Functional Programming
lab session 3b

Lab session 3 has been split in two parts: lab 3a and lab 3b. The objective of lab 3 is to implement an interpreter for a (very) small subset of the Prolog Programming language. We will call this language `uProlog` (micro-Prolog).

The interpreter consists of two major parts. The first part, the so-called *front-end* of the interpreter parses the input (a program), while the second part (called the *back-end*) performs the actual execution.

In lab 3a the focus was solely on parsing the `uProlog` language in the front-end. In lab 3b we will extend the program with a back-end. In case you are completely satisfied with the front-end that you produced in lab 3a (and did not make changes to the prescribed data types) then you can use this code as the basis for lab 3b. However, on Nestor a complete solution for lab 3a is available that you can use as the basis for lab 3b. In either case, there is also a module `Analysis.hs` which performs some semantic checking on a `uProlog` program (e.g. checking arguments of functions) and a driver (main) module `uProlog.hs` which contains a main function.

Introduction: Logical Inference

Before we start lab 3b it is important to understand some theoretical aspects about logical reasoning using an automatic inference technique called *resolution*. You may have encountered this topic before in a course like *Artificial Intelligence I*, but it won't hurt to recap the topic.

Most general Unifiers (MGU)

We start by introducing the notion of *unification*. The unification process in the context of First Order Logic (FOL) is more general than explained here, but this stripped explanation is sufficient to make lab 3b.

We consider (as an example) two predicates (i.e. logical expressions with arguments) $\alpha =$ `child(X,Y)` and $\beta =$ `child(john,Y)`.

A *subsitution* is a list of pairs. Each pair consists of a variable (starts with an upper case letter) and a literal (variable name that starts with an upper case letters or an object name that starts with a lower case letter). Three examples of substitutions are $\theta_0 = [(\texttt{X}, \texttt{john}), (\texttt{Y}, \texttt{sue})]$, $\theta_1 = [(\texttt{Y}, \texttt{sue})]$, and $\theta_2 = [\texttt{(Y,B)}]$. The function `subst` takes two arguments: a substitution and a predicate. The result is a new predicate that is obtained by replacing each variable in the original predicate by the corresponding literal from the substitution. Hence, applying `subst` to a predicate is a purely syntactic operation. It is defined as follows:

$$\begin{aligned} \texttt{subst []}\ \alpha &= \alpha \\ \texttt{subst ((X,E):}\theta\texttt{)}\ \alpha &= \texttt{subst}\ \theta\ \alpha_E^X \end{aligned}$$

Here, the notation α_E^X means "replace each occurrence of `X` in α by `E`". As a result:

$$\begin{aligned} \texttt{subst}(\theta_0, \alpha) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_1, \alpha) &= \texttt{child(X,sue)} \\ \texttt{subst}(\theta_2, \alpha) &= \texttt{child(X,B)} \\ \texttt{subst}(\theta_0, \beta) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_1, \beta) &= \texttt{child(john,sue)} \\ \texttt{subst}(\theta_2, \beta) &= \texttt{child(john,B)} \end{aligned}$$

A *unifier* is a substitution that *unifies* two predicates (i.e. makes them equal). From the above examples, it is clear that θ_0 is a unifier for the predicates α and β because $\texttt{subst}(\theta_0, \alpha) = \texttt{subst}(\theta_0, \beta)$. Here 'equality' means syntactically the same expression.

Given two predicates, there might exist more than one unifier. For example, we can unify the predicates `f(A,B,c,D,E)` and `f(B,a,C,c,F)` using the unifiers $\theta_0 = [(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})]$ and $\theta_1 = [(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{a}), (\texttt{F}, \texttt{a})]$. In fact, there exist more unifiers for this example. However, there is only one single unifier that we call the *most general unifier (MGU)*, which is in this case θ_0.

