LEAN-aided verification of datalog reasoning results

Johannes Tantow

December 29, 2023

1 Introduction

2 Preliminaries

todo: - define safeness/propositional programs

3 Datalog in Lean

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. Recall the defintion of validness we established earlier:

```
    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 non-empty. 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 non-emptyness 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 1. *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 2. *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 substituion that then no solution exists.

Lemma 3. *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'. □

4.3 Tree validation

5 Completeness