LEAN-aided verification of datalog reasoning results

Johannes Tantow

December 28, 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

4.3 Tree validation

5 Completeness