theorem groundingSubstitutionEquivalence
    (r: groundRule τ) (r': rule τ):
    (∃ (g: grounding τ), ruleGrounding r' g = r) ↔
    (∃ (s: substitution τ), applySubstitutionRule s r'= r)
```

This allows us to replace the grounding check with a substitution check when trying to validate trees and by this, we can bypass the problems that were illustrated in the example above.

For the forward direction, we want to find an equivalent substitution for every grounding. We can do this by mapping every variable to the same value of the grounding and placing this in the some constructor to obtain an element of the Option type.

```
def groundingToSubstitution (g: grounding τ): substitution τ
    := fun x => Option.some (g x)
```

We have to prove that these are equivalent. We defined in the previous section a coercion from ground atoms to atoms. We can also coerce constants to terms by the `term.constant` constructor which enables us to use an equality in the following lemma.

Lemma 9 (groundingToSubsitutionEquivTerm)**.** *Let t be a term and g be a grounding. Then the grounding of t by g is equal to the application of* `groundingToSubstitution` *g to t.*

Proof. If t is a constant, then both elements will simply return t.

If t is a variable v, then the term grounding by g will return $g(v)$. By the definition of `groundingToSubstitution` all variables v' return a some type which contains $g(v')$. Therefore the application replaces v by $g(v)$ and both results are equal. □

Using this result we can extend this equality to the atom level because in both cases only the terms change by the application of the functions of the previous lemma. Finally, we can extend this to the rule level. (groundingToSubsitutionEquivAtom, groundingToSubsitutionEquivRule)

For the back direction, we need an additional property illustrated by the following example.

Example 17. *Consider the program $\mathcal{P} = \{P \leftarrow, Q \leftarrow P\}$ and the signature $C = \emptyset$, $V = \{?x, ?y, ?z\}$ and $P = \{P, Q\}$*

Any rule in $\mathcal{P}$ is already a ground rule and there exists a substitution, the empty substitution that maps all variables to none, so that the rule is equal to itself as a ground rule.

There is however no grounding that can achieve this. We cannot define a grounding since we have no constant available but have variables that need to be mapped somewhere. Therefore the equivalence does not hold here.

For the equivalence to hold we always need at least one constant symbol. There exist at least two possibilities to achieve this in Lean. We could require that the type of constants is inhabited or that it is non-empty. We decided to use the first as the inhabited class offers directly a constant, the default value, whereas the class non-empty only proves that such an element exists and requires the axiom of choice to extract it.

```
def substitutionToGrounding [ex: Inhabited τ.constants]
(s: substitution τ): grounding τ :=
fun x =>
if p:Option.isSome (s x)
then Option.get (s x) p
else ex.default
```

Example 18 (continued)**.** *If we add the fresh symbol a to C, we can use the function $f : V \to C, v \mapsto a$. This will not cause any problems during the verification of a proof tree. While it increases the number of groundings it will not result in any match if used, because it does not occur in the proof tree.*

This is similar to the Herbrand base where we also add a constant symbol if no constant is present.

For the back direction, we prove again that it is equivalent on terms and then the atom and rule case easily follow. As applying a substitution to a term does not always yield a constant, we require that all variables of the term are mapped to a constant by the substitution. We formalize this requirement using the substitution domain as the set of defined variables of a substitution.

```
def substitution_domain (s: substitution τ): Set (τ.vars) :=
    {v | Option.isSome (s v) = true}
```

Lemma 10 (substitutionToGroundingEquivTerm)**.** *Let τ be a signature that contains at least one constant symbol. For any substitution s and term t such that the term variables of t are a subset of the substitution domain of s, also grounding t by substitutionToGrounding of s yields c.*

Proof. If t is a constant, then it must be equal to c and neither the grounding nor the substitution change the term.

If t is a variable v, then $s(v)$ must be c in order to be equal to c. Therefore $s(v)$ is defined and substitutionToGrounding replaces v by the same value, so that the equality holds as well. □

In the end, we obtain the desired theorem with the additional requirement on the set of constants. Note that for the back direction if the application of a substitution to a rule yields a ground rule, then all variables of the rule are in the substitution domain.
(applySubstitutionRuleIsGroundImplVarsSubsetDomain)

Theorem 2 (groundingSubstitutionEquivalence)**.** *Let τ be a signature that contains at least one constant symbol. Then for any τ rule r' and ground rule r we have that there exists a grounding g such that grounding r' by g yields r iff there exists a substitution s such that applying s to r' yields r.*

When introducing substitutions, we had the goal to only add what is needed to a substitution and usually, we want the smallest possible substitution. In order to formalize this, we want to define a partial order on substitutions, that is denoted by $\subseteq$.

A substitution s_1 is a subset of a substitution s_2 if both substitutions agree on the substitution domain of s_1. Outside of this s_1 is never defined, whereas s_2 might be, so we view s_1 as smaller.

```
def substitution_subs (s1 s2: substitution τ): Prop :=
∀ (v: τ.vars), v ∈ substitution_domain s1 → s1 v = s2 v
```

This relation is by defintion reflexive (`substitution_subs_refl`). It is also anti-symmetric (`substitution_subs_antisymm`): Let s_1 and s_2 be two substitutions with $s_1 \subseteq s_2$ and $s_2 \subseteq s_1$. In order to show that s_1 equals s_2 it suffices to show that they have the same result for every variable v due to the principle of function extensionality. We can do a case distinction on whether $s_1(v)$ is defined. If $s_1(v)$ is defined then v is in the substitution domain of s_1 and since $s_1 \subseteq s_2$ $s_2(v)$ must be the same value. If $s_1(v)$ is not defined, i.e. it maps to none, then also $s_2(v)$ must be not defined. If $s_2(v)$ would be defined, then also $s_1(v)$ would have to be defined as we know that $s_2 \subseteq s_1$ holds. In any case, the functions return the same value so that they are equal.

Lastly, it is also transitive (`substitution_subs_trans`), so it is an instance of a partial order.

4.2 Unification

We know that instead of finding a grounding, it suffices to find a substitution. Now we want to describe an algorithm that tells us whether the ground rule that is formed from a vertex of the proof tree and its children is in the ground program. For this, we take inspiration from the unification problem of first-order logic.

In the unification problem of first-order logic we are given a set of equations $s_1 = t_1 \ldots s_n = t_n$ between first-order terms and are required to present the most general unifier. A unifier μ is a function from variables to terms so that if we replace every variable v in s_i and t_i by $\mu(v)$ then all equations are fulfilled. A unifier μ is called the most general unifier if for every unifier σ there exists a function τ with $\sigma = \tau \circ \mu$.

Our problem is similar and is in general an instance of the following problem.

Definition 2. *Let A and B be types and $f : A \to \text{substitution } \tau \to B$ be a function. We call the following problem an instance of the unification problem.*

Input: *A substitution s and elements a, b of A and B.*

Question: *Does there exist a substitution s' with $s \subseteq s'$ and $f(a, s') = b$. If such an s' exists we call it a solution. We call s' a minimal solution if we have $s' \subseteq s^*$ for any solution s^*.*

An example would be to consider A as the type of terms and B as the type of constants. Then f is `applySubstitutionTerm`. Another instance would be with atoms and ground atoms. a is in general the normal object and b the ground object.

An algorithm to solve the first-order unification problem is the algorithm of Martelli and Montanari [27] and is depicted below.

This algorithm offers a good starting point for our algorithms, but we know that certain transformations cannot occur in the limited fragment form we operate in. Additionally, we want to output a substitution instead of just answering whether a substitution exists. It is sufficient to do it here, but will later be important. Instead of mapping all x to t as done there in step 5, we will add

Algorithm 1 Algorithm of Martelli and Montanari

```
while There exists some equation for which a transformation is possible do
    Pick this equation $e$ and do one of the following steps if applicable
    1. If $e$ is of the form $t = t$, then delete this equation from the set.
    2. If $e$ is of the form $f(t_1, .., t_n) = f(s_1, .., s_n)$, then delete $e$ and add $n$ new
       equations of the form $t_i = s_i$
    3. If $e$ is of the form $f(t_1, .., t_n) = g(s_1, .., s_m)$ with $g \neq f$, then stop and
       reject.
    4. If $e$ is of the form $f(t_1, .., t_n) = x$ for a variable $x$ and delete $e$ and add
       an equation with the swapped order to the set
    5. If $e$ is of the form $x = t$ for some variable $x$, then check if x occurs in t.
       If it does, then stop and reject. If not map all $x$ to $t$ in the set.
end while
```

$x \mapsto t$ to a substitution that is presented as an input. If a variable occurs on the left side, we will check whether it is already in the domain of the substitution and if so check if its current value is consistent with the right side. Steps 2 and 3 deal with function symbols that are not allowed in our language apart from constants, so that we can replace them by checking whether two constants are equal in the term case. For atoms or rules, we use something similar and split the problem into multiple smaller problems for terms or atoms.

Finally, as the one side of the equation is always a ground object there will never be a variable on this side, so we do not have to swap the equation as in step 4.

We will start by matching a term to a constant with the following algorithm.

```
def matchTerm (t: term τ)(c: τ.constants) (s: substitution τ):
Option (substitution τ) :=
match t with
| term.constant c' =>
    if c = c'
    then Option.some s
    else Option.none
| term.variableDL v =>
    if p:Option.isSome (s v)
    then
        if Option.get (s v) p = c
        then
            Option.some s
        else
            Option.none
    else
        extend s v c
```

We are given a term t, a constant c and a current substitution s and want to return the minimal solution.

This is done by case distinction. If t is a constant, then we either return s if t is equal to c or return none as two different constants can not be unified by a substitution. If t is variable, we check if t is in the domain of s. If it is already defined we check if the value matches the required value. If it is not defined we extend s with the new mapping $v \mapsto c$. Formally extend is defined in the following way:

```
def extend (s: substitution τ) (v: τ.vars) (c: τ.constants) :
    substitution τ
:= fun x => if x = v then Option.some c else s x
```

We now formally prove the correctness of this algorithm.

Lemma 11 (matchTermFindsSolution)**.** *Let t be a term, c be a constant and s be a substitution. If matchTerm t c s returns a substitution s', then s' is a solution.*

Proof. The proof is done via case distinction. Suppose firstly that t is a constant c'. Since matchTerm returned a substitution we must have that c and c' are the same constant and therefore s' is s. Applying a substitution to a constant does not change it, so $s't = s'(c') = c' = c$. Additionally since $\subseteq$ is a partial order and $s' = s$, we have that $s \subseteq s'$

Now we assume that t is a variable v. Now we do another case distinction on whether sv is defined or not. If it is defined, v must already be mapped to c and we return s as this is a solution as seen previously. If it would be mapped to something else, then matchTerm would return none, which would violate our assumptions. If it is not defined, we use extend. After that v is mapped to c, so that $s't$ will be equal to c. Now we finally have to show that $s \subseteq$ extend s v c. We only change the value of v. Since sv was not defined earlier and v is the only variable whose result was changed, s and extend s v c are equal on the substitution domain of s, so that it is fulfilled(s_subset_extend_s). $\square$

We have proven so far that matchTerm returns a solution, but it might not be a minimal solution. As we can transform any grounding into a substitution a match function might return the grounding in example 16. As we are interested in matching rules and ground rules, we will have to match many terms so that the minimality is an important property for later proofs.

Lemma 12 (matchTermFindsMinimalSolution)**.** *Let t be a term, c be a constant and s be a substitution. If matchTerm t c s returns a substitution s', then s' is a minimal solution.*

Proof. We already know that s' is a solution and only have to show its minimality. Let s^* be an arbitrary solution.

We prove the minimality again via case distinction on t. If t is constant, then s' must be equal to s. Since s^* is a solution we have that $s \subseteq s^*$ and since $s = s'$, we gain the desired result.

Now we consider the case of t being a variable v and do a case distinction whether sv is defined. If it was already defined, then s' must again be equal to s, so that the claim is fulfilled by the argument above. If sv was not defined, we have to show that extend $s\ v\ c$ is a subset of any such s^*. We assume for a contradiction that this is not the case. Then there must be a variable in the domain of extend $s\ v\ c$ such that extend $s\ v\ c$ and s^* differ. Suppose this variable is v. Then s^* would either not be defined for v or map v to some other constant c'. In both cases $s^*v \neq c$, so that s^* would not be a solution and we would have reached a contradiction. If it is some other variable v', then the value of extend $s\ v\ c$ is simply the value of s. Since s^* maps v to a different value compared to s, s would not be a subset of s^* and we have reached another contradiction. □

So we know that if matchTerm returns a substitution then it is a minimal solution. This holds however already if we never return a solution. We have to prove that if no substitution exists, then there is no solution.

Lemma 13 (matchTermNoneImpNoSolution)**.** *Let t be a term, c be a constant and s be a substitution. If matchTerm t c s returns none, then there exists no solution.*

Proof. This is again done via case distinction on the type of t. If t is a constant c', then c' must be different from c, so that matchTerm returns none. Then no substitution can map t to c.

If t is a variable v, then sv must be defined and mapped to a different value compared to c. Then again no such solution can exist. A solution would have to agree with s on the substitution domain of s, but then cannot map v to c. □

After proving the correctness for terms we now want to move up to atoms. Unfortunately, we cannot use recursion directly on the term list of an atom because an atom requires a proof that the length of the list is equal to the arity of the relation symbol, which fails when we do recursion on the list. Therefore we first establish a new procedure that matches a list of terms with a list of constants, if possible.

