LEAN-aided verification of datalog reasoning results

Johannes Tantow

January 19, 2024

1 Introduction

2 Preliminaries

2.1 Datalog

Syntactically datalog bears many similarities to first-order logic and may be viewed as a subset of it, but the semantics are more specialized. In order to define the syntax we will introduce some definitions from first-order logic.

Firstly, we consider in general three sets R, that consists of relation symbols, C, that consists of constants, and V, that consists of variables. In order to differentiate between constants and variables we require that $V \cap C = \emptyset$. Each relation symbol has an associated arity $ar : R \rightarrow \mathbb{N}$.

A *term* is either a constant or variable. In contrast to first-order logic or logic programming, we do not allow function symbols in datalog, but there are extensions that introduce some function symbols like addition or multiplication. Assuming that the set of constants is finite, the Herbrand universe will later be finite as well.

An *atom* is of the form $A(t_1, ..., t_{ar(A)})$ where A is a relation symbol and $t_1, ..., t_{ar(A)}$ are terms. The atom variables of $A(t_1, ..., t_{ar(A)})$ are the set of those t_i which are variables. A *ground atom* is an atom where the atom variables are the empty set, i.e. every term is a constant.

A *rule* is a expression of the form $H \leftarrow B_1 \wedge ... \wedge B_n$, where H and all B_i are atoms. H is called the head of the rule and $\{B_1, .., B_n\}$ is the body of the rule. If H and all B_i are ground atoms, then we call this rule a *ground rule*. If the body is empty, we call a rule a *fact*. A rule is called *safe*, if every variables in the atom variables of the head is also in the atom variables of some atom in the body.

A *program* is a finite set of rules. The program is safe, if all its rules are.

A datalog engine calculates the semantics of a datalog program, which we will define shortly. The non-fact rules of program are often general rules that can be used in many contexts, whereas the facts encode a specific context. In order to reuse programs, the facts are often outsourced in database, which we can imagine as a set of ground atoms. In the semantics we assume that these facts are then added back to the program for easier definitions.

Datalog has three major semantics: model-theoretic, proof theoretic and fixed-point based. These three semantics coincide. The semantics of a datalog program $P = \{r_1, ..., r_k\}$ is a set of ground atoms.

For the model-theoretic semantics we describe the semantics as a specific model of the program. So far P might however contain variables, which have to be mapped to something. Therefore we first transform P into the *ground program*, $\mathit{ground}(P)$. For this we consider *groundings* or *instantiations*, which are functions from V to C. We can apply a grounding g to an atom a to aquire a ground atom by replacing a variable v with $g(v)$. We may write this also as $g(a)$ and can apply grounding to a rule r as well by applying it to every atom in the body and the head, which we denote as $g(r)$. The ground program is then defined as $\mathit{ground}(P) = \{r' \mid \exists r \in P. \exists g. g(r) = r'\}$

A set of ground atoms I (*interpretation*) is a *model* of a ground rule r, if whenever the body of r is a subset of I also the head appears inside I. An interpretation is a model of a ground program P if it is all model for all ground rules in P.

We would like to define the model-theoretic semantics of P as the model of $\mathit{ground}(P)$. In general there are however multiple models of $\mathit{ground}(P)$, so that this would not be unambiguous. It might contain ground atoms without a ground rule having this atom as a head. It can however be proven that there always exists a minimal model of $\mathit{ground}(P)$. We define this as the model-theoretic semantics of P

A different view is to look why an atom a is inside the minimal model. It might be because there was a ground rule in $\mathit{ground}(P)$ of which a is the head and for which the body is already in the minimal model. Now we can again ask this for all elements in the body until we reach facts, which have to be true. This can be seen as a proof for a. This proof can be modelled as a rooted tree which nodes are ground atoms. A tree is a *valid proof tree* for a in P if the following conditions hold.

1. The root is a
2. For each leaf exists a fact in $\mathit{ground}(P)$
3. For each node n and its predecessors $m_1, ..., m_k$ exists a rule in $\mathit{ground}(P)$ which the head n and the body $\{m_1, ..., m_k\}$

The proof-theoretic semantics is then the set of ground atoms for which a valid proof tree in P exists.