A *most general unifier* is a unifier that consists of a minimal number of subtitutions. Hence, it transforms the two predicates in the most general unified expressions.

$$\begin{aligned} \texttt{f(a,a,c,c,F)} &= \texttt{subst}([(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})],\ \texttt{f(A,B,c,D,E)}) \\ &= \texttt{subst}([(\texttt{A}, \texttt{a}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})],\ \texttt{f(B,a,C,c,F)}) \end{aligned}$$

The unifier θ_1 would unify the predicates to predicate `f(a,a,c,c,a)` which is clearly less general, because an explicit value for the last argument of `f` was substituted.

It is well known that the most general unifier of two predicates is unique upto renaming of (some) variables. For example, in the MGU θ_0 we rename the variable E to F, but we could just as well decide to rename F to E.

Note that we could have expressed the unifier θ_0 as:

$$\theta_0' = [(\texttt{A}, \texttt{B}), (\texttt{B}, \texttt{a}), (\texttt{C}, \texttt{c}), (\texttt{D}, \texttt{c}), (\texttt{E}, \texttt{F})]$$

We consider θ_0' to be the same unifier as θ_0, since $\texttt{subst}\ \theta_0\ \phi = \texttt{subst}\ \theta_0'\ \phi$ for any predicate ϕ.

Clausal Form

A *clause* is a disjuntion of predicates. For example, the predicate $f(X, Y) \vee g(X) \vee \neg h(Y)$ is a clause. Because there is only one connective ($\vee$), clauses can be represented as a list of predicates. Actually, the list is a set since the order of the elements in a clause is irrelevant, and there are no duplicates in a clause. For the given example this list would be the list $[f(X, Y), g(X), \neg h(Y)]$.

Because a `uProlog` program (or a Prolog program in general) is a collection of facts and implications (rules), we can easily transform the program into a set of clauses: facts are transformed into singleton clauses, while rules are transformed using the fact that $\alpha \Rightarrow \beta$ is equivalent with $\neg\alpha \vee \beta$.

For example, in the following `uProlog` program fragment this translation in clausal form is given in a comment for program line (where `~` denotes negation):

```
child(sue,john).                    % [child(sue,john)]
child(sam,john).                    % [child(sam,john)]
child(sue,jane).                    % [child(sue,jane)]
male(john).                         % [male(john)]
parent(Y,X) :- child(X,Y).          % [~child(X,Y),parent(Y,X)]
father(A,B) :- male(A),child(B,A).  % [~male(A),~child(B,A),father(A,B)]
```

Inference by Resolution

The logical inferences that the `uProlog` interpreter will make is based on one single inference rule, called the *resolution rule*. Let α and β be atomic predicates. Let γ, δ be atomic predicates or disjunctions of predicates. Also, let θ be a most general unifier for α and β. Then the resolution rule states:

$$\frac{\alpha \vee \gamma \quad \neg\beta \vee \delta}{\texttt{subst}\ \theta\ (\gamma \vee \delta)}$$

The conclusion $\texttt{subst}\ \theta\ (\gamma \vee \delta)$ is called the *resolvent*.

The resolution rule is sound because if we apply the unifier θ to $\alpha \vee \gamma$, then we obtain $\texttt{subst}\ \theta\ \alpha \vee \texttt{subst}\ \theta\ \gamma$, which is logically equivalent to $\neg\texttt{subst}\ \theta\ \alpha \Rightarrow \texttt{subst}\ \theta\ \gamma$. Similary, we can apply the unifier to $\neg\beta \vee \delta$ to

obtain `subst` θ $\beta \Rightarrow$ `subst` θ δ. Because θ is a unifier for α and β, we can replace `subst` θ β by `subst` θ α to obtain `subst` θ $\alpha \Rightarrow$ `subst` θ δ. Since `subst` θ α is either true or false, we conclude `subst` θ $\gamma \vee$ `subst` θ δ which equals `subst` θ $(\gamma \vee \delta)$.