```
def matchTermList (s: substitution τ) (l1: List (term τ)) (l2:
    List (τ.constants)): Option (substitution τ) :=
    match l1 with
    | List.nil =>
        match l2 with
        | List.nil =>
        some s
        | List.cons _ _ =>
        none
    | List.cons hd tl =>
        match l2 with
        | List.nil => none
        | List.cons hd' tl' =>
        let s' := matchTerm hd hd' s
        if p: Option.isSome s'
        then matchTermList (Option.get s' p) tl tl'
        else none
```

Here we are given as previously a substitution as an input with the two lists. We see that if both lists differ in length then the function always returns none. If both lists are the same length we start matching the front and recursively call matchTermList with the resulting substitution until no substitution is found or the entire list is processed.

Lemma 14 (`matchTermListFindsSolution`)**.** *Let s be a substitution, l_1 a list of terms and l_2 a list of constants. If matchTermList s l_1 l_2 returns a substitution s', then s' is a solution.*

Proof. We prove this by induction on l_1 for arbitrary s and l_2. In the base case, l_1 is the empty list. Since matchTermList returns something, l_2 must also be the empty list. Then s' is equal to s. Applying this to an empty list returns an empty list and by reflexivity, we have that $s' \subseteq s$, so that s' is a solution.

In the induction step we have that l_1 is of the form $hd :: tl$ and we can similarly assume that l_2 is of the form $hd' :: tl'$ since matchTermList returned a substitution. Since matchTermList returned a substitution, matchTerm hd hd' also must return a substitution s^*.

We then use this as an input to gain s' from matchTermList s^* tl tl'. From the induction hypothesis, we know that $s^* \subseteq s'$ and applying s' to tl results in it being equal to tl' because s' is a solution to tl tl' and s^*. First, we have to show that $s \subseteq s'$. From the correctness proof of matchTerm, we know that $s \subseteq s^*$ and from the induction hypothesis we know that $s^* \subseteq s'$. Since $\subseteq$ is transitive, the result follows.

Secondly, we prove that s' is a solution for $hd :: tl$ and $hd' :: tl'$. From the induction hypothesis, we know that mapping tl with s' yields tl'. We know that s^* was a solution for hd and hd' and since $s^* \subseteq s'$, s' agrees on the domain of s^* with s^*. Therefore it is still a solution for hd and hd'. If hd was a constant, then any substitution would do. If hd was a variable, then it is mapped to the same element as by s^* so that applying s' to hd yields hd' as well, so that applying s' to $hd :: tl$ yields $hd' : tl'$. □

After proving that it is a solution, we prove that the solution is minimal.

Lemma 15 (`matchTermListFindsMinimalSolution`)**.** *Let s be a substitution, l_1 a list of terms and l_2 a list of constants. If matchTermList s l_1 l_2 returns a substitution s', then s' is a minimal solution.*

Proof. We prove this again via induction on l_1 for arbitrary l_2 and s. If l_1 is empty then l_2 is empty as well and we return s. The substitution s is the minimal solution as any solution s^* must fulfill $s \subseteq s^*$.

In the induction step we have that l_1 has the form $hd :: tl$ and we can assume that l_2 has the form $hd' :: tl'$ because a substitution was returned. Let s' be the result of *matchTermList* s l_1 l_2 . We have to show that for any solution s^*

for l_1, l_2 and s, we have that $s' \subseteq s^*$. Let s^* be a solution for l_1, l_2 and s. We obtain s' from *matchTermList matchTerm hd hd' s tl tl'*. By the induction hypothesis we have that s' is the minimal solution for any substitution $\bar{s}$ that maps tl to tl' and *matchTerm hd hd'* $s \subseteq \bar{s}$. As s^* is a solution for $hd :: tl$, $hd' :: tl'$ and s it must map tl to tl'. Additionally, must be a solution for hd, hd' and s and due to lemma 12 we have that *matchTerm hd hd'* $s \subseteq s^*$ because matchTerm returns the minimal solution. Therefore we have proven that s' is a subset of s^* and finish the proof. □

Finally, the negative case confirms that the return value of none does indeed state that there is no solution.

Lemma 16 (`matchTermListNoneImplNoSolution`)**.** *Let s be a substitution, l_1 a list of terms and l_2 a list of constants. If matchTermList s l_1 l_2 returns none, then there exists no solution s'.*

Proof. We prove this again via induction on l_1 for arbitrary l_2 and s. If l_1 is the empty list then l_2 must be non-empty and then no solution exists.

In the induction step we have that l_1 has the form $hd :: tl$. If l_2 is empty, the claim is proven as no substitution can be applied that yields two equal lists, so we assume that l_2 has the form $hd' :: tl'$. There are two cases where matchTermList can return none. The first case is when matchTerm hd hd' s returns none. If that is the case there is no s' with $s \subseteq s'$ and applying s to hd will make it equal to hd'. Therefore the two lists cannot be equal either and the proof is finished.

If matchTerm hd hd' s returns a substitution s^*, then matchTermList s^* tl tl' must return none. From the induction hypothesis, we know that there is no substitution s' with $s^* \subseteq s'$ and applying s' to tl bringing it equal to tl'. Since s^* is already the minimal solution to match hd with hd' there cannot exist a solution whose application will lead to hd and hd' being equal and tl and tl' being equal. □

This can be used to create a matchAtom procedure. We are given an atom and a ground atom and check if the symbols are equal. If they are, we use matchTermList to find a unifier. If the symbols are different, we will never unify them and can already return none.

```
def matchAtom (s: substitution τ) (a: atom τ) (ga: groundAtom τ):
Option (substitution τ) :=
    if a.symbol = ga.symbol
    then
        matchTermList s a.atom_terms ga.atom_terms
    else none
```

The correctness proofs follow from the proofs of matchTermList.

Now we can create a `matchAtomList` function similar to `matchTermList` and use this to define a `matchRule` function. We first try to match the head of the rule with the head of the ground rule. If this returns a solution, we use this as the start for matching the bodies. If not, we will not find a solution and return none.

```
def matchRule (r: rule τ) (gr: groundRule τ):
    Option (substitution τ):=
    let s := matchAtom emptySubstitution r.head gr.head
    if p: Option.isSome s
    then matchAtomList (Option.get s p) r.body gr.body
    else none
```

We now prove again that if this returns a substitution then this is a solution and if none is returned then there is no solution using our previous results. We want a statement with the quantifier to combine it with the theorem about the equivalence between groundings and substitutions. We can prove this statement by using the result of `matchRule` as a witness for the quantifier. For the back direction, we see that this statement implies that a solution exists, which we always find with `matchRule`.

Theorem 3 (`matchRuleIsSomeIffSolution`)**.** *Let r be a rule and gr be a ground rule. Then there exists a substitution s such that applying s to r yields gr iff `matchRule` returns some substitution.*

We now have a method to replace this quantifier with a computable function and can finally devise the check for tree validation.

4.3 Tree validation

We have so far introduced substitutions and have shown a way to decide whether a substitution exists that maps a rule to a ground rule. Now we want to finally show a way to validate trees. As the first step, we want to find whether a ground rule gr we gain from the tree is in the ground program, i.e. that there exists a grounding g (or by theorem 2 a substitution s) and a rule r from the program P so that applying g to r yields gr. We want to use the `matchRule` function defined earlier for this. We can simply pass each rule of the program given as a list of rules into this function and stop when `matchRule` returns a some object. This works but requires in the worst case to match every rule in the program with a given ground rule. Suppose that the relation symbol of the head of the given ground rule is T. We can safely discard any rule whose head is not an atom with the relation symbol T. This can be generalized into the symbol sequence which is a list of relation symbols that starts with the relation symbol of the head and then the relation symbols of the body atoms in the order the atoms occur in the body.

```
def symbolSequence (r: rule τ): List τ.relationSymbols :=
    r.head.symbol::(List.map atom.symbol r.body)
```

The equality of symbol sequences is a necessary condition for the existence of substitution that matches a rule with a ground rule, as the application of any substitution does not change the symbols and therefore not the symbol sequence. (`symbolSequenceNotEq`).

Now we could iterate over the whole program and calculate the symbol sequence for every rule and only start the matchRule process for those whose symbol sequences are equal to the given ground rule. In the worst case, we still have to iterate over the whole program. It would be even better to preprocess the program into a look-up structure that returns a list of rules for a given symbol sequence.

We use a function of the type `List τ.relationSymbols → List (rule τ)` as the look-up structure and build it iteratively by adding the first rule of the given program to the right symbol sequence element.

```
def parseProgramToSymbolSequenceMap (P: List (rule τ))
(m: List τ.relationSymbols → List (rule τ)):
List τ.relationSymbols → List (rule τ) :=
match P with
| [] => m
| hd::tl =>
    let seq:= symbolSequence hd
    parseProgramToSymbolSequenceMap tl
        (fun x => if x = seq then hd::(m x) else m x)
```

If we start this with a function that maps any symbol sequence to the empty list and gain a function m' then for any list of relation symbols l, $m'(l)$ returns exactly those rules in P whose symbol sequence matches l. This follows from the following lemma.

Lemma 17 (`parseProgramToSymbolSequenceMap_mem`). *Let P be a program and m be a function that maps a list of relation symbols to the list of rules. Let m' be the result of `parseProgramToSymbolSequenceMap P m`. Then for any list of relation symbols l and rule r, we have r is in $m'(l)$ iff r is in $m(l)$ or r is in P and the symbol sequence of r equals l.*

Proof. We prove this by induction on the structure of P for arbitrary m.

If P is empty, then $m' = m$ and no rule is a member of P so that the claim follows.

If P has the structure $hd :: tl$, then m' is the result of

```
parseProgramToSymbolSequenceMap tl (fun x => if x =
    (symbolSequence hd) then hd::(m x) else m x)
```

Applying the induction hypothesis we see that for any rule r and list l, r is in $m'(l)$ iff it is in tl and has the symbol sequence l or was in `(fun x => if x = (symbolSequence hd) then hd::(m x) else m x)`

The only element of P not yet considered was hd and is is either obtained by its symbol sequence or was already there in m so that the proof follows. □

Since at the start $m(l)$ has no member for any l, the desired property holds. Now we can use this to check whether a substitution exists. `List.any` checks whether any element in $m(symbolSequencegr.toRule)$ matches the ground rule. If that is the case, we simply return unit as a matching rule in P was found. If that is not the case, we return an error message. The Except type allows us to combine these two return types.

```
def checkRuleMatch (m: List τ.relationSymbols → List (rule τ))
    (gr: groundRule τ): Except String Unit :=
  if List.any (m (symbolSequence gr.toRule))
    (fun x => Option.isSome (matchRule x gr)) = true
  then Except.ok ()
  else Except.error ("No match for " ++ ToString.toString gr)
```

Lemma 18 (checkRuleMatchOkIffExistsRuleForGroundRule)**.** *Let P be a program, m be the result of* `parseProgramToSymbolSequenceMap P (fun _ => [])` *and gr a ground rule. Then* `checkRuleMatch m gr` *returns ok iff there exists a rule r in P and a grounding g such that grounding r with g yields gr.*

Proof. Let l be the symbol sequence of gr. If checkRuleMatch returns ok, then there exists a rule r in $m(l)$ there exists a grounding g such that grounding r by g yields gr. By lemma 17 this rule r is a member of P so the claim is proven.

If checkRuleMatch returns an error, then we have to show that for rule $r \in P$ and grounding g grounding r by g yields a different ground rule compared to gr. If r has the same symbol sequence as gr, then it occurred in $m(l)$ and since checkRuleMatch returned an error matchRule r gr must return none so that the property holds by the correctness of matchRule.

If r has a different symbol sequence to gr then this property holds as well due to symbolSequenceNotEq and theorem 2 because the grounding does not change the symbol sequence so that r can never be grounded to be equal to gr. □

This function is almost enough to complete the rule case. We just need to also verify all subtrees. For this we introduce the `List.map_except_unit` function that applies a function of the type `A → Except B Unit` to a list l until the first error occurs. If no error occurs, then we return Unit which is a type that has only one element () and is the placeholder because we have to return something.

```
def List.map_except_unit {A B: Type} (l: List A)
(f: A → Except B Unit): Except B Unit :=
match l with
| [] => Except.ok ()
| hd::tl =>
    match f hd with
    | Except.ok () => List.map_except_unit tl f
    | Except.error b => Except.error b
```

We can prove by induction that this function returns unit iff f returns unit on all elements of the list l (List.map_except_unitIsUnitIffAll).

Now we have the necessary tools to check whether a given proof tree is valid which is done by the `treeValidator`. For a tree with the root a and subtrees l, we do a case distinction whether l is empty. If a is additionally an element of the database, then we accept a because we are in the database case. If not we use checkRuleMatch to check for the rule case. If this returns ok, then we can already accept a because there are no subtrees.

If not we can only be in the rule case and first use checkRuleMatch and if this returns ok, then we check all subtrees using the treeValidator again called via `List.map_except_unit`.

```
def treeValidator (m: List τ.relationSymbols → List (rule τ))
    (d: database τ) (t: proofTree τ) : Except String Unit :=
  match t with
  | tree.node a l =>
    if l.isEmpty
    then  if d.contains a
          then Except.ok ()
          else
            match checkRuleMatch m {head:= a, body := List.map
    root l} with
            | Except.ok _ => Except.ok ()
            | Except.error msg => Except.error msg
    else
      match checkRuleMatch m {head:= a, body := List.map root l}
    with
      | Except.ok _ => List.map_except_unit l.attach (fun ⟨x, _h⟩
    => treeValidator m d x)
      | Except.error msg => Except.error msg
```

Using the previous lemmas and induction over the height of the tree for the recursive calls, we can prove the correctness of the treeValidator.

Theorem 4 (`treeValidatorOkIffIsValid`). *Let P be a program, m be the result of* `parseProgramToSymbolSequenceMap P (fun _ => [])` *and d be a database. For any proof tree t,* `treeValidator m d t` *returns ok iff t is valid for P and d.*

If we want to validate the whole result of a datalog program, we will often need many proof trees and a function that can validate them. Additionally, we need to preprocess the program to gain the map from symbol sequences to rules. We can reuse the `List.map_except_unit` function for this purpose

```
def validateTreeList (P: List (rule τ)) (d: database τ) (l: List
    (proofTree τ)) : Except String Unit :=
let m:= parseProgramToSymbolSequenceMap P (fun _ => [])
List.map_except_unit l (fun t => treeValidator m d t)
```

Using theorem 4 and `List.map_except_unitIsUnitIffAll`, we can show that this function returns ok iff all trees in l are valid. We know that a ground atom ga is in the proof-theoretic semantics if there exists a proof tree with the root ga so that we can conclude the set of roots of the trees in l is a subset of the semantics.

It turns out however that we can do a bit better and prove a stronger statement as the following example shows.

Example 19. *Consider the program P :*

$$\begin{aligned}
&A(?x,?z) \leftarrow B(?u,?x,?y,?z).\\
&B(?u,?x,?y,?z) \leftarrow C(?u,?x), D(?z,?y).\\
&C(a,b).\\
&D(c,d).
\end{aligned}$$

The result of this program over an empty database is

$$\{A(b,c), B(a,b,d,c), C(a,b), D(c,d)\}$$

and a proof tree is depicted below for each of them.

$$\begin{tikzcd}
D(c,d) & C(a,b) & & B(a,b,d,c) \arrow[dl, dash] \arrow[dr, dash] & & & & A(b,c) \arrow[d, dash] & \\
& & C(a,b) & & D(c,d) & & & B(a,b,d,c) \arrow[dl, dash] \arrow[dr, dash] & \\
& & & & & & C(a,b) & & D(c,d)
\end{tikzcd}$$

The proof tree for $A(b,d)$ already contains proofs for all the other elements of the program so it would be enough to check this single tree.

If we want to check a complete datalog result, we therefore do not have to include a proof tree for every element as they can occur in other proof trees already. We formalize this using the element member function that is true if this element is the root or a member in a subtree.