The previous two semantics both provide clear definitions, but do not tell us how to arrive at this set. The fixed point semantics offers way to do this. For this we consider the following operator, that takes a set of ground atoms as an input.

$$T_P(I) = \{a \mid \exists r \in \mathit{ground}(P), \mathit{head}(r) = a \wedge \mathit{body}(r) \subseteq I\}$$

This operator can be proven to be monotonic and therefore a least fixed-point exists by the Knaster-Tarski theorem exists. It can be computed by starting with $\emptyset$ for I and repeatedly apply T_P until we find the fixed-point. This semantics is the basis for the practical implementations of datalog engines.

All three semantics can be proven to be equivalent, which we will do later in Lean for the proof-theoretic and model-theoretic semantics.

Additionally, it turns out that every non-safe program can be transformed into a safe program by introducing new relation symbols, such that for all original relation symbols the semantics coincide.

3 Datalog in Lean

We introduced Datalog in the previous chapter. Here we show our implementation in Lean that we will use later to show the correctness of our algorithms.

We started defining datalog using sets for constants, variables and relation symbols. In order to have a more compact representation we use the idea of a signature from logic, that stores all these informations.

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

We no longer require that the constants and variables are distinct sets as the constructor for terms allows us to identify the type of a term. We will continue to define the other syntactic elements of datalog. The signature unless displayed otherwise will be τ.

```
inductive term :
| constant : τ.constants → term τ
| variableDL : τ.vars → term τ
deriving DecidableEq
```

As we want to use these objects in practical algorithms we need a way to see if two terms are the same to e.g. replace a variable by a constant. We use Leans ability to automatically create a decidable function for equality for our inductive types and structure from the given functions for the equality of the types from the signature.

For atoms we have to make sure that the amount of terms matches the arity of the relation symbol. Therefore the structure has an additonal element that is a proof of this fact. Two atoms are equal if all fields in the structure are equal. Since propositions as this proof are always equal, it is enough to show that the terms and the symbols are equal.

```
structure atom where
```

```
    (symbol: τ.relationSymbols)
    (atom_terms: List (term τ ))
    (term_length: atom_terms.length = τ.relationArity symbol)
deriving DecidableEq

lemma atomEquality (a1 a2: atom τ):
a1 = a2 ↔ a1.symbol = a2.symbol ∧ a1.atom_terms = a2.atom_terms
```

Rules and programs can be expressed relatively straight forward. For programs we use `abbrev`, which introduces program as an alias for `Finset(rule τ)`, instead of defining a new type as this allows us to use the common set operations without the need to redefine them.

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

abbrev program := Finset (rule τ)
```

We have previously introduced ground atoms as atoms whose terms are only constants. There are at least two options to represent them here. Firstly we could define a structure for ground atoms that extends atoms by a predicate that shows that there are only constants present. The second option is to define an entirely new structure similar to atoms but allow only constants instead of terms. The first options show directly that every ground atom is an atom which we would have to establish separatedly in the second option. The second option allows directly to define the functions on a list of constants instead of having to cast it directly to it and allows to use the complete induction schema when defining functions on ground atoms. The second option seemed to offer more advantegous and was therefore chosen.

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

Now we need a function that maps `groundAtom` back to `atom`. For this we can map the `atom_terms` list in `groundAtom` with `term.constant` to term. Additionally we need to show that the `term_length` property is preserved. Luckily, this is already proven in mathlib.

```
lemma listMapPreservesTermLength (ga: groundAtom τ):
(List.map term.constant ga.atom_terms).length =
 τ.relationArity ga.symbol :=
by
    rw [List.length_map]
    apply ga.term_length

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

Additionally, we need later that two `groundAtom` are equal iff the result of `toAtom` is equal.

Similarly, we can define ground rules and functions to convert them to rules.