The conclusion is that we can use the resolution rule to make inferences in `uProlog` programs as follows. Consider, the clauses `[male(john)]` and `[~male(A),~child(B,A),father(A,B)]`. We can unify the complementary terms `male(john)` and `male(A)` using the most (and only) general unifier `[(A,john)]`. Hence, using the resolution rule we infer the resolvent (i.e. a new clause) `[~child(B,john),father(john,B)]`. Next, we apply resolution again on this new clause together with the clause `[child(sue,john)]` (using the mgu `[(B,sue)`) to infer `[father(john,sue)]`.

An imperative style pseudo-code of an algorithm that infers all possible clauses by repeated application of the resolution rule is given below. It is the driving heart of the `uProlog` interpreter.

```
clauses := list of clauses representing the uProlog program
continue := true
while continue do {
  inferences := empty set
  for each pair C0,C1 from clauses do {
    inferences := union(inferences,
                        clauses that can be inferred by applying the resolution rule to C0 and C1)
  }
  if inferences is a subset of clauses
  then continue := false
  else clauses := union(clauses,inferences)
}
```

Lab Problems

We start by extending the file Types.hs with a few types (already done for you on Nestor):

```
type Substitution = (String,Argument)
type Unifier = [Substitution]
type Clause = [(FuncApplication,Bool)]
type Clauses = [Clause]
```

The type `Substitution` represents a substitution of a variable (first element of the pair) by a literal (a constant object or variable). For example, the substitution `(X,john)` is represented by `("X",Const "john")` (which is of the type `Substitution`). A renaming of a variable like `(F,E)` is represented by `("F",Arg "E")`.

A unifier is represented by a list of `Substitution`s, so the unifier `[(X,john),(F,E)]` is represented by `[("X",Const "john"),("F",Arg "E")]`.

A clause is represented by the data type `Clause`, which is a list of pairs. The first element of the pairs is a function application (i..e a predicate), while the second is a boolean value. This boolean is `True` for a positive predicate and `False` for a negated predicate. For example, the clause `[~male(A),~child(B,A),father(A,B)]` is represented by `[(FuncApp "male" [Arg "A"],False),(FuncApp "child" [Arg "B",Arg "A"],False),((FuncApp "father" [Arg "A",Arg "B"],True)]`. On its turn, a set of clauses is represented by the type `Clauses`, which is implemented as a list.

Exercise 1: Conversion into Clauses

Make a module `Clause.hs` which exports only the function `programToClauses`. The type of this function is

```
programToClauses :: Program -> Clauses
```

The function accepts as its input a `uProlog` program (parsed and checked by the front-end of the interpreter) and outputs the clausal form of this program. Note that in this conversion queries must be skipped. Only facts and rules must be converted to clausal form.

Note that the order in the list of clauses must be the same as the lines in the `uProlog` program in order to pass tests in Themis. Moreover, within a clause the order of terms should be as in the comment lines in the example `uProlog` program above. You should submit only the module `Clause.hs` to Themis.

Exercise 2: Unification of Predicates

Make a module `Unify.hs` which exports two functions: `mgu`, and `applyUnifier`. Their types are

```
mgu :: FuncApplication -> FuncApplication -> Maybe Unifier
applyUnifier :: Unifier -> FuncApplication -> FuncApplication
```

The function `mgu` accepts two predicates (in the form of a `FuncApplication`). If these predicates cannot be unified, then the function returns `Nothing`. Otherwise, it returns `Just theta` where `theta` is the most general unifier of the input predicates.

For example, if we wat to unify the predicate `male(john)` and `male(robert)` then this function should return `Nothing` since `john` and `robert` cannot be unified. If we want to unify `male(john)` and `male(X)` then the function should return `Just [("X",Const "john")]`.

You should submit only the module `Unify.hs` to Themis.

Exercise 3: Resolution

Make a module `Resolution.hs` which exports only one function:

```
resolveClauses :: Clauses -> Clauses
```

The function gets as its input a list of clauses, and outputs an extension of all the possible clauses that can be inferred using the resolution algorithm (of which the pseudo code is given in the introduction). Note that you have to solve a tricky detail, because predicates may have overlapping variable names. Therefore