```
def elementMember (a: A) (t: tree A): Bool  :=
    match t with
    | tree.node a' l => (a=a') ∨
        List.any l.attach (fun ⟨x, _h⟩ => elementMember a x)
```

A tree is valid if all of its subtrees are valid. In the rule case, we explicitly required this, whereas there are no subtrees in the database case. Any element member is the root of some subtree and therefore any element member of a valid proof tree is in the proof-theoretic semantics
(`allTreeElementsOfValidTreeInSemantics`).

This allows us to establish the alternative property of validateTreeList. We added the requirement of all proofs being valid to prove the back direction. As the validity of a proof tree is in general not preserved under permutation, it is not sufficient to just require every element member of the trees to be in the proof-theoretic semantics.

Lemma 19 (`validateTreeListUnitIffSubsetSemanticsAndAllValid`)**.** *Let P be a program, d be a database and l a list of proof trees. Then validateTreeList returns ok iff all element members of the trees in l are in the proof-theoretic semantics and all trees in l are valid.*

Proof. For the forward direction, we already noted that if validateTreeList returns ok, then every tree in l must be valid. By the previous lemma therefore any element member of one of these trees is in the proof-theoretic semantics.

For the back-direction follows from the previous observation, that validateTreeList returns ok iff all trees in l are valid. □

Now it is enough to pass just the proof tree for $A(a, c)$ from our example into the checker to validate the whole input and use in general fewer trees.

Chapter 5

Validating graphs

In the previous section, we described an algorithm to verify proof trees. Proofs trees are a very natural way to encode certificates for datalog results but have some drawbacks.

Example 20. *Consider the following program that computes the transitive closure. We double an atom in the body which does not change the semantics of the rule.*

$$T(?x,?y) \leftarrow E(?x,?y). \tag{5.1}$$

$$T(?x,?z) \leftarrow T(?x,?y), T(?x,?y), E(?y,?z). \tag{5.2}$$

The database shall encode the elements of a chain $E(i, i+1)$ for $i \in \{0, n\}$. In the proof tree representation, we need the same proof tree twice for $T(?y, ?z)$ in the second rule and will need copies for it later, which results in large trees. The proof tree for $T(0, i+1)$ needs a vertex representing the head of eq. (5.2), one vertex for the E fact $E(i, i+1)$ and two proof trees for $T(0, i)$. Therefore the size of $T(0, i+1)$ in the number of nodes is more than double the size of $T(0, i)$ which leads to an exponential growth.

Proof trees offer no reusability of atoms as they need to stay in the tree shape. If an atom is used multiple times in the derivation we need for each usage a new vertex that is labeled by this atom. Formally we can view the proof tree as a directed tree $T = (V_T; E_T)$ with a label function $label : V_T \rightarrow groundAtom$ because different vertices can have the same label in a proof tree as in the example above. We need this as a ground atom a can be used in multiple ground rules in the derivation but cannot have connections from multiple vertices to the vertex representing a if we are in a directed tree.

A more compact representation would be a directed graph $G = (V_G, E_G)$ of T with $V_G = \{a \mid \exists v \in V_T, label(v) = a\}$ and $E_G = \{(a_1, a_2) \mid \exists v_1, v_2 \in V_T, label(v_i) = a_i \wedge (v_1, v_2) \in E_T\}$. Then a vertice can be considered equal to its label as no label occurs multiple times anymore. Examples of a proof tree and the corresponding graph are in fig. 5.1.

$$\begin{tikzcd}
 & & & T(0,3) \arrow[dll] \arrow[dr] \arrow[drr] & & \\
 & T(0,2) \arrow[dl] \arrow[d] \arrow[dr] & & & T(0,2) \arrow[dl] \arrow[d] \arrow[dr] & E(2,3) \\
T(0,1) \arrow[d] & T(0,1) \arrow[d] & E(1,2) & T(0,1) \arrow[d] & T(0,1) \arrow[d] & E(1,2) \\
E(0,1) & E(0,1) & & E(0,1) & E(0,1) &
\end{tikzcd}$$

(a) A proof tree for $T(0,3)$

$$\begin{tikzcd}
 & T(0,3) \arrow[dl, bend right] \arrow[dl, bend left] \arrow[dr] & \\
T(0,2) \arrow[d, bend right] \arrow[d, bend left] \arrow[drr] & & E(2,3) \\
T(0,1) \arrow[d] & & E(1,2) \\
E(0,1) & &
\end{tikzcd}$$

(b) A proof graph for $T(0,3)$

Figure 5.1: An example of a proof tree and a proof graph for the same fact from example 20

Another advantage occurs when we want to validate multiple trees or even a complete result. Multiple trees will often share labels as well and combining them into a graph, i.e. the union of G_{T_i} for trees T_i will again include every label just once. In the previous section, we introduced the possibility of not having to check every proof tree as they occur in other proof trees as well. Selecting a minimal amount of proof trees to check a complete result is however still an instance of the NP-hard subset cover problem.

We require for this that the successors of a ground atom are the same elements in every tree if the ground atom occurred in this tree. If not an atom would have more successors in the graph than previously in the trees and there will probably exist no matching ground rule. In practice, this requirement was always fulfilled.

Another requirement is acyclicity. We cannot use an atom to explain itself. Either we could have used it directly before or it cannot be explained by a proof tree at all.

5.1 Graph model

As a number of different results from mathematics are already formalized in Mathlib, it is no surprise that graphs are already modeled in Mathlib as depicted below:

```
structure SimpleGraph (V : Type u) where
  /-- The adjacency relation of a simple graph. -/
```

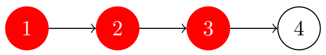


Figure 5.2: An example of an incomplete pregraph for example 21

```
  Adj : V → V → Prop
  symm : Symmetric Adj
  loopless : Irreflexive Adj
```

This graph representation is unfortunately not what we need in an algorithm. Firstly, the adjacency relation of this graph is always symmetric so it does not encode the directed graphs we need. The direction of the edges allows us to differentiate between the ground atoms that were used to derive a ground atom and the ground atoms that were derived from a ground atom. Secondly, the adjacency relation is a function to `Prop` and thus in general not computable but we want to use it in an algorithm. Finally, we need the successors of a vertex to verify the vertex as they are supposed to represent the body of a ground rule. When we want to obtain the successors in this graph model we would have to query the adjacency relation for every vertex assuming it would be computable.

Therefore we design our own implementation of a graph. We use an adjacency list approach so that we can quickly get all the successors of a vertex. The graph is represented by a hash map whose keys are the vertices. Each vertex is mapped via the hash map towards its successors.

```
abbrev PreGraph (A: Type) [DecidableEq A] [Hashable A] :=
    Std.HashMap A (List A)

def vertices (g : PreGraph A) : List A := g.toList.map Prod.fst
def successors (g : PreGraph A) (a : A) : List A := g.findD a []
```

`Std.HashMap.toList` returns here a list of pairs of the type $(A, List\ A)$, i.e. a vertex and its successors. Using `Prod.fst` we obtain the first element of every pair in the list, which are all vertices. `Std.HashMap.findD a []` returns the value saved with a in the hash map which are the successors of a. If a vertex is not present (the key is not in the hash map), then [] is returned which is the fallback option provided to `Std.HashMap.findD`.

This approach contains all important elements in a graph, but there is a small technical problem with it.

Example 21. *Consider the graph depicted in fig. 5.2. An element is colored, if it is in the list of vertices. Hence, the vertices are the list* $[1, 2, 3]$. *The vertex 3 has the element 4 as a successor, but 4 is not a vertex of the graph. Such a behavior is undesirable as searching for vertices that satisfy some criteria may lead outside of the graph.*

We define a pregraph to be complete if every successor of a vertex is a vertex itself and thus can avoid the problem.

```
def complete (pg: PreGraph A) :=
∀ (a:A), pg.contains a →  ∀ (a':A), a' ∈ (pg.successors a) →
    pg.contains a'
```

A graph is then a complete pre-graph and we can lift all previous operations to this new type.

```
abbrev Graph (A: Type) [DecidableEq A] [Hashable A] :=
    { pg : PreGraph A // pg.complete }
```

The completeness predicate was necessary because not all elements of the type A are vertices but the successors of a vertex can be any element of the type A. We start by formalizing walks in a graph. Walks allow the repetition of vertices in itself in contrast to paths which suffices our needs.

A walk w in a graph G is a list of vertices that are all vertices of G and are connected via the successor relation of G, i.e. for every i with $0 < i < i.length$ we have that $w[i-1] \in G.successors\ w[i]$. The starting vertex of the walk is consequently at the back.

```
def isWalk (l: List A) (G: Graph A): Prop :=
    (∀ (a:A), a ∈ l → a ∈ G.vertices ) ∧
    ∀ (i: ℕ), i > 0 → ∀ (g: i < l.length), l.get (Fin.mk i.pred
    (pred_lt i l.length g)) ∈ G.successors (l.get (Fin.mk i g) )
```

The `pred` function returns the predecessor of any natural number that is not zero and zero otherwise. Because we use the `List.get` function we not only have to specify a position but also a proof that this position is smaller than the length of the list to not get an error. These two elements are combined into a Fin type. In the future, we usually omit these proofs in this work because of the length. The proofs can however be found in the actual formalization.

Due to the completeness predicate it would suffice to require that the starting vertex is in G but this variant is simpler to state. We also allow the empty walk [] as this will be beneficial in later applications.

Lists are better in this use case in contrast to arrays because they can easily be extended at the front which we will use when exploring a graph algorithmically. We can always extend a walk by a successor of the leading element.

Lemma 20 (`isWalk_extends_successors`)**.** *Let w be a list of vertices and a and b be vertices. If $a :: w$ is a walk in G and b is a successor of a in G then $b :: a :: w$ is also a walk in G.*

A cycle is supposed to be a walk that starts and ends with the same node and an acyclic graph is a graph that has no cycles. Any list that only contains a single vertex of the graph would consequently be a cycle. Hence, we require that a cycle has additionally at a minimal length of two.

```
def isCycle (l: List A) (G: Graph A): Prop :=
  if h: l.length < 2
  then False
  else
  isWalk l G ∧ l.get (Fin.mk 0 _) = l.get (Fin.mk l.length.pred
    _)

def isAcyclic (G: Graph A) := ∀ (l: List A), ¬ isCycle l G
```

5.2 Validation of a graph

After defining the graph model, we want to specify how the graph validation takes place. For trees, we defined the recursive predicate `isValid` and implemented it using `treeValidator`. As the graph we receive may not even be acyclic we cannot simply define a recursive predicate as this will then not terminate. Instead, we define a local predicate for each vertex and its successor. Then we can check every vertex individually.

```
def locallyValid (P: program τ) (d: database τ) (v: groundAtom τ
    ) (G: Graph (groundAtom τ)): Prop :=
 (∃(r: rule τ) (g:grounding τ), r ∈ P
    ∧ ruleGrounding r g = {head:= v, body:= (G.successors v) })
 ∨ ((G.successors v) = [] ∧ d.contains v)
```

Except for the rule case, this is very similar to `isValid`. Therefore we can reuse earlier concepts and build a checker for this. We expect the program again parsed into a look-up structure m, a database d, the vertex a and its successors passed as a list l.

```
def localValidityCheck (m: List τ.relationSymbols → List (rule τ
    )) (d: database τ) (l: List (groundAtom τ)) (a: groundAtom τ
    ) : Except String Unit :=
  if l.isEmpty
  then
    if d.contains a
    then Except.ok ()
    else checkRuleMatch m (groundRule.mk a l)
  else
    checkRuleMatch m (groundRule.mk a l)
```

The correctness is proven similar to before assuming that m is the result of `parseProgramToSymbolSequenceMap` of the program P.

Lemma 21 (`localValidityCheckUnitIffLocallyValid`)**.** *Let d be a database and P be a program. If m is the result of `parseProgramToSymbolSequenceMap` with a function that maps any element to the empty list and l is the list of successors of a ground atom a. Then the `localValidityCheck` returns ok iff a is locally valid with respect to P and d.*

In the next section, we will show a method to show that a graph is acyclic and all vertices are locally valid with respect to P and d. In the remainder of this section, we assume that this is true and devise a method to show that then all vertices of the graph are in the proof-theoretic semantics of P and d.

Our goal is to create a valid proof tree for every vertex. We devise a function that creates such a proof tree for a given vertex by recursively calling it on the successors.

```
def extractTree (a: A) (G: Graph A) (mem: a ∈ G.vertices)
    (acyclic: isAcyclic G): tree A :=
  tree.node a (List.map (fun ⟨x, _h⟩ => extractTree x G
    (G.complete a mem x _h) acyclic) (G.successors a).attach)
```

This function only terminates because the graph is acyclic. Therefore we require the acyclicity as an explicit argument in the function. We will show the termination by using the number of vertices that are reachable from the current vertex. These decrease the further we walk in the graph or else we would be able to reach a previous node and have a cycle which is not allowed. We start by formalizing the notion of reachability. We say that a vertex a `canReach` a vertex b if there exists a walk w from a to b. (The start of the walk is the last element in the list).

```
def canReach (a b: A) (G: Graph A):=
∃ (p: List A) (neq: p ≠ []),
    isWalk p G ∧ p.get (Fin.mk 0 _) = b
    ∧ p.get (Fin.mk p.length.pred _) = a
```

Any vertex a can reach itself via the walk $[a]$(`canReach_refl`).

We want to use the finite set of the vertices that are reachable from the current node so that we can argue that the cardinality of this set decreases and by this, the function terminates. This set will be a subset of the vertices of the graph G since any element reachable by a walk in G must be in the vertices of G. Ideally, we want to filter this finite set to only keep those vertices reachable from the current node. Unfortunately, `Finset.filter` requires a decidable predicate. This predicate is in principle decidable for example by Dijkstra's algorithm. Instead of implementing and validating it, we instead use classical logic to obtain that any predicate is decidable and reuse the previous filter version. This is now no longer computable but we only want to use it in the termination proof anyway.

```
noncomputable def Finset.filter_nc (p: A → Prop) (S: Finset A):=
    @Finset.filter A p (Classical.decPred p) S

lemma Finset.mem_filter_nc (a:A) (p: A → Prop) (S: Finset A): a ∈
     Finset.filter_nc p S ↔ p a ∧ a ∈ S
```

The global successors in a graph G of a vertex a are then all the vertices b that are reachable from a in G.

```
noncomputable def globalSuccessors (a:A) (G: Graph A): Finset A
    := Finset.filter_nc (fun b => canReach a b G)
    G.vertices.toFinset
```

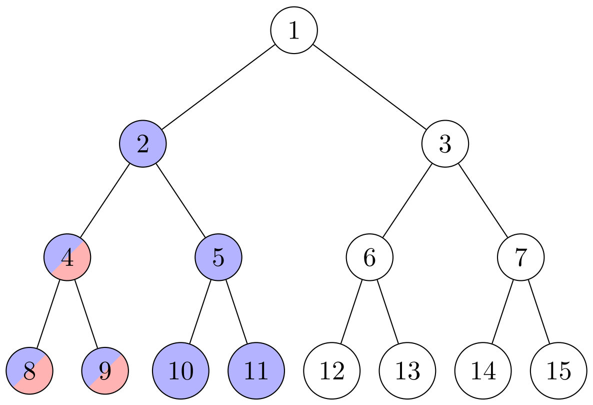


Figure 5.3: The global successors of 2 are colored blue, the global successors of 4 are colored red.

We already see in fig. 5.3 that the global successors of 2 are a strict subset of the global successors of 4 and 4 is the successor of 2. We can generalize this. In any graph we have that for every vertex a and successor b of a we have that $globalSuccessors(b) \subseteq globalSuccessors(a)$, because any element c that b can reach can also be reached from a by first going to b and then following the walk from b to c
(globalSuccessorsSubsetWhenSuccessor).

If the graph is additionally acyclic, we get that $globalSuccessors(b) \subset globalSuccessors(a)$, because a can reach b via the successor edge. If b could reach a, then this would lead to a cycle which is not allowed.
(globalSuccessorsSSubsetWhenAcyclicAndSuccessor)

Since we call extractTree on all successors b of a and the global successors of b are a strict subset of the global successors of a the cardinality of the global successors of the vertex we call the function on always decreases.