We formalize two semantics here. Firstly, we need the proof-theoretic semantics. Proof trees are a good way to show why an element must be in the solution and output already by multiple datalog engines. There is however as our current knowledge no short certificate similar to proof trees to show the absence of elements. In order to show that a solution is complete we need another method. Both the fixed-point and the model-theoretic semantics offer help in this case. Both are the least element of some set (of fixed-point or of models respectively). Therefore it is enough to show that it is a member of this set for the other direction.

For the semantics we need the concept of the database. We do not want to deal with the implementation details of a database. We consider it as a black box that returns a ground atom whether it is in the database. It is a function to Bool, because we want computable databases.

```
class database (τ: signature)
[DecidableEq τ.vars] [DecidableEq τ.relationSymbols]
[DecidableEq τ.constants]:=
(contains: groundAtom τ → Bool)
```

We start by defining the proof-theoretic semantics. For this we need to formalize trees. As the size of the body may be unbounded we represent the children using a list.

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

In contrast to binary trees, we will need to use functions for lists to define functions on trees. Most interesting function will however need a proof of termination due to this tree model.

```
def root: tree A → A
```

```
| tree.node a _ => a

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

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

Height is recursively called on the elements of in the list via the listMax function. Lean can not identify that this terminates so we have to do this. We use the `sizeOf` function that is built in for any inductive type and maps an element to the sucessor of the sum of the `sizeOf` values of the elements used in the sucessor. If we show that this always decreases then the function will terminate as the natural numbers are well-founded.

In contrast to just using the list, we call listMax on `List.attach` instead. `List.attach` takes a list l and creates a new list consisting of pairs of the original elements and a proof that the element is in l. This proof can be later used when proving termination.

We have to show a statement of the form $\texttt{sizeOf} x < 1 + \texttt{sizeOf} a + \texttt{sizeOf} l$. We use the transitivity property of the linear order to show this via showing first that $\texttt{sizeOf} x < \texttt{sizeOf} l$ and secondly that $\texttt{sizeOf} l < 1 + \texttt{sizeOf} a + \texttt{sizeOf} l$. This second statement is obviously true and can be found by `simp`. For the first statement we can use the fact that the size of a member of a list is smaller than the size of the list. Due to using `l.attach` we have a proof for this available and finish the proof. This way of proving termination is the standard way we use in this model and will be omitted for further tree functions.

Proof trees are then trees that have `groundAtom` as it vertices and we can formulate the validness criteria. There are two cases. If we are at leaf, i.e. the list of children is empty, then we simply check the database. Else there must exist a rule r from the program and a grounding g so that applying g to r matches the rule created from the current node and its children. Additionally all child trees must be valid as well.

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

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)
```

The proof-theoretic semantics is then simply the set of ground atoms that are the root of a valid proof tree.

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

After that we define the model-theoretic semantics. We choose the model-theoretic semantics over the fixed points semantics as we can directly construct the model and do not have to show that a fixed-point exists. Checking them in practice seems to be similar.

An interpretation is a set of ground atoms, that may be a model. A model of a program over a database is an interpretation that contains every database element and fulfills every rule. Choosing the database to be a part of the model simiplifies grounding a bit later, as there is only one place to check for matches. Additionally, we want to show that both semantics are equal. From the definition above one can see that any database element has a valid proof tree and is therefore in the proof-theoretic semantics.

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

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

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

The usual way of defining the model-theoretic semantics is via the model intersection property. If we have two models for a datalog program, then their intersection is again a model. Intersecting all models yields therefore the minimal model. In order to use $\bigcap$ in lead we would need to transform the set of models firstly into an indexed set. We instead used the following more simple notion. We know that an element a is a member of $X \cap Y$ iff a is a member of X and a member of Y. Consequently, a is a member of $\bigcap_{M \in models(P,d)} M$, if it is a member in all models.

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

This describes the same set, but does not offer yet the minimal model property, which we still have to prove.

