Constructive Logic (15-317), Spring 2022
Assignment 9: Simple Prolog Interpreter

Instructor: Klaas Pruiksma
TAs: Runming Li, Onyekachi Onyeador, Viraj Puri, Xiao Yu

Submit to Gradescope by Thursday, March 31, 2022, 11:59 pm

For this homework, you will be submitting only coding solutions to Gradescope:

- `hw9.sml` (your coding solutions to task 1 and task 2)

1 Implementing a simple Prolog Interpreter

We saw in class how Prolog, a standard backward-chaining logic programming language, uses the idea of Backtracking and Unification. In this homework, you will implement a simple Prolog interpreter in SML that combines unification with backtracking to find proper substitutions for terms in goals if provable.

For this homework, we provide you with four datatypes in `Unification.sml`:

- `datatype term = Var of string | Pred of string * term list`
  `Var(s)` represents a variable whose name is the string `s`, and `Pred(s, ts)` represents a predicate whose name is the string `s` with `ts` being the arguments. A constant can be represented as a zero-argument predicate `Pred (a, [])`.
- `datatype constraint = Eq of term * term`
  `Eq (t1, t2)` represents the constraint that terms t1 and t2 are equal.
- `datatype rule = Rule of term * term list`
  `Rule(t, ts)` represents a rule whose conclusion is `t`, and whose premises are the list `ts`. `ts` may be empty if there's no premise.
- `datatype unifier = VarEq of string * term`
  `VarEq (s, t)` represents the substitution setting the variable with name `s` equal to `t`.

Task 1. Implement the function `unify:constraint list -> unifier list option` that returns `SOME unfl` where `unfl` is a substitution that satisfies all constraints in the constraint list, or `NONE` if no such substitution exists. For example, some possible solutions for `unify [Eq(Var "a", Var "b")]` include:

- `SOME [VarEq ("a",Var "b")]`
- `SOME [VarEq ("b",Var "a")]`
- `SOME [VarEq("a", Var "new_var_name"), VarEq("b", Var "new_var_name")]`

Notes and Hints:

- A substitution `VarEq("s", t)` satisfies a constraint `Eq(t1, t2)` if t1 and t2 are equal after substituting the term t for any instances of `Var "s"` in t1 and t2.

- For this task, you can assume that all predicates are different i.e., no substitution satisfies the constraint `Eq(Pred(s1, tl1), Pred(s2, tl2))` if `s1` $\neq$ `s2`.
- In lecture, we learnt that Prologs algorithm for unification is unsound. In order to build a sound unification, you need to check for cyclic terms. For example,`unify ([Eq(Var "a", Pred("Sum", [Var "a", Var "b"]))])` should return `NONE`.
- We provide you with helper functions `subst_terms:unifier -> term list -> term list` and `subst_all:unifier list -> term list -> term list` in `Unification.sml` that implement substitution on terms. Given an unifier `VarEq(s,t)` and a term list `ts`, `subst_terms VarEq(s, t) ts` replaces all occurrences of `Var "s"` in `ts` with the term `t`. You might need to write more helper functions with similar behavior for other datatypes.
- Redundant solutions are acceptable - e.g., your unifier list can have both `VarEq("x", Var "y")` and `VarEq("y", Var "z")` if you want to substitute "x" with `Var "z"`.

Task 2. Implement the function `solve:rule list -> term list -> unifier list option` so that `solve rules goals` returns `SOME unfl` where `unfl` is a substitution that, when applied to the terms in goals, makes each of them provable from the list of rules, and `NONE` if no such substitution exists.

Notes and Hints:

- A simple example: Suppose we have two rules for equality — in Prolog syntax:

```
Eq(x, y, z) :- Eq(x, y), Eq(y, z).
Eq(x, x).
```

  Note that these rules define two different predicates `Eq/2` and `Eq/3`. Given `[Eq(a,b,c)]` as our goal list, one solution would be to substitute each of a, b, c with `Var v` for any choice of v.
- Problems from lecture, Homework 8, and recitation may be good sources for test cases. Note that your interpreter will not be as fully-featured as Prolog, however, so, for instance, `\=` and `\+` will not work.
- You may find the `unify` function from task 1 very useful. Think about what the constraint(s) should be when you try to apply unification on a goal using a specific rule.
- In order to make rules generic and avoid collisions of variables across multiple instances of a rule, we provide you with a helper function `fresh_rule:rule -> rule` that takes a rule and produces a copy of it with all of its variables replaced by fresh variables. Remember to create a fresh copy of a rule when you try to apply it.
- Similar to task 1, redundant solutions are acceptable. In addition, your unifier list can contain extra information (substitution for terms that don't appear in the goals). This is because you might have substitutions for terms in rules when you try to apply the rules.
- Implementing a function that finds substitution for terms given a rule list is similar to implementing a proof search function. As such, we highly suggest using continuation passing style (CPS) to simplify your code. For a refresher on CPS, please feel free to look at the 15-150 notes on the topic.