```
def extractTree (a: A) (G: Graph A) (mem: a ∈ G.vertices)
    (acyclic: isAcyclic G): tree A :=
  tree.node a (List.map (fun ⟨x, _h⟩ => extractTree x G
    (G.complete a mem x _h) acyclic) (G.successors a).attach)
termination_by Finset.card (globalSuccessors a G)
decreasing_by
  simp_wf
  apply Finset.card_lt_card
  apply globalSuccessorsSSubsetWhenAcyclicAndSuccessor
  apply acyclic
  apply _h
  apply mem
```

We see from the definition that the root of the tree returned by extractTree is always the vertex a used in the arguments. (rootOfExtractTree)

It remains to be shown this leads to a valid proof tree.

Lemma 22 (`extractTreeValidIffAllLocallyValidAndAcyclic`)**.** *Let G be an acyclic graph where all vertices are locally valid with respect to a program P and database d. Then extractTree results in a valid proof tree for any vertex a of G.*

Proof. We prove this by strong induction on the cardinality of the global successors of a.

Since a is a vertex of G, a is locally valid. There are two cases for this. If a is locally valid because it has no successors and is in the database, then the map operation creates no subtrees so that the resulting proof tree is valid as well.

If a is locally valid because a and its successors form a ground rule from the ground program of P, then we create the subtrees with extractTree. By the previous observation, the roots of the subtrees equal the successors of a, so that we have again a ground rule from the ground program. The global successors of any successor of a are a subset of the global successors of a and thus their cardinalities are smaller. Therefore we can apply the induction hypothesis and see that all these trees are then valid as well, so that the proof tree for a is valid. □

Using this result the vertices of an acyclic graph whose vertices are all locally valid with respect to P and d are a subset of the proof-theoretic semantics of P and d.

5.3 Depth-first search

By the results of the previous result, we know that in order to verify a graph we have to call the `localValidityCheck` on every vertex and its successors and determine whether the graph is acyclic. A typical solution to answer the second question is the depth-first search, which we will implement and verify in this section. During the depth-first search, the algorithm has to visit every vertex. With the aim of reducing the number of iterations over the vertex list, we combine this with executing `localValidityCheck` on every vertex and its successors.

We generalize this by considering a function `f: A → List A → Except String Unit` for a `Graph A`. The Except type is again chosen to return an error message to the user in the negative case.

We follow [16] when implementing depth-first search and use several helper functions that simplify the proofs. A first function `dfs` starts the process on every vertex and the actual exploration of the graph is done in `dfs_step`. So that we do not explore vertices multiple times, we store already explored vertices in a finite set. For performance reasons, this is a hash set instead of the finite set from Mathlib.

The function `isOkOrMessage` takes an arbitrary exception whose error type is a string and transforms it into an `Except String Unit` object to have a similar output type as the treeValidator. This returns ok iff the original exception was also an ok element(isOkOrMessageOkIffExceptionIsOk).

We use foldl_except_set as a variant as the common *foldl* or *reduce* function. It stops and returns the first found exception. If no exception is found it executes the function on the first element with the input set and then calls itself recursively with the resulting sets. We use this to share the set of already explored vertices in further explorations.

```
def isOkOrMessage (e: Except String A): Except String Unit :=
    match e with
    | Except.error msg => Except.error msg
    | Except.ok _ => Except.ok ()

def foldl_except_set
(f: A → HashSet B → (Except String (HashSet B)))
(l: List A) (init: HashSet B): Except String (HashSet B) :=
  match l with
  | [] => Except.ok init
  | hd::tl =>
    match f hd init with
    | Except.error msg => Except.error msg
    | Except.ok S => foldl_except_set f tl S

def dfs (G: Graph A) (f: A → List A → Except String Unit) :
    Except String Unit :=
    isOkOrMessage (foldl_except_set (fun ⟨x,_h⟩ S => dfs_step x G
    f [] (isWalkSingleton G x _h) (List.not_mem_nil x) S)
    G.vertices.attach HashSet.empty )
```

The main work is done in dfs_step which we define next. This function takes a variety of arguments:

1. an element a that is the vertex we want to start exploring from,
2. a graph G,
3. a function `f: A → List A → Except String Unit` that shall be evaluated on every vertex and its successors,
4. a list $currWalk$ of nodes that are the walk we used to arrive at a when exploring the graph,
5. a proof that $a :: currWalk$ are indeed a walk in G,
6. a second proof that a is not in $currWalk$ for the termination proof of this function and
7. a hash set *visited* of already explored vertices to allow earlier termination.

We start the algorithm by checking if a is already in *visited*. If that is the case, then we have no further exploration to do and return *visited*. If not we check if f raises an error on a and its successors and return any found error. If f returns ok, we check for a cycle by intersecting a's successors with the current walk. If any successor of a occurs in the current walk, we have found a cycle and return this as an error. If we find no cycle, then we recursively call `dfs_step` using `foldl_except_set` to update the set of visited vertices on every successor. If no error is found during this exploration, we add a using `addElementIfOk` to the returned set by `foldl_except_set` before returning it.

```
def addElementIfOk [Hashable A] (e: Except B (HashSet A)) (a:A):
    Except B (HashSet A) :=
  match e with
  | Except.ok S => Except.ok (S.insert a)
  | Except.error msg => Except.error msg

def dfs_step [Hashable A] (a: A) (G: Graph A)
(f: A → List A → Except String Unit) (currWalk: List A)
(walk: isWalk (a::currWalk) G) (not_mem: ¬ (a ∈ currWalk))
(visited: HashSet A) : Except String (HashSet A) :=
  if visited.contains a
  then Except.ok visited
  else
    match f a (G.successors a) with
    | Except.error msg => Except.error msg
    | Except.ok _ =>
      if succ_walk: (G.successors a) ∩ (a::currWalk) = []
      then

      addElementIfOk (foldl_except_set (fun ⟨x, _h⟩ S =>
        dfs_step x G f (a::currWalk)
        (isWalk_extends_successors walk x _h)
        (not_mem_of_empty_intersection succ_walk x _h) S)
        (G.successors a).attach visited) a
      else
        Except.error "Cycle detected"
```

This procedure terminates because we call `dfs_step` only on elements that do not occur on the current walk. Formally, we show that the cardinality of the finite set `(List.toFinset G.vertices \ List.toFinset currWalk)` decreases. Since any element of a walk is a vertex, the cardinality of `List.toFinset currWalk` is smaller than the cardinality of the set of vertices. In the recursive calls, the walk is instead $a :: currWalk$ and since a is not in $currWalk$ the cardinality of `List.toFinset a::currWalk` is larger than the cardinality of `List.toFinset currWalk`. The graph stays the same and hence the overall cardinality decreases.

The desired theorem describing the behavior of depth-first search is the

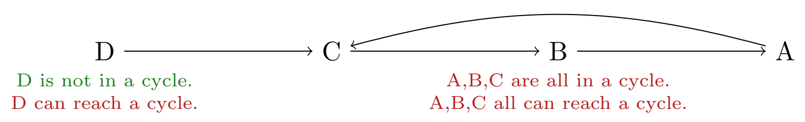


Figure 5.4: Propagating Acyclicity Check Results via Depth-First Search in example 22.

following:

Theorem 5 (`dfs_semantics`). *For any graph G and function* `f: A → List A → Except String Unit`*,* `dfs` *G f returns ok iff the graph is acyclic and f is evaluated to ok for every vertex a of G and its successors.*

We defined acyclicity by stating that no list of elements of type A is a cycle in G but we do not check all lists for being a cycle in G. Therefore we need another criteria for acyclicity that works on the vertex level. A first try is the membership in a cycle. A graph is obviously acyclic if no vertex is a member in a cycle. This is however not sufficient as the next example shows.

Example 22. *Suppose that we start* `dfs_step` *on the vertex D on the graph depicted in fig. 5.4. We will start exploring D, then C, B and A. During the exploration of A we notice that its successor C already occurs in the current walk. Hence, we found a cycle. If we use membership in a cycle, then this information is not propagated back to D.* `dfs_step` *returned an error for D while D is not a member in a cycle. Instead, we desire an iff relation between our criterion and the output of* `dfs_step`*. An alternative is the ability to reach a cycle. This is propagated back and a criterion for when* `dfs_step` *returns ok, which we will formalize now.*

We reuse the `canReach` predicate for the new `reachesCycle` predicate. A vertex a reaches a cycle, if there exists a cycle c with a member b that a can reach.

```
def reachesCycle (a:A) (G: Graph A):=
∃ (c: List A), isCycle c G ∧ ∃ (b: A), b ∈ c ∧ canReach a b G
```

We see from the definitions that a graph G is acyclic iff every vertex in G does not reach a cycle.

Lemma 23 (`acyclicIffAllNotReachCycle`). *A graph G is acyclic iff all vertices of G do not reach a cycle.*

Proof. If G is acyclic, then showing that all vertices of G are not reached from a cycle is equivalent to showing that any cycle in G cannot reach any element. Due to the acyclicity, we know that no cycles exist in G so the first direction is shown.

The back direction is proved via contradiction. Assuming that the graph is not acyclic, we know that there must exist a cycle c in G. Cycles in G are nonempty lists of vertices that are all in G. As any vertex can reach itself, there is a vertex reached from a cycle in contrast to our assumption, so we have reached the contradiction. □

This property is propagated back from the successors.

Lemma 24 (`NotreachesCycleIffSuccessorsNotReachCycle`). *A vertex a does not reach a cycle iff every successor of a does not reach a cycle.*

Proof. Both directions are proven via the contraposition. For the forward direction, we have that there is a successor b of a that reaches a cycle. Then we can simply extend the walk by adding a at the back and then a can reach a cycle.

For the backward direction, we assume that a reaches a cycle by a walk w and try to show that one of its successors must also be able to reach a cycle. If a reaches a cycle c with an element b we consider two cases. If b would be a successor of a, we have shown our goal.

Now we assume that b is not a successor of a. Again we can consider two cases. If the walk w is of length one then a and b must be equal and a is a member in a cycle itself. As long as a is not the first element in the cycle, we can simply pick the preceding element in the cycle due to the connectedness property of the walk. This does not work if a is the first element of the cycle, but since it is a cycle a must also be the last element and we can pick the predecessor of the last element, which is a successor of a and in a cycle.

If a reaches b with a walk w longer than length 1, then the walk w must have at least three elements since b is not a successor of a. Therefore the walk must have the form $b, \dots, p, a$ for a successor p of a. Then also p reaches a cycle. □

We detect a cycle if a successor b of the current vertex a already occurs in the current walk or if it is equal to a. This tells us that we can reach a node from itself by a path that has at least the length two. Indeed, the walk must contain a vertex from a or the current path which has length one and we can add b to it to obtain the cycle. Formally, we use the following function that takes an element, a list and a proof that the element is a member of the list. It returns the part of the list until we find the required element which will be the last element of the list. Since the list has a member the empty case can never occur.

```
def getSubListToMember (l: List A) (a: A) (mem: a ∈ l): List A :=
match l with
| [] =>
  have h: False :=
  by
    simp at mem