We start by showing that it is a subset of every model.

```
lemma leastModel (P: program τ) (d: database τ)
(i: interpretation τ) (m: model P d i):
modelTheoreticSemantics P d ⊆ i :=
by
    unfold modelTheoreticSemantics
    rw [Set.subset_def]
    intro a
    rw [Set.mem_setOf]
    intro h
    apply h
    apply m
```

This follows quickly from the definition. We have to show that if some arbitrary ground atom a is in the model-theoretic semantics then it is in i from the subset definition. a is in the model-theoretic semantics, if it is in every interpretation that is a model. Specifically, it must be a member of i, which is a model by assumption.

While the previous lemma is called leastModel, we still have to show that it is a model.

```
lemma modelTheoreticSemanticsIsModel (P: program τ)
    (d: database τ):
model P d (modelTheoreticSemantics P d)
```

Lemma 1. *Let P be a program and d be a database.*

Then the `modelTheoreticSemantics` of P over d is a model for P over d.

Proof. We show that both conditions hold and start by showing that any rule must be rule. Let r be a arbitrary rule the ground program of P. Assume that the body of r is a subset of the `modelTheoreticSemantics`. If not, then the rule is already true since the body is not true. We assume for a contradiction that the head of the rule is not in the `modelTheoreticSemantics`. From the definition we know that there must exists a model i, which does not contain the rule head. We do know however that the body is a subset of the `modelTheoreticSemantics` and therefore also a subset of i. Then the rule r would not be true and i would not be a model and we have reached a contradiction. □

After defining both semantics we finally want to show that they are equal, i.e. the following theorem.

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

This is supposed to be done via the anti-symmetric property of the subset relation, i.e. $\forall X, Y. X \subseteq Y \land Y \subseteq X \to X = Y$.

First we want to show that the proof-theoretic semantics are a subset of the model-theoretic semantics. We show the bit stronger statement, that all element in the proof-theoretic semantics are in every model. As the model-theoretic semantics are a model, the first direction follows.

Lemma 2. *Let P be a program, d be a database and i a model of P over d. Then `proofTheoreticSemantics P d` is a subset of i*

Proof. We want to show that for every ground atom a from the proof-theoretic semantics and valid tree t with the root a a is a member of i via strong induction on the height of t.

Due to using strong induction, we do not need a induction base and can directly prove the induction step. □

For the second direction, we need to show that the `modelTheoreticSemantics` is a subset of the `proofTheoreticSemantics`. As we know that the model-theoretic semantic is the least model it suffices to show that the proof-theoretic semantic is a model as well.

Lemma 3. *Let P be a program and d a database.*
Then `proofTheoreticSemantics P d` is a model.

Proof. We have to show that the conditions for a model hold.

Firstly, that the database is a subset of the proof-theoretic semantics. For every ground atom in the database we can construct a tree with no children that has only this element as the node. This is valid and has the database element as its head, so that the condition is fulfilled.

Secondly, we have to show that every rule is fulfilled. Let r be a ground rule from $ground(P)$ and assume that the body of r is a subset of proof-theoretic semantics of P over d. Then we have to show that the head of r is also in the proof-theoretic semantics of P over d.

Since

□

4 Soundness

After introducing the problem and modelling in Lean, we now describe the algorithm to verify a solution. In this chapter we deal with the soundness, which means here that every atom in the interpretation is actually in the semantics. For this we use the proof trees as the certificate and the proof theoretic semantics.

A ground atom is in the proof theoretic semantics if there exists a valid proof tree, that has this ground atom as its head. We were provided with all the proof trees and checking the heads is rather easy, so what remains to be checked is the validness of a proof tree, that was defined in the previous chapter.

```
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)
```

The second part of this disjunction consists of a database check and an easy check of list emptyness. The first part is more interesting. Since we use there existential quantifiers, we have to implement something to check this. As the program is given as a list of rules, we can simply iterate over this list. For the grounding we however can something more sophisticated, but groundings are not our object of choice for that.

4.1 Substitutions

A grounding is function from variables to constants. This mean that 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 in the unification case problems as the following example demonstrates.

Example 1. *Consider the signature consisting of $C = \{a, b, c\}$, $V = \{x, y, z\}$ and $R = \{R\}$. Suppose we want to match a list of terms with a list of constant. The first term is $t_1 = x$ and the first constant is a. We might use the the grounding $g = x \mapsto a, y \mapsto a, z \mapsto a$.*

Now we want to use this result and match another term $t_2 = y$ 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.

Instead, we want to use substitutions that were already introduced in [**?**]. 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. If we apply a substitution to a term, we only replace a variable by a constant, if the substitution is defined for this variable and the constant will be the result of the substitution in this case.

```
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
```

We can use similar defintions as previously for groundings to apply substitutions to atoms or rules.

The main result we want to prove is the following.

```
theorem groundingSubstitutionEquivalence
    [Nonempty τ.constants] (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 by 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 implication, we can transform any grounding in a simple way to a substitution. In this substitution every value is defined with the value of the grounding.

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

It is very easy to prove that this is equivalent on every rule.

For the back direction, we need additionally that the set of constants is nonempty. We can ensure this during the input phase by adding a fresh constant symbol to the constant symbols similar to Herbrand universes. This symbol does not appear in any proof trees and does not influence the results. Since we only look at safe programs, it will also not introduce any new ground atoms to the model.

The following example shows the problems that occur without the nonemptyness assured.

Example 2. *Consider the program $P = \{p \leftarrow, q \leftarrow p\}$ and the signature $C = \emptyset$, $V = \{x, y, z\}$ and $R = \{p, q\}$*

Any rule in 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.

Since the set of constants is non-empty, we can use the axiom of choice to get values for which the substitution is not defined.

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

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 linear relation on substitutions, that is denoted by $\subseteq$

Firstly, we define the substitution domain of a substitution as the set of variables for which the substitution is defined.

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

A substitution s_1 is then 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 that 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 can be proven to be a linear 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 node of the proof tree is the substituted rule of some rule of the program. For this we take inspiration from the unification problem of first-order logic.

In the unification problem we are given a set of equations between first-order terms and are required to present the most-general unifier.

Our problem is similar. The equations will not be between terms, but between an object and a ground object of the same corrosponding type and we require a substitution that solves all equations and is minimal in our subset relation.

An algorithm to solve the first-order unfication problem is the algorithm of Martelli and Montanari [**?**] and is depicted below:

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
```

This algorithm offers a good starting point for our own algorithm, but we certain transformation can't occur in the limited syntactic form we operate in. Additionally, we want to output a substitution instead of just answering whether a substitution exists. It is sufficent 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 $x \mapsto t$ 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. As function symbols appart from constant symbols are not allowed, we can simplify steps 2 and 3, as we never add new equations and instead only check if the constant symbol matches. Finally, as the one side of the equation is always a ground object there will never be a variable on this side, so that we do not have to swap the equation as in step 4.

We will start with 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 substitution s' so that $s \subseteq s'$ and applying s' to t will make it equal to c or none if no such s' exists.

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 know formally prove the correctness of this algorithm.

Lemma 4. *Let t be a term, c a constant, s a substitution and let matchTerm t c s return a new substitution s'. Then $s't = c$ and $s \subseteq s'$*

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 linear 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 be in violation to 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 v was not defined earlier, for any variable in the domain of s, s and extend s v c, so that it is fulfilled. □

We have proven so far the matchTerm returns a solution, but it might not be a minimal solution. This is however later needed, when we want to match atoms to ground atoms as there we match a list of terms to a list of constants and pass the previous results on.

Lemma 5. *Let t be a term, c a constant, s a substitution and let matchTerm t c s return a new substitution s'. Then s' is a minimal solution, i.e. forall substitution s^* with $s \subseteq s^* \wedge s^*t = c$ we have $s' \subseteq s^*$*

Proof. This is again done via case distinction on the type of t. If t is constant, then s' must be equal to s. For any s^* with $s \subseteq s^* \wedge s^*t = c$ we have b that $s' \subseteq s^*$ by the assumption of the property of s^*

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 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. We additionally have to prove that if matchTerm does not return a substitution that then no solution exists.