  False.elim h
| hd::tl =>
  if p: a = hd
  then [hd]
  else
    have mem': a ∈ tl :=
    by
      simp[p] at mem
      apply mem
    hd::getSubListToMember tl a mem'
```

We keep the starting element of the list (`getSubListToMemberPreservesFront`) and also preserve the walk property (`getSubListToMemberPreservesWalk`) since we keep the elements in their exact order and only remove the end. The list also ends with the required element (`getSubListToMemberEndsWithElement`).

Lemma 25 (`frontRepetitionInWalkImpliesCycle`). *Let a be a vertex and l be a list of vertices such that $a :: l$ is a walk in a graph G. If a is a member of l, then the graph G contains a cycle.*

Proof. We use `getSubListToMember` on l with the element a. This returns a walk in G that ends with a and starts with an element of which a is a successor. Hence, `(a::(getSubListToMember l a mem))` is again a walk that starts and ends with a. It also has a length of at least two since getSubListToMember always contains at least one element and we add a to the front. Therefore it is a cycle in G. □

As we also evaluate the function f during the depth-first search, we need a similar criterion that propagates the results of f back as we might encounter later a vertex where f is not evaluated to ok even if the starting vertex is evaluated to ok similar to example 22.

We reuse the reachability predicate again to state that all vertices reachable from a vertice fulfill f.

Lemma 26 (`allTrueIfAllCanReachTrue`). *Consider a function `f: A → List A → Except String Unit` and a graph G. Any vertex a in G is evaluated with its successors on f to ok iff any vertex b reachable in G from a vertex a in G is evaluated with its successors to ok.*

Proof. The forward direction holds because any vertex reachable in G must be a vertex of G itself. The backward direction holds because any vertex can reach themselves. □

The propagation can be similarly stated but requires the vertex themselves to also be evaluated to ok. The proof follows from the recursive view on reachability.

Lemma 27 (`canReachLemma`). *Consider a graph G and a function `f: A → List A → Except String Unit`. Any vertex reachable from a vertex a in G is evaluated with its successors on f to ok iff a is evaluated with its successors on f to ok and any vertex reachable from a successor of a is evaluated to ok with its successors on b.*

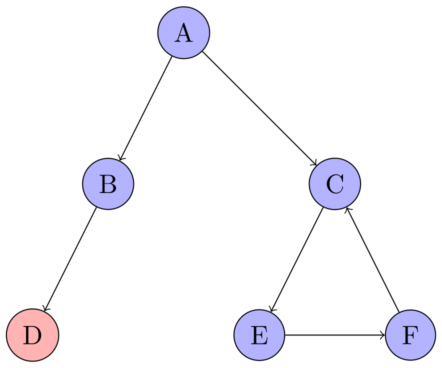


Figure 5.5: A graph for depth-first search. The result of f on a vertex and its successors is denoted by the color. Red denotes an error and blue denotes ok.

After establishing these criteria, we want to return to proving the correctness of the depth-first search algorithm. We start by proving the correctness of `dfs_step`. This function has a lot of arguments which we will call just as $\vec{v}$ and if needed refer to an argument by its name.

Ideally, we would want to prove the following statement: Let a be the vertex in $\vec{v}$ and G be the graph. Then `dfs_step` $\vec{v}$ returns ok iff a does not reach a cycle in G and any vertex that a reaches in G is evaluated with its successors to ok. This statement is not true because we can use the visited set to trick the algorithm.

Example 23. *Suppose we start `dfs_step` on the vertex A in the graph depicted in fig. 5.5. Additionally, suppose that the set of visited vertices is $\{B, C, D, E, F\}$.*

As A is not yet visited, we start the exploration process by evaluating f on A and its successors and as seen by the color this evaluates to ok. We have not seen any of the vertices in our walk and therefore call `dfs_step` on B and C and it returns ok since B and C are already in the set of visited vertices. Therefore the function in total returns ok for A.

This is however not what we expect it to do. A can reach the vertex D on which f evaluates to an error and it can reach the cycle $[C, E, F, C]$. These wrong evaluations occurred because we assumed in the implementation that the visited set reports this property truthfully. Therefore we require this in every proof. If we run this with an empty set of already visited vertices, then we gain the expected result.

Definition 3. *Let a be a vertex in a graph G and `f: A → List A → Except String Unit` be a function. We say that a has the DfsStepSemantics property if a does not reach a cycle in G and every vertex reachable from a in G evaluates to ok on f with its successors.*

This statement allows us to state the theorem we want to prove for `dfs_step` more succinctly.

Theorem 6 (`dfs_step_sematics`)**.** *Let $\vec{v}$ be the input arguments with the vertex a and visited set V. If every element in V has the DfsStepSemantics property, then `dfs_step` $\vec{v}$ returns ok iff a has the DfsStepSemantics property.*

We will prove this statement by induction on the number of vertices not covered by the current walk, i.e. `(List.toFinset G.vertices \ List.toFinset currWalk)`, for arbitrary walks and sets of visited vertices. We note that the base case is always true, because if this is equal to zero, then every vertex of G is in the current walk. In $\vec{v}$ we have however explicit proofs that a is a vertex in G and that a is not in the current path, so we have a contradiction. The interesting case is therefore only the induction step. There we need a criteria for when `foldl_except_set` returns an error.

In general, this seems hard to do. Here we already see that the set of visited vertices plays no really important role in the result of the statement as long as all elements in it have the DfsStepSemantics property. As long as we preserve this property in creating the sets we can view `foldl_except_set` as individual calls when considering whether it returns an error or not.

Lemma 28 (`foldl_except_set_is_ok`)**.** *Let l be a list of type A and S be a hash set of type B. Consider a property $p : B \to$ `Prop` and a function `f: A →  HashSet B → (Except String (HashSet B))`, that fulfills the following two properties. Firstly, that f preserves p, i.e. if every element in S has the property p and f returns for the input a and S a set S', then also every element in S' has the property p.*

Secondly, that f's acceptance status does not change on sets whose elements fulfill the property p, i.e. for any hash sets S_1, S_2 such that all elements in each hash set fulfill the property p, we have that for any element a f a S_1 returns ok iff f a S_2 returns ok.

Then there exists a hashset S' that `foldl_except_set` f l S returns S' iff $fa'S$ returns ok for any element a' in l.

Proof. We prove this by induction on the structure of l for arbitrary S.

If l is empty `foldl_except_set` always returns ok and there are no elements in l so that the claim holds.

If l has the shape $hd :: tl$, then $fhdinit$ must return a set S. If not then `foldl_except_set` returns an error and there is an element a from $l = hd :: tl$ for which $fainit$ is false so that both sides are false which finishes the proof.

The result of `foldl_except_set` is therefore `foldl_except_set f tl S`. From the induction hypothesis, we know that then any element a from tl we have that faS returns ok iff `foldl_except_set` returns ok. We already have the required result for hd, it remains to show that always f a $init$ returns ok. Since f preserves p and any element in $init$ has the property p, also any element in S has the property p. Therefore we can use the second requirement for f to conclude that f a $init$ returns true for any element a from $hd :: tl$. □

The property in our case is the DfsStepSemantics property. The first requirement will follow from the induction hypothesis. Therefore it remains to show that `dfs_step` preserves the DfsStepSemantics property.

Lemma 29. *[`dfs_step_preserves_notReachesCycleAndCounterExample`] Let $\vec{v}$ be the input arguments with the vertex a and visited set V. If every element in V has the DfsStepSemantics property and `dfs_step` $\vec{v}$ returns a set V', then every element in V' has the DfsStepSemantics property.*

Proof. We will prove this statement by induction on the number of vertices not covered by the current walk.

There are two possibilities for `dfs_step` to return a set. Firstly, a may be a member of V. Then we return V and the claim follows from the assumption. Now we consider the possibility of a not being a member of V. Then f must return ok for a and its successors and we evaluate the recursive calls on the successors of a. By the induction hypothesis, they all return sets with all elements having the DfsStepSemantics property. We can prove by induction on the list, that then also their combination in `foldl_except_set` returns a set $V' \setminus \{a\}$ with all elements having the DfsStepSemantics property. Now the claim almost follows but we add the current vertex a into this set. The claim follows as soon as we can show that all successors of a are in $V' \setminus \{a\}$ since none of the successors reach either a cycle or an element where f is evaluated to an error and f is evaluated to ok on the vertex itself. Therefore also a has the DfsStepSemantics property and by that, any element in V' has the property. □

We now want to show that all successors of a are in $V' \setminus \{a\}$.

Lemma 30 (`dfs_step_returns_root_element`, `dfs_step_subset`)**.** *Let $\vec{v}$ be the input arguments with the vertex a and visited set V. If `dfs_step` $\vec{v}$ returns a set V', then a is member of V' and $V \subseteq V'$.*

Proof. We will prove this statement by induction on the number of vertices not covered by the current walk.

If a is a member of V, then we return V and the claim follows as $\subseteq$ is reflexive. If a is not a member of V, then we return the result of `foldl_except_set` on all the successors. Each individual call returns a superset of the input set by the inductive hypothesis. By induction on the size of the list, we see that then also `foldl_except_set` returns a superset of V (`foldl_except_set_subset`). In this set, we insert a to obtain V'. Therefore it contains a and it is still a superset of V. □

Using these results we have a criterion to show when `foldl_except_set` contains the original list. Due to Lean internal features, the result must be a bit more technical. As we need to show termination to unfold our function, we call `foldl_except_set` not on the successors of a directly, but instead on the attached version of this list, which is a different type. Therefore we talk about the function g in the following result which is supposed to map elements of the subtype `{x // x ∈ G.succesors}` back to the type A.

Lemma 31 (`foldl_except_set_contains_list_map`)**.** *Let `f: A → HashSet B → (Except String (HashSet B))` and g: $A \to B$ be functions such if $f(a, S)$ returns a hash set S' then $S \subseteq S'$ and $g(a) \in S'$. Then for any list l and hash* *set S, if `foldl_except_set f l S` returns a hash set S', then for all elements a of l, $g(a) \in l$.*

Proof. We prove this by induction on l. If l is empty all elements of l are in their mapped form in the resulting set.

If l has the shape $hd :: tl$, then we first compute $f(hd, S)$ which must return a set S' since the whole function returns a set. S' must contain $g(hd)$ and is passed into `foldl_except_set` which by the induction hypothesis contains all elements of tl in their mapped form. Due to the subset property, all elements of S' must also be contained, so that also $g(hd)$ is in the result. Therefore all elements of the list are contained in their mapped form in the resulting set, if it exists. □

Now we know that all successors of a are in $V' \setminus \{a\}$ and we finish the claim in lemma 29. This also finishes the preparations to prove theorem 6.

Proof of theorem 6. We prove this by induction on the number of uncovered vertices i.e. vertices that are in G, but not in the current walk. The base case is trivial as it contradicts the inputs `dfs_step`.

In the induction step, we prove both directions separately. In the forward direction `dfs_step` returns a set V' and we want to show that a has the DfsStepSemantics property. We know that every element in V' has the DfsStepSemantics property and that a is in V'. Hence, the forward direction is proven.

In the backward direction, we know that a has the DfsStepSemantics property and want to show that `dfs_step` returns ok. If a is in V, then we return ok and the claim is proven. Now assume that a is not in V. Since a has the DfsStepSemantics property and a can reach itself f must be evaluated to ok on a and its successors. Additionally, none of the successors of a can occur in the current path as this would imply that a can reach a cycle. It remains to show that `foldl_except_set` returns ok for which we use lemma 28. We know that `dfs_step` returns a superset of the input set lemma 29 and by the induction hypothesis all individual calls are ok iff the vertex has the DfsStepSemantics property which marks the independence of the visited set. We only have to show that all successors of a have the DfsStepSemantics property and use the induction hypothesis to show the claim. This follows due to the propagating properties of the DfsStepSemantics property. If any successor of a reaches a cycle, then also a would reach a cycle, which we know is not true. Similarly, any element reachable from a successor of a is also reachable from a and therefore has to evaluate to ok with its successor. □

Now we can show the proof of theorem 5.

Proof of theorem 5. As `dfs` is only the repeated evaluation of `dfs_step` wrapped in `foldl_except_set`, we again employ lemma 28 to reduce it to individual calls and use lemma 29 and theorem 6 to fulfill the requirements as in the previous proof.

Therefore we only have to show that all vertices have the DfsStepSemantics property iff the graph is acyclic and every vertex of the graph is evaluated to ok on f with its successors which we have already shown in lemma 23 and lemma 26 □

5.4 HashSets for depth-first search

In the previous section, we implement and prove the correctness of our depth-first search implementation. The proofs are independent of the underlying set implementation and also work with finite sets which we used first. A faster implementation is possible by using hash-based data structures such as hash maps and hash sets. Unfortunately, hash maps and hash sets almost have no lemmas proven about them already in Mathlib. Finite sets have in contrast already have many results proven about them. We try to bridge the gap by showing some results about the correctness of operations and presenting which assumptions we use as axioms. As the main focus of the thesis is not to formally verify hash sets we will only give a general overview of our results.

Hash maps are defined in Lean's standard library. We define hash sets as a hashmap mapping elements to the unit type. This will allow us to reuse the results we prove for hash sets in the hash map case which was our underlying graph model.

```
abbrev HashSet (A:Type) [Hashable A] [DecidableEq A] := HashMap
    A Unit
```

A hash map consists of buckets that are implemented as associative lists, which are essentially lists of pairs. The buckets are stored in a non-empty array.

```
def Std.HashMap.Imp.Buckets (α : Type u) (β : Type v) := {b :
    Array (AssocList α β) // 0 < b.size}
```

The hash map implementation combines the buckets with the size property as a natural number to see when the hash map has to be resized as very large buckets decrease the advantages in speed.

```
structure Std.HashMap.Imp (α : Type u) (β : Type v) where
  size    : Nat
  buckets : Imp.Buckets α β
```

This is so far only a reorganization of the data and does not explain the advantages. Indeed, placing elements randomly in buckets is not effective. The important property is called the well-formedness of the hash map buckets in Lean. This requires the underlying type to be hashable, i.e. a hash function for this type exists, and to have a boolean equality (`BEq`) which is a user-defined equality in contrast to the real equality based on the type. We do not care in this work about the boolean equality and use the equality of the type directly.

The well-formedness of a hash map bucket consists of two properties. Firstly, in every bucket, every element occurs at most once so that after an update no old values remain. Secondly, any element is only the bucket whose position in the array equals the hash value modulo the size. This allows us to only check a small part of the array. The hash map implementation is well-formed if the buckets are well-formed and the size property matches the number of buckets.

```
structure Std.HashMap.Imp.Buckets.WF [BEq α] [Hashable α]
    (buckets : Buckets α β) : Prop where
  distinct [LawfulHashable α] [PartialEquivBEq α] : ∀ bucket ∈
    buckets.1.data,
    bucket.toList.Pairwise fun a b => ¬(a.1 == b.1)
  hash_self (i : Nat) (h : i < buckets.1.size) :
    buckets.1[i].All fun k _ => ((hash k).toUSize %
    buckets.1.size).toNat = i
```

A hash map is then finally a well-formed hash map implementation. The hash map is mostly a wrapper around the hash map implementation and therefore we focus in this section on the proofs on the `Std.HashMap.Imp` level.

```
def Std.HashMap (α : Type u) (β : Type v) [BEq α] [Hashable α]
    := {m : Imp α β // m.WF}
```

The contains function uses the well-formedness of the hash map implementation to look in the bucket placed in the position of the hash value of the element. The function `mkIdx` calculates the position and provides a proof that there is a bucket at this position.

```
def contains [BEq α] [Hashable α] (m : Imp α β) (a : α) : Bool
    :=
  let ⟨_, buckets⟩ := m
  let ⟨i, h⟩ := mkIdx buckets.2 (hash a |>.toUSize)
  buckets.1[i].contains a
```

In the depth-first search implementation, we only need the insertion and contains function as we only add the visited vertices. We never unvisit an element and therefore focus on the insertion function. We are given two elements a and b. If a is already present in the bucket of its hash value we simply replace the old value of a by b in a's bucket. If a is not present we add the element at the front of the corresponding bucket. This might increase the size to a value that is larger than the capacity. In this case, expand creates a new larger empty bucket and reinserts all the previous elements.

```
def insert [BEq α] [Hashable α] (m : Imp α β) (a : α) (b : β) :
    Imp α β :=
  let ⟨size, buckets⟩ := m
  let ⟨i, h⟩ := mkIdx buckets.2 (hash a |>.toUSize)
  let bkt := buckets.1[i]
  bif bkt.contains a then
    ⟨size, buckets.update i (bkt.replace a b) h⟩
  else
    let size' := size + 1
    let buckets' := buckets.update i ( AssocList.cons a b bkt) h
    if numBucketsForCapacity size' ≤ buckets.1.size then
      { size := size', buckets := buckets' }
    else
      expand size' buckets'
```

The main result we want to prove is concerned with the correctness of the insertion with regard to the contains function.

Lemma 32 (HashSet.contains_insert)**.** *Let S be a hash set and a an element. For any element a', a' is contained in the insertion of a into S iff a' is contained in S or a' and a are equal.*

As hash maps are not only used to serve as hash sets but also as the graph model we want a more general result to reuse. The contains function does not tell us something about the value that is contained with the key. In order to reason about the key-value pairs we introduce the key-value member `kv_member` which is true whenever a key-value pair is in the bucket corresponding to the key.

```
def HashMap.Imp.kv_member (m: HashMap.Imp A B) (a: A) (b:B):
    Bool :=
  let ⟨_, buckets⟩ := m
  let ⟨i, h⟩ := mkIdx buckets.2 (hash a |>.toUSize)
  let bkt := buckets.1[i]

  (a,b) ∈ bkt.toList
```

An element a is contained in m iff there exists an element b with a and b being a `HashMap.Imp.kv_member` as we look in the same bucket in both cases. Both functions use however the mkIdx function to access the elements. As we are only using hash maps, we can consider any well-formed hash map implementations and collect all elements in the buckets in a single finite set, the key-value set `HashMap.Imp.kv`.

```
def HashMap.Imp.Buckets.kv (buckets: HashMap.Imp.Buckets A B):
    Finset (A × B) := Array.foldl (fun x y => x ∪
    y.toList.toFinset) ∅ buckets.val

def HashMap.Imp.kv (m: HashMap.Imp A B): Finset (A × B) := m.2.kv
```

By induction on the size of the list, we manage to show that an element is in `HashMap.Imp.kv` iff it is in some bucket given as an input to (`Array.foldl_union`). Additionally, we show that `HashMap.Imp.kv` is equivalent to `HashMap.Imp.kv_member`.

Lemma 33 (HashMap.Imp.kv_member_iff_in_kv)**.** *Let m be a well-formed hash map implementation. Then for any elements a and b, a and b are a key-value member of m iff a and b are in $m.kv$.*

Proof. Any a and b that are a key-value member of m are in a bucket and hence in $m.kv$. For the reverse direction, we need the well-formedness. If a and b are in $m.kv$, they are in some bucket bkt of m. Due to the well-formedness of m, this bucket is accessible by the hash value of a. Therefore a and b are also a key-value member of m. $\square$

In the following proof for the insertion, we can therefore use the key-value set and do not have to use the more complicated functions using the hash-value.

Lemma 34 (`HashMap.Imp.insert_semantics`)**.** *Let m be a well-formed hash map implementation. Then for any keys a and a' and values b and b', we have that a' and b' are a key-value member of $m.insertab$ iff a' and b' are a key-value member of m and a and a' are different, or if (a, b) and (a', b') are equal.*

Proof. The first two cases are rather obvious as we only change a single bucket. If a is already contained in m, we replace its value by b. Hence all key-value members of this hash map implementation are all the previous members of m except for the replaced previous value for a or the new pair (a, b). The second case is where a is not contained in m and adding the new pair does not bring the hash map implementation over its maximum size. In this case, we add the element to the front of the bucket. All previous key-value members are preserved and as a is not contained, there is no other pair with the key a, so the claim follows.

In the last case, adding (a, b) brings the hash map implementation over its maximum size. Hence a larger hash map implementation is created and every element reinserted. Until then we added (a, b) as in the previous case so that it only remains to show that all key-value members are preserved which we show by showing that the key-value set is equal (`HashMap.Imp.expand_preserves_mem`). The main work of this proof is done in `HashMap.Imp.expand_go_mem` by induction but is rather low-level and hence omitted here. $\square$

This result is the main result of this section and we will use it to now formally verify the real operations that we use in the program. As the hash map is a pair of the implementation and a proof of the implementation's well-formedness, we can lift these results to hash maps as desired. We already noted that a is contained in a hash map m iff there exists a b such that a and b are a key-value member of m. Hence we gain the desired result for the contains function.

Lemma 35 (`HashMap.contains_insert`)**.** *Let m be a hash map. Then for any elements a and a' and b, a' is contained in $m.insertab$ iff a' is contained in m or a and a' are equal.*

A hash set of type A was defined as a hash map mapping elements of type A to the unit type. Hence, the result for hash sets follows as well.

Lemma 36 (`HashSet.contains_insert`)**.** *Let S be a hash set and a an element. For any element a', a' is contained in the insertion of a into S iff a' is contained in S or a' and a are equal.*

We additionally use the `findD` function to look up values for a key a. This function uses `find?` to look for a key a which returns an option. If this option contains an element it is returned else we return the given default element.

```
def find? [BEq α] [Hashable α] (m : Imp α β) (a : α) : Option β
    :=
  let ⟨_, buckets⟩ := m
  let ⟨i, h⟩ := mkIdx buckets.2 (hash a |>.toUSize)
  buckets.1[i].find? a

def findD (self : HashMap α β) (a : α) (b₀ : β) : β :=
    (self.find? a).getD b₀
```

An alternative characterization of findD that simplifies our later work is the following.

```
lemma HashMap.findD_eq_find? (m: HashMap A B) (a:A) (b:B):
    m.findD a b = match m.find? a with
                  | some b' => b'
                  | none => b
```

In a well-formed hash-map implementation m, b is the result of $m.find?\ a$ iff a and b are key-value members of m as we look in the same bucket and every element in a well-formed bucket has a unique key. (`HashMap.find_iff_kv`) Therefore we can reuse the insertion semantics in the $find?$ case:

Lemma 37 (`HashMap.find_insert`)**.** *Let m be a hash map and a and b be a key-value pair. Then for any element a', `(m.insert a b).find? a'` yields `some b` iff a and a' are equal or else the result of $m.find?\ a'$.*

In the graph, we never use $find?$ as we do not want to deal with the option type. We instead use $findD$ and use the empty list as the default element. One requirement would be that the result of $findD$ is correct for any key a after inserting a with a value b.

Lemma 38 (`HashMap.findD_insert'`)**.** *Let m be a hash map and a and b be a key-value pair. For any elements b',`(m.insert a b).findD a b'` yields b.*

Proof. By lemma 37 `(m.insert a b).find? a` yields `some b`. By the definition of `findD`, therefore `findD` also returns b and the claim is proven. □

The operations `contains` and `findD` are sufficiently verified for our purposes. These results can also be reused as long as the used equality is the equality of the type and not a user-defined `BEq`.

We did not manage to verify the `ofList` operation that takes a list l of pairs of the type $A \times B$ and converts it into a `HashMap A B` m. We use it to create the initial graph and desire that the result of findD is the same on l and m so that the graph is exactly the given input (`HashMap.findD_ofList_is_list_find_getD`).

Chapter 6

Completeness

In the previous chapter, we describe a method to check why an atom is in the datalog semantics. This criteria is however not sufficient to recognize a solution. The empty list passes this test for any program while not being the semantics in most cases. Proof trees were a method to recognize why a ground atom is part of the semantics, but we are not aware of any simple way to describe why a ground atom is not in the semantics. Instead, we want to show that the set of ground atoms in the proof trees, S, is complete in the sense that nothing else can derived from it anymore. This is the case when the T_P operator has a fixed point for S or when S is a model. In this chapter, we are going to create a certified model checker to show the completeness. If S passes the tree validation algorithm and is a model the following statements hold:

$$S \subseteq \texttt{proofTheoreticSemantics P d} = \texttt{modelTheoreticSemantics P d} \subseteq S$$

We only accept safe rules for the model checker which we define using `atomVariables`. A rule is safe if every variable in the head occurs already in the body. Safe rules allows us to ground a rule using only the atoms in the given interpretation we want to test. Unsafe rules would require us to replace a variable that does not occur in the body with every constant symbol which is depending on the constant type might yield an infinite ground program. As many practical datalog reasoners also only accept safe rules, this restriction seems acceptable to us.

```
def rule.isSafe (r: rule τ): Prop :=
atomVariables r.head ⊆ List.foldl_union atomVariables ∅ r.body
```

6.1 Partial ground rules

We defined the model property on the ground program *ground*(P). In order to check if a set of ground atoms is a model for a program and a database we therefore have to ground the program. We want to avoid simply grounding all the rules at once and instead do it more intelligently because the number of groundings is very large or even infinite. For this, we introduce a new data structure, the partial ground rule. This bears some similarities to the rules we defined in chapter 3. It has a head that is an atom. The body is split into two lists. The first list contains the ground atoms and represents the atoms in the rule we already grounded, whereas the second list consists of the so far ungrounded atoms in the rule body. We want to move the ungrounded atoms one by one into the grounded list by applying substitutions, which map all variables of this atom to constants so that we can transform this atom into a ground atom.

```
structure partialGroundRule (τ: signature)  where
    head: atom τ
    groundedBody: List (groundAtom τ)
    ungroundedBody: List (atom τ)
```

Example 24. *A rule* $r := q(X) : -r(a,b), t(X,c), s(c,d), u(d,X)$. *may be viewed as the following partial ground rule* $pgr_1 =$

```
{
    head:= q(X),
    groundedBody := [],
    ungroundedBody := [r(a, b), t(X, c), s(c, d), u(d, X)]
}
```

This representation does not look any different from the rule itself as we do not use the grounded body at all. We can however move ground atoms from the ungrounded body into the grounded body. The order of the atoms in the body does not matter semantically as we use a set definition when defining the criteria for a rule being true. Therefore we can simply move all ground atoms in the grounded body. Another representation of the same rule as a partial ground is therefore $pgr_2 =$

```
{
    head:= q(X),
    groundedBody := [r(a, b), s(c, d)],
    ungroundedBody := [t(X, c),  u(d, X)]
}
```

We can transform any rule into a partial ground rule by setting the head as the head, the body as the ungrounded body and setting the grounded body to be empty. We have done this in the previous example when creating pgr_1.

```
def partialGroundRule.fromRule (r: rule τ): partialGroundRule τ
    :=
{
    head := r.head,
    groundedBody := [],
    ungroundedBody := r.body
}
```

We choose this representation instead of the approach used for pgr_2 as this does not require iterating over the whole body to find ground atoms. As we will apply multiple substitutions in the grounding process, we will create ground atoms in different places anyway.

Any partial ground rule can also be transformed back into a rule by concatenating the grounded and ungrounded body.

```
def partialGroundRule.toRule (pgr: partialGroundRule τ)
: rule τ :=

{
    head:= pgr.head,
    body := (List.map (groundAtom.toAtom) pgr.groundedBody)
    ++ pgr.ungroundedBody
}
```

This operation is inverse to the `partialGroundRule.fromRule` operation.

Lemma 39. *[`partialGroundRuleToRuleInverseToFromRule`] For any rule r, r equals `(partialGroundRule.fromRule r).toRule`*

This does not hold if we swap the operations as we do not explicitly move atoms without variables from the start of the body into the grounded body.

Example 25. *The application of `partialGroundRule.toRule` on pgr_1 yields r as predicted by the lemma.*

If we swap both functions and first apply `partialGroundRule.toRule` to a partial ground rule and then convert the resulting rule back to a partial ground rule, we will not receive the original partial ground rule in most cases. Applying partialGroundRule.toRule to pgr_2 results in the rule r'

$$q(X) : -r(a, b), s(c, d), t(X, c), u(d, X).$$

Converting r' back into a partial ground rule with `partialGroundRuleFromRule` we gain

```
{
    head:= q(X),
    groundedBody := [],
    ungroundedBody := [r(a, b), s(c, d), t(X, c),  u(d, X)]
}
```

which is different from pgr_2.

Using the transformation to rules we can lift a rule being true to the partial ground rules.

```
def partialGroundRule.isTrue (pgr: partialGroundRule τ) (i:
    interpretation τ): Prop := ∀ (g: grounding τ), ruleTrue
    (ruleGrounding pgr.toRule g) i
```

We also define safety for partial ground rules. Since ground atoms have no variables a rule r is safe if the partial ground rule created from r is safe.

```
def partialGroundRule.isSafe (pgr: partialGroundRule τ): Prop :=
atomVariables pgr.head ⊆ List.foldl_union atomVariables ∅
    pgr.ungroundedBody
```

So far we only split the body into two parts and have the goal of applying substitutions to move everything into the grounded body. This is not too different from just applying groundings directly. Splitting the grounding into multiple substitutions allows us to potentially stop early. If the substitutions we applied so far resulted in a ground atom in the body that is not part of the interpretation i we already know that the rule is true in i. No matter how the remaining variables are mapped, the body will never be a subset of i and the rule is therefore true. We call a partial ground rule *active in an interpretation* i when all ground atoms in the grounded body are in i.

```
def partialGroundRule.isActive (pgr: partialGroundRule τ) (i:
    interpretation τ):=
∀ (ga: groundAtom τ), ga ∈ pgr.groundedBody → ga ∈ i
```

Lemma 40 (notActiveRuleIsTrue). *Let pgr be a partial ground rule. If pgr is not active in an interpretation i, then it is true in i.*

Any partial ground rule that is created from a rule r is active in any interpretation i, since the grounded body is empty.
(partialGroundRule.fromRuleIsActive).

6.2 Explore grounding

Now we want to present an algorithm that checks whether a list of rules is true in an interpretation that is given as a list of ground atoms using partial ground rules.

This is done by modelChecker which calls exploreGrounding on the partial ground rule created from every rule and accepts if no rule raises an error. We require a proof that every rule in the program P is a safe rule and derive from this that any partial ground rule created from a rule in P is safe.

```
def modelChecker (i: List (groundAtom τ)) (P: List (rule τ))
    (safe: ∀ (r: rule τ), r ∈ P → r.isSafe): Except String Unit
    :=
have safe': ∀ (r: rule τ), r ∈ P → (partialGroundRule.fromRule
    r).isSafe := by
    intros r rP
    rw ←[ safePreservedBetweenRuleAndPartialGroundRule]
    apply safe r rP
List.map_except_unit P.attach (fun ⟨x, _h⟩ => exploreGrounding
    (partialGroundRule.fromRule x ) i (safe' x _h) )
```

This function returns ok iff all rules in P are true in the interpretation i viewed as a list.

Theorem 7. *[modelCheckerUnitIffAllRulesTrue] Let i be a list of ground atoms and P be a list of rules that are all safe. Then modelChecker returns ok for P and i iff all rules in P are true in i.*

The main work is done in exploreGrounding which takes a partial ground rule, an interpretation as a list and a proof that the given partial ground rule is safe.

```
def exploreGrounding (pgr: partialGroundRule τ) (i: List
    (groundAtom τ)) (safe: pgr.isSafe): Except String Unit :=
match h:pgr.ungroundedBody with
| [] =>
    let head' := atomWithoutVariablesToGroundAtom pgr.head
    (headOfSafePgrWithoutGroundedBodyHasNoVariables pgr safe h)

    if head' ∈ i
    then Except.ok ()
    else Except.error ("Unfulfilled rule: " ++ ToString.toString
    pgr.toRule)
| hd::tl =>
    if noVars:atomVariables hd = ∅
    then
    if atomWithoutVariablesToGroundAtom hd noVars ∈ i
    then exploreGrounding (moveAtomWithoutVariables pgr hd tl
    noVars) i (moveAtomWithoutVariablesPreservesSafety pgr hd tl
    h noVars safe)
    else Except.ok ()
    else
    List.map_except_unit (getSubstitutions i hd).attach (fun ⟨s,
    _h⟩ =>
        let noVars':= inGetSubstitutionsImplNoVars i hd s _h
        exploreGrounding (groundingStep pgr hd tl s noVars') i
    (groundingStepPreservesSafety pgr hd tl s h noVars' safe)
    )
```

This function works recursively on the list of ungrounded atoms in the body. If this list is empty, then there are no variables in the head since *pgr* is a safe partial ground rule. We can convert the head into a ground atom and check whether it is in i. If the head is in i, we accept, else we raise an error.

If there is at least one element *hd* in the ungrounded body we consider two cases. If there are no variables in *hd*, we do not have to do any grounding and can move *hd* directly into the grounded body. We can however stop earlier if the resulting element is not in *i* as then the resulting rule is not active anymore and thus true in *i*.

```
def moveAtomWithoutVariables (pgr: partialGroundRule τ) (hd:
    atom τ) (tl: List (atom τ)) (noVars: atomVariables hd = ∅):
    partialGroundRule τ :=
{
    head := pgr.head,
    groundedBody := pgr.groundedBody ++
    [atomWithoutVariablesToGroundAtom hd noVars]
    ungroundedBody := tl
}
```

The safety of *pgr* is preserved by `moveAtomWithoutVariables` if the ungrounded body was $hd :: tl$, since we only removed *hd* from the ungrounded body which has no variables at all (`moveAtomWithoutVariablesPreservesSafety`). Therefore we can call `exploreGrounding` again on the resulting partial ground rule.

If *hd* has variables we need to apply a substitution to transform *hd* into a ground atom occurring in *i*. If no such atom exists, then we can stop as the rule will not be active. For this, we reuse `matchAtom` we defined in chapter 4 and check for every atom *a* in *i* if there is a substitution *s* that maps *hd* to *a*. This may return none for some atoms if no such *s* exists. The none values are filtered out using `List.filterMap`.