Lemma 6. *Let t be a term, c a constant, s a substitution and let matchTerm t c s return none. Then there does not exists a substitution s' with $s \subseteq s'$ and $s't = c$.*

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 s' can exist. If s would be a subset of s', then s' would not unify t with c and if s' would unify t with c then s would not be a subset of s'. □

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. 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 => some s
| 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. For the correctness it is required that both lists have the same length, but this is the case for atoms. If the first list is empty we return the substitution. If instead it has a first element and the second list has as well a first element, we match these elements using matchTerm and if this results in a solution, we return the result of matchTermList with the remaining lists and the resulting substitution.

As previously, we have to prove the correctness of this algorithm. We again want to show that the algorithm returns the minimal solution iff it exists.

Lemma 7. *Let l_1 be a list of terms, l_2 be a lists of constants with the same length as l_1 and let s be a substitution. If matchTermList s $l1$ $l2$ returns a substitution s', then $s \subseteq s'$ and applying s' to every element in l_1 results in both lists being equal.*

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 both lists have the same length l_2 must also be the empty list. matchTermList then returns s. Applying this to an empty list returns an empty list. Therefore the base case is complete.

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 $hd' :: tl'$ and that tl and tl' have the same length by our assumption. 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'. By the induction hypothesis $s^* \subseteq s'$ and applying s' to tl results in it being equal to tl'. To show that both lists are equal, we just have to show that after applying s' the heads will be equal. We know that applying s^* to hd will result in it being equal to hd'. Since $s^* \subseteq s'$ and our previous result that if a substitution maps a term to a constant then extension of this substitution will also map the term to the same constant, we have this. Lastly, 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. □

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

Lemma 8. *Let l_1 be a list of terms, l_2 be a lists of constants with the same length as l_1 and let s be a substitution. If matchTermList s $l1$ $l2$ returns a substitution s', then for all substitutions $\hat{s}$ that satisfy $s \subseteq \hat{s}$ and that after the application of $\hat{s}$ $l1$ will be equal to $l2$, we have that $s' \subseteq \hat{s}$*

Proof. We prove this again via induction on l_1 for arbitrary l_2 and s. If l_1 is empty, we return s and the claim is true by assumption.

In the induction step we have that l_1 has the form $hd :: tl$ and we can assume that l_2 has the form $hd' :: tl'$ and that tl and tl' have the same length. Additionally, we know that matchTerm $hd\ hd'$ s returns a substitution s^*. From the induction hypothesis we know that matchTermList s^* tl tl' $\subseteq \hat{s}$ if □

Finally the negative case.

Lemma 9. *Let l_1 be a list of terms, l_2 be a lists of constants with the same length as l_1 and let s be a substitution. If matchTermList s l1 l2 returns none, then there does not exist a substitution s' with $s \subseteq s'$ and that has that property that applying s' to l_1 will result in it being equal to l_2.*

Proof. We prove this again via induction on l_1 for arbitrary l_2 and s. If l_1 is the empty list, we cannot return none. As the prerequisite is not fulfilled, the statement is correct.

In the induction step we have that l_1 has the form $hd :: tl$ and we can assume that l_2 has the form $hd' :: tl'$ and that tl and tl' have the same length. 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 here. □

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 also get that their term lists must be equal as the length of the term list is equal to the arity of the relation symbol. The same symbol implies the same length. If the symbols are different, we will never unify them.

```
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 since we already established the same length of both lists from the symbol.

Now we can again create a matchAtomList function and finally a matchRule function. The correctness proofs are of the same form as for matchAtom and are therefore omitted.

The final result is the following:

```
theorem matchRuleIsSomeIffSolution (r: rule τ) (gr: groundRule τ)
(len: r.body.length = gr.body.length):
Option.isSome (matchRule r gr) ↔
∃ (s: substitution τ), applySubstitutionRule s r = gr
```

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

4.3 Tree validation

5 Completeness