```
def getSubstitutions (i: List (groundAtom τ))(a: atom τ): List
    (substitution τ) := List.filterMap (fun x => matchAtom
    emptySubstitution a x) i
```

After applying a substitution from `getSubstitutions` to *hd* it has no further variables as it is equal to a ground atom (`inGetSubstitutionsImplNoVars`). Therefore we can move the resulting atom after applying such a substitution into the grounded body. We apply these substitutions to every atom that is not a ground atom so that a variable does not get mapped to different constants later. For the correctness of this transformation, we require that $hd :: tl$ is the ungrounded body of *pgr*.

```
def groundingStep (pgr: partialGroundRule τ) (hd: atom τ) (tl:
    List (atom τ)) (s: substitution τ) (noVars: atomVariables
    (applySubstitutionAtom s hd) = ∅ ): partialGroundRule τ :=
{
    head := applySubstitutionAtom s pgr.head,
    groundedBody := pgr.groundedBody ++
    [atomWithoutVariablesToGroundAtom (applySubstitutionAtom s
    hd) noVars],
    ungroundedBody := List.map (applySubstitutionAtom s) tl
}
```

Since we remove the same variables from the head as the body, the safety of this rule is preserved (groundingStepPreservesSafety) and we can again call exploreGrounding on it.

In any recursive call, the number of atoms in the ungrounded body decreases so that the function terminates.

The desired property of exploreGrounding is the following.

Theorem 8. *[exploreGroundingSemantics] Let pgr be an active and safe partial ground rule and i be a list of ground atoms. Then exploreGrounding returns ok for pgr and i iff pgr is true in i.*

Before proving this theorem, we finish the proof of theorem 7.

Proof of theorem 7. The modelChecker returns ok iff exploreGrounding returns ok for the partial ground rule obtained from a rule r. The resulting partial ground rule is safe and active. From theorem 8 we know that it returns ok iff the partial ground rule is true. From the definition of partialGroundRule.isTrue we know that a partial ground rule is true, if the resulting rule from toRule is true. By lemma 39 this is equal to r. □

The remainder of this section is now spent proving theorem 8. We do this by induction on the length of the ungrounded body for arbitrary partial ground rules *pgr*.

If the ungrounded body *pgr* has the length zero, then it is the empty list. As *pgr* is active, we have that all elements in the grounded body are in the interpretation i. As the ungrounded body is empty, the body of the rule r resulting from *pgr* is a subset of i. Then r is true iff the head of *pgr* is in i which is exactly the case when exploreGrounding returns ok.

In the induction step, we assume that for all partial ground rules whose ungrounded body has the length n exploreGrounding returns ok iff the partial ground rule is true.

Let *pgr* be a partial ground rule whose ungrounded body has the length $n + 1$. If the leading element *hd* has no variables we check whether it is in i. If it is not in i, explore grounding returns ok. The resulting rule r of *pgr* is always true since *hd* is in the body and grounding an atom without variables yields the same atom(groundingAtomWithoutVariablesYieldsSelf). Therefore the body will never be a subset of i and is always true.

If *hd* is in i, then we apply moveAtomWithoutVariables to *pgr*. Since *pgr* was active and *hd* is in i also the resulting partial ground rule is active and its ungrounded body is shorter so that we can apply the induction hypothesis. What remains to show is that moveAtomWithoutVariables *pgr hd tl* $(p : atomVariables\ hd = \emptyset)$ is true in i iff *pgr* is true in i. We note from the definitions that a partial ground rule is true if the resulting rule from the toRule function is true. Hence it suffices to show that moveAtomWithoutVariables *pgr hd tl* $(p : atomVariables\ hd = \emptyset)$ and *pgr* result in the same rule.

(partialGroundRule.isTrue_of_equal_toRule)

This is the case as we only move *hd* from the start of the ungrounded body to the end of the grounded body which results in the same rule body and since converting an atom into a ground atom if it has no variables results in the same atom (`groundAtomToAtomOfAtomWithoutVariablesToGroundAtomIsSelf`).

Now we only have to consider the case that *hd* has variables. Then we apply all substitutions of `getSubstitutions` to *pgr* and call `exploreGrounding` recursively on every resulting partial ground rule. Using the properties of `matchAtom` we know that a substitution *s* is in `getSubstitutions` *i hd* iff it is the minimal substitution that matches *hd* to some ground atom in *i* (`inGetSubstitutionsIffMinimalSolutionAndInInterpretation`).

Using the induction hypothesis and `List.map_except_unitIsUnitIffAll`, it remains to show that for any substitution *s* from `getSubstitutions` *i hd* `groundingStep` *pgr hd tl s* is true in *i* iff *pgr* is true in *i*. Any resulting partial ground rule is active since applying *s* to *hd* yields a ground atom from *i*. Recall that the definition of is true only depends on the `toRule` conversion and that the grounding step has the following shape.

```
{
head := applySubstitutionAtom s pgr.head,
groundedBody := pgr.groundedBody ++
    [atomWithoutVariablesToGroundAtom (applySubstitutionAtom s
    hd) noVars],
ungroundedBody := List.map (applySubstitutionAtom s) tl
}
```

Since applying substitutions to ground atoms does not change a ground atom, we can first convert the partial ground rule to a rule and then apply the substitution (`swapPgrApplySubstitution`) to gain the same rule. We then only have to show that for any substitution *s* from `getSubstitutions` *i hd* `applySubstitutionRule` *s pgr.toRule* is true iff *pgr* is true, which follows from the following lemma. In this lemma we call a rule *r* true in an interpretation *i* if grounding *r* with any *g* yields a ground rule that is true in *i*. We additionally call a substitution minimal for an atom *a* and an interpretation *i* if *s* is the minimal substitution according to $\subseteq$ that matches *a* to some ground atom in *i*.

Lemma 41 (`replaceGroundingWithSubstitutionAndGrounding`)**.** *Let r be a rule, a an atom from the body of r and i an interpretation. Then r is true in i iff for all minimal substitutions s for a and i applying s to r yields a rule r' that is true in i.*

Proof. For the forward direction, we know that grounding r with any grounding g results in a true ground rule in i. Let s be some minimal substitution for a and i and r' be the rule that is the result of applying s to r. We want to show that r' is true in i. That is the case if grounding r' by any grounding g yields a true ground rule gr in i. We obtained gr from r by first applying s and then g. We can combine s and g into a grounding by using the result of s for a variable when it is defined and else the result of g.

```
def combineSubstitutionGrounding (s: substitution τ) (g:
    grounding τ): grounding τ := fun x => if h: Option.isSome (s
    x) then Option.get (s x) h else g x
```

Grounding a rule with `combineSubstitutionGrounding` s g is equivalent to first applying s and then grounding with g for terms, atoms and rules (`combineSubstitutionGroundingEquivRule`). Hence we can combine g and s into the grounding `combineSubstitutionGrounding` and use the assumption to conclude the forward direction.

For the backward direction, we know that for any minimal substitution s for a and i applying s to r yields a true rule in i. We have to show that then we can also apply any grounding g' to r and obtain a true ground rule in i. If g' grounds a to an atom that is not in i, then the rule is true in i since the body is not a subset of i.

If g' grounds a to an atom in i, we can convert g' into a minimal substitution for a and i using `atomSubOfGrounding` by only mapping the variables of a to $g'(a)$.

```
def atomSubOfGrounding (a: atom τ) (g: grounding τ):
    substitution τ := fun x => if x ∈ atomVariables a then some
    (g x) else none
```

Applying first `atomSubOfGrounding` a g' and then grounding using g' is equivalent to just grounding with g', because any variable v is mapped to the result of $g(v)$ (`atomSubOfGroundingGroundingEqGroundingOnRule`). By the assumption we therefore know that grounding r by g' yields a true ground rule in i which concludes the backward direction. □

Chapter 7

Evaluation

In the previous sections, we proved the correctness of the algorithms to check the soundness and completeness of datalog reasoning results. Now, we are interested in the practicability of these algorithms on actual data. We combined these algorithms into a command line tool that takes a file consisting of the problem and the certificates and tells us whether the result is correct according to the certificate.

7.1 Input format

The input format is JSON-based because Lean offers already direct support for JSON. Similarly, as we were able to derive decidable equality or the inhabited class, we can also derive functions that convert Lean objects to JSON objects or try to create a Lean object from a JSON object.

We can define mock terms similar to `term` but with the variables and constants as simple strings. Lean knows how to read and write strings into JSON hence we can derive the JSON methods.

```
inductive mockTerm
| constant: String → mockTerm
| variable: String → mockTerm
deriving DecidableEq, Lean.FromJson, Lean.ToJson, Repr
```

Using this type we can similarly as to `atom` define `mockAtom`. In contrast to real atoms, we do not require a proof that the number of terms matches the arity of the predicate symbol as encoding such a proof is difficult and we have no information about the arity. The symbol is again just a string.

```
structure mockAtom where
  (symbol: String)
  (terms: List mockTerm)
deriving DecidableEq, Lean.FromJson, Lean.ToJson, Repr
```

Mock atoms form mock rules similarly as atoms form rules and a program is simply a list of mockRules. Lists are a basic feature of the JSON decoder which allows us to get the from a JSON file.

Example 26. *The program*

$$
\begin{aligned}
&P = \{ \\
&T(?x, ?y) \leftarrow E(?x, ?y), Q(a). \\
&\}
\end{aligned}
\tag{7.1}
$$

is represented as following in JSON:

```
"program": [
{
    "head": {
        "symbol": "T",
        "terms": [
            {
                "variable": "?x"
            },
            {
                "variable": "?y"
            }
        ]
    },
    "body": [
        {
            "symbol": "E",
            "terms": [
                {
                    "variable": "?x"
                },
                {
                    "variable": "?y"
                }
            ]
        },
        {
            "symbol": "Q",
            "terms": [
                {
                    "constant": "a"
                }
            ]
        }
    ]
}]
```

Afterwards, we go through the program twice. In the first run, we collect all the predicate symbols and their arities (`parsingArityHelper`) into a list and report an error if a predicate symbol is used in multiple atoms with different amounts of terms. Using the list we can construct a signature (`parsingSignature`). We use for constants and variables simply the set of strings as types and for predicate symbols the subset of strings that occurred as symbols. These choices allow us to directly inherit the requirements for the signature elements such as decidable equality or hashability.

In the second run, we then transform every mock object into the corresponding datalog object of the previously created signature.

The second part is the input file is either a list of trees or a graph. For each of the objects, we define again mock objects and transform them into the corresponding datalog objects after we have transformed the program.

```
inductive jsonTree (A: Type)
| node (label: A) (children: List (jsonTree A))
deriving Lean.FromJson, Lean.ToJson

-- graph validation
structure mockEdge where
  (vertex: mockAtom)
  (successors: List (mockAtom))
deriving DecidableEq, Lean.FromJson, Lean.ToJson

structure mockGraph where
  (edges: List mockEdge)
deriving Lean.FromJson, Lean.ToJson
```

Additionally, there are two command line options that can be set. Firstly, the option *-g* specifies that the input file uses a graph instead of a list of trees which is the default option. Secondly, we tell the program with *-c* to also use the `modelChecker` to check for completeness.

The file does not include a database yet because these databases are often very large in practice which requires more work to replicate in Lean. Additionally, we would need to prove that the database is correctly read from the input. All evaluations are done with a mock database which assumes that any leaf of a tree or graph is in the database and that the database is always contained in the model during the model checking.

7.2 Results

In this section, we want to evaluate the program in practice. We are interested in the feasibility of our approach to verify proof trees and to check a complete result. Additionally, we want to compare the tree verification with the graph-based verification.

We will use three kinds of datalog programs to test our approach:

1. We will use the transitive closure programs we considered in this work multiple times. The transitive closure is often used as a subroutine in recursive datalog programs and thus an important benchmark. The exact program uses the transitive predicate once in the body and is depicted below:

   $$\begin{aligned} Trans(?x, ?y) &\leftarrow Edge(?x, ?y). \\ Trans(?x, ?z) &\leftarrow Edge(?x, ?y), Trans(?y, ?z). \end{aligned}$$

   This program can be used with different directed graphs and may be considered a reachability problem. We create the graphs in a Python script using networkx's `gnm_random_graph` which uniformly selects a graph from the graphs for a given number of vertices and edges. We use different densities to test different graphs. The density describes the ratio between the number of edges and the maximal number of edges.

2. We will reuse the program from example 20 on chains of different lengths. We have theoretically analyzed that the proof trees are exponentially larger than the corresponding proof graphs and want to see if the algorithms also respect that in practice.

3. Finally, we will use a large example from practice. The problem deals with reasoning in the OWL EL profile of the web ontology language OWL whose transformation to datalog is described in [25]. As the database, we reuse the medical ontology GALEN which is also used in the original paper. The ontology is preprocessed in CSV files and leads to around 2.4 million derived atoms.

The program and all scripts to generate the example data and run the experiments available in the diplomarbeit subfolder of the examples in the repository.

We use the datalog engine Nemo[22] to evaluate these examples. Nemo allows us to specify multiple facts as a file and ask for their derivations. This is returned in a machine-readable JSON format which allows us to transform this and the program into the input file format described in the previous section.

Any other datalog engine that returns proof trees or graphs can also be used as long as such a program exists to convert it into the input format. This would also be possible with Soufflé[24] but the proof trees there are only given in an ASCII art style which complicates the parsing.

The experiments run on a modern laptop with an AMD Ryzen 7 processor, 8 GB of RAM using Ubuntu in the Windows subsystem for Linux. We will measure the processor time using Python's time module in three categories. Firstly, we measure the time our tool takes to verify the solution. Secondly, we measure the time Nemo takes to calculate the solution in our scenario. As

Table 7.1: Comparision between tree validation and a completeness check in scenario 1

		Nemo time [s]		**Preparation time [s]**		**Validation time [s]**	
		mean	std	mean	std	mean	std
Density	**Completeness**						
0.01	False	0.03	0.01	0.84	0.30	0.04	0.01
	True	0.03	0.01	0.84	0.30	0.11	0.06
0.05	False	0.05	0.00	239.50	17.18	0.45	0.04
	True	0.05	0.00	239.50	17.18	26.24	1.64
0.1	False	0.06	0.00	180.94	1.41	0.42	0.03
	True	0.06	0.00	180.94	1.41	39.30	3.53
0.3	False	0.10	0.01	171.27	1.54	0.34	0.01
	True	0.10	0.01	171.27	1.54	104.95	3.14
0.5	False	0.16	0.01	193.46	1.58	0.31	0.00
	True	0.16	0.01	193.46	1.58	193.41	4.94

there is to the best of our knowledge currently no alternative program capable of verifying datalog reasoning results this is the only other time available for comparison. We believe that the verification can take a bit longer than the computation but it should not take hours for a task that Nemo can solve in seconds. These two times ignore however an important aspect. We need to create first the input file that contains the program and the proof trees or proof graph. This is the third time we measure, the preparation time. This consists of the time that Nemo takes to create the trace which takes longer than just calculating the solution and the time we need to transform the trace into our JSON format which is done by a Python script.

We repeat each measurement five times and display in each table the arithmetic mean and standard derivation in seconds for each time calculated by Python's pandas library.

In scenario (1), depicted in table 7.1, we created a random graph with the given density and asked Nemo to explain all derivations. The number of derivations is up to 10000 facts in the case of a density of 0.3 or 0.5. We use the tree-based input and want to compare the time it takes to verify all proof trees with the time that we require to additionally perform a completeness check. We see that validating all trees is fast while being a bit larger than the computation times by Nemo. In contrast to that, a completeness check takes in most cases much more time than just verifying all trees. The time for a completeness check increases with the density of the graphs as the graph has more edges with increasing density which increases the number of edges we check for substitutions. In contrast to that the validation time without a completeness check does not increase with the density. We propose that with the increasing densities, there are more edges which lead to shallower trees which can be faster validated even if their number increases.

We see in this scenario that it is definitely feasible to validate all trees of the example as the time is roughly comparable to Nemo's time. We did not encounter an invalid tree or an invalid model. Next, we want to compare the performance of the tree-based input format with the graph-based input format.

Table 7.2: Comparision between trees and graphs in Scenario 1

		Number of nodes		**Preparation time [s]**		**Validation time [s]**	
		mean	std	mean	std	mean	std
Density	**Type**						
0.01	graph	606.20	117.57	0.19	0.06	0.03	0.00
	tree	3858.40	1763.22	0.84	0.30	0.04	0.01
0.05	graph	10295.40	157.80	18.07	1.55	0.24	0.02
	tree	59235.60	1374.27	239.50	17.18	0.45	0.04
0.1	graph	10990.00	0.00	17.68	0.21	0.27	0.04
	tree	44750.00	126.88	180.94	1.41	0.42	0.03
0.3	graph	12970.00	0.00	19.06	0.24	0.28	0.01
	tree	34061.60	1.67	171.27	1.54	0.34	0.01
0.5	graph	14950.00	0.00	21.12	0.22	0.27	0.01
	tree	30100.00	0.00	193.46	1.58	0.31	0.00

Table 7.3: Results for scenario 2

		Number of nodes	**Nemo time [s]**		**Preparation time [s]**		**Validation time [s]**	
		mean	mean	std	mean	std	mean	std
Type	**Size**							
graph	10	18.00	0.03	0.01	0.03	0.01	0.05	0.06
	15	28.00	0.02	0.00	0.03	0.00	0.02	0.00
	20	38.00	0.02	0.00	0.03	0.00	0.03	0.00
tree	10	1022.00	0.03	0.01	0.07	0.01	0.03	0.00
	15	32766.00	0.02	0.00	2.71	0.09	0.16	0.00
	20	1048574.00	0.02	0.00	207.66	3.98	4.38	0.20

We do not use the completeness check this time as we have seen in the previous results that the completeness check takes the majority of the time while not depending on the particular input format.

In table 7.2 see for each density the times we require for trees and the graph. We see that the validation of the graph is a bit faster than validating the trees in each case but still in the same area. Graphs are especially faster for the densities of 0.1 and 0.05 where the trees need the most nodes as there are many nodes connected while still needing long paths. If the paths are shorted with a higher density, the proof trees are smaller and the advantage diminishes. The main advantage of the graph-based input is however seen in the preparation times. Transforming Nemo's trace output into a directed graph is much faster than transforming the trace output into a list of trees because Nemo's trace output is graph-like. Therefore graphs seem preferable in this use-case as we cannot validate the results without preparing them.

In scenario (2), depicted in table 7.3, the advantage of graphs over trees is seen clearly even in the validation time. For chains of the length 15 and 20 the tree-based validation takes significantly more time while the graph-based validation is still comparable to Nemo's computation time. Additionally, the preparation for graphs is again much faster than the preparation time for the tree-based input format. This is because Nemo's trace looks graph-like so we only have to transform the atoms into the JSON format. Creating a tree requires us to first build the graph and then explore all paths. When trying to evaluate

Table 7.4: Results for scenario 3

Number of atoms	Type	Number of nodes mean	Number of nodes std	Preparation time [s] mean	Preparation time [s] std	Validation time [s] mean	Validation time [s] std
1	graph	18.60	7.20	8.90	0.06	0.02	0.00
	tree	19.00	7.94	8.93	0.02	0.03	0.00
10	graph	230.80	39.33	9.10	0.11	0.03	0.00
	tree	286.00	46.03	9.13	0.09	0.03	0.00
100	graph	1876.00	148.04	11.04	0.44	0.05	0.00
	tree	2835.40	373.87	11.46	0.49	0.04	0.00
1000	graph	12813.40	187.66	38.88	1.18	0.20	0.01
	tree	28159.80	399.11	68.04	3.03	0.18	0.00
10000	graph	85851.20	484.08	1301.97	210.82	1.60	0.16
	tree	286027.60	2172.81	4101.78	319.53	1.91	0.17

the scenario for chains of the length 25 the preparation failed on this setup whereas it was still possible to verify this using graphs with a comparable time in the previous graph experiments.

In scenario (3), depicted in table 7.4, we test the verification on another data set. This data set is a fixed ontology GALEN. Thus we change the size by increasing the number of atoms we ask traces for proofs which is depicted in the left-most column. The results of the program are computed by Nemo in around 10s on the setup used in this section. We see again that verification is very fast for trees and graphs. The difference in times between the graph and tree input is not as significant as in scenario (2). We see in the case of asking for one fact the number of nodes in the tree and graph are almost equal so atoms are not as often reused in this scenario as in the previous scenario. The difference in node numbers is larger in the case of multiple facts as multiple proof trees will use the same fact. The preparation time for graphs is in this scenario for larger instances of at least 1000 atoms much faster than for trees. We also see that a completeness check is futile here as we need a certificate for every atom while creating certificates for 10000 atoms in the graph format already takes around 20 minutes. Creating certificates for all 2.4 million facts was not possible in this setup.

In summary, we see that validating a proof tree or proof graph is possible in practice in most cases. The graph approach is significantly faster in the worst case but in most user-defined cases they are similar. The input format for the validation is however faster created for graphs due to the shape of Nemo's trace output. If the output is more tree-like then this might change. Therefore we believe that the graph input is better to use with Nemo. We also see that completeness is possible to check for small instances but takes much longer than just verifying the soundness or calculating the result itself. Our completeness check is similar to the computation that is done by a datalog engine but without the typical explanation. Therefore this behaviour is expected in larger problems. Additionally, we have never detected an error in any of Nemo's output. We note that on inputs that were built to provoke errors, these errors were found as aspected by our correctness proofs.

Chapter 8

Conclusion and further work

In this thesis, we developed and formally verified a certificate checker for datalog. For this, we formalized the syntax and semantics of datalog and formally proved that the proof-theoretic and the model-theoretic semantics are equal. The checker can check both the soundness and the completeness of a reasoning result. The soundness certificate may take two forms: a list of proof trees or a directed acyclic graph as this allows a more compact representation with the cost of a more complicated checking algorithm. We formalized a simple unification algorithm for datalog rules and a variant of depth-first search for directed graphs. Completeness of the datalog reasoning result can be checked using a certified model checker.

These algorithms are not only formally verified but can also be used in practice. The soundness checks are very fast even for larger instances and can be used in practice whereas the completeness check is only possible for small examples. This is to be expected as it is very similar to the actual reasoning done by a datalog engine. The checker is independent of any tool as long as the tool offers some sort of trace which can be transformed into proof trees or graphs.

The choice of using Lean proved to be good as the checker was practical and we did not have to worry about the conversion into the proof assistant. Lean's standard library currently lacks some proofs for the properties of operations on practical data structures such as hash maps which either requires the developer of a formally verified implementation to verify these as well or accept these as axioms for the correctness.

This is an avenue for further work as we still have some results about the correctness of hash maps as axioms. Further possible improvements for this work are an integration of the database into the verification process so that we no longer have to trust the leaves to be correct and an improved model checker.

We currently can only check a subset of the programs a modern datalog reasoner such as Nemo or Soufflé accepts. Extending the checker to more features will allow more usages in practice.

A first extension would be numbers and function terms. In many practical we may want to count the number of witnesses or use mathematical operations such as addition or multiplication. This is currently not supported by our terms and we would need to formalize datalog with function symbols and numbers starting from terms. Additionally, we would need a different unification algorithm to recognize that $R(12, 13)$ is an instance of $R(?x, ?x+1)$. We could however reuse the validation of proof trees for the most part except that we might have leaves expressing the equality of a variable to an operation on terms that we would have to check. A related extension are aggregate functions such as the sum or mean operators. Here it is more difficult because this is in a sense a global property that depends on all atoms in the interpretation. It is unclear how to express that we have used all available atoms to compute the aggregate without rerunning the complete derivation.

Datalog is a monotone language so that adding more facts to a program will still derive all the previous elements as well. This is usually not a desired behaviour in many applications. Adding a negation operator to the language will lead to non-monotone behaviour. There are multiple times of negation possible. One of them is stratified negation where all variables of a rule must occur in some non-negated atom in the rule's body. Here we can reuse the unification algorithm we defined in our work. Suppose we want to match the following stratified rules:

$$R(?x, ?y) \leftarrow S(?x, ?y, ?z), \neg T(?x, ?z).$$

$$R(a, b) \leftarrow S(a, b, c), \neg T(a, c)$$

As both rules have the same number of negated atoms we can transform them into the following rules of pure datalog:

$$R(?x, ?y) \leftarrow S(?x, ?y, ?z), T(?x, ?z).$$

$$R(a, b) \leftarrow S(a, b, c), T(a, c)$$

If a substitution exists that matches these rules then there exists also a substitution that matches the original rules. In contrast to pure datalog there stratified datalog is not defined using a proof-theoretic semantics[1]. In order to verify reasoning results of stratified datalog we need to first find an appropiate proof tree format. We need a prove that an atom in the body was not derived which is again a global property as this concerns all atoms. For this we would have to check if nothing further can be derived. In this work we used a completeness check using a model checker but this is expensive in practice.

Another extensions are existential rules. There we require the existence of elements in our model if the the body is true. If no elements exists new elements often known as nulls are added instead.

$$\exists y. Line(x, y) \leftarrow reachableByBus(x, y)$$

Such a rule is rather close to the pure datalog rules and we could reuse the unification algorithm and check if a derivation using existential rules is correct. Unfortunately, it is not that simple as in datalog. Calculating the derivations of existential rules is done with the chase algorithm. This may require to only create new arguments if no other witness for the existential quantifier exists or to first apply rules without an existential quantifier if possible. This is again a global property as we need to see the complete model (and verify that it is correctly derived) to see that a rule with an existential quantifier can be applied or that no other witness currently exists.

Bibliography

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

[2] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. A framework for the verification of certifying computations. *J. Autom. Reason.*, 52(3):241–273, 2014.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach.* Cambridge University Press, 2009.

[4] Anne Baanen, Alexander Bentkamp, Jasmin Blanchette, Johannes Hölzl, and Jannis Limperg. The hitchhiker's guide to logical verification. `https://raw.githubusercontent.com/blanchette/interactive_theorem_proving_2024/main/hitchhikers_guide_2024_lmu_desktop.pdf`, 2024.

[5] Seulkee Baek. A formally verified checker for first-order proofs. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[6] Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:2351–2375, 2005.

[7] Pierre-Léo Bégay, Pierre Crégut, and Jean-François Monin. Developing and certifying datalog optimizations in coq/mathcomp. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 163–177. ACM, 2021.

[8] Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. Certifying standard and stratified datalog inference engines in ssreflect. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2017.

[9] Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. Certified graph view maintenance with regular datalog. *Theory Pract. Log. Program.*, 18(3-4):372–389, 2018.

[10] Marco Calautti, Ester Livshits, Andreas Pieris, and Markus Schneider. The complexity of why-provenance for datalog queries. *CoRR*, abs/2303.12773, 2023.

[11] David Thrane Christiansen. Functional programming in lean. `https://lean-lang.org/functional_programming_in_lean/`, 2024.

[12] Lean community. mathlib4. `https://github.com/leanprover-community/mathlib4`, 2024.

[13] Lean community. std4. `https://github.com/leanprover/std4`, 2024.

[14] Anna Cooban. Prison. Bankruptcy. Suicide. How a software glitch and a centuries-old British company ruined lives . *CNN*, jan 2024.

[15] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[17] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[18] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[19] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: Semantics and query answering. In Diego Calvanese, Maurizio Lenzerini, and Rajeev Motwani, editors, *Database Theory — ICDT 2003*, pages 207–224, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[20] Marijn Heule, Warren A. Hunt Jr., Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2017.

[21] Martin Hils and François Loeser. *A first journey through logic*, volume 89. American Mathematical Soc., 2019.

[22] Alex Ivliev, Stefan Ellmauthaler, Lukas Gerlach, Maximilian Marx, Matthias Meißner, Simon Meusel, and Markus Krötzsch. Nemo: First glimpse of a new rule engine. In Enrico Pontelli, Stefania Costantini, Carmine Dodaro, Sarah Alice Gaggl, Roberta Calegari, Artur S. d'Avila Garcez, Francesco Fabiano, Alessandra Mileo, Alessandra Russo, and Francesca Toni, editors, *Proceedings 39th International Conference on Logic Programming, ICLP 2023, Imperial College London, UK, 9th July 2023 - 15th July 2023*, volume 385 of *EPTCS*, pages 333–335, 2023.

[23] Soonho Kong Jeremy Avigad, Leonardo de Moura and Sebastian Ullrich. Theorem proving in lean 4. `https://lean-lang.org/theorem_proving_in_lean4/title_page.html`.

[24] Mitchell Jones, Julián Mestre, and Bernhard Scholz. Towards memory-optimal schedules for SDF. In Dora Blanco Heras, Luc Bougé, Gabriele Mencagli, Emmanuel Jeannot, Rizos Sakellariou, Rosa M. Badia, Jorge G. Barbosa, Laura Ricci, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer, editors, *Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*, volume 10659 of *Lecture Notes in Computer Science*, pages 94–105. Springer, 2017.

[25] Yevgeny Kazakov, Markus Krötzsch, and Frantisek Simancik. The incredible ELK - from polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. *J. Autom. Reason.*, 53(1):1–61, 2014.

[26] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, jul 2009.

[27] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[28] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011.

[29] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[30] Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *The Journal of Symbolic Logic*, 33(4):514–520, 1968.

[31] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data,*

SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 1135–1149. ACM, 2016.

[32] Niels van der Weide, Deivid Vale, and Cynthia Kop. Certifying Higher-Order Polynomial Interpretations. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:20, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[33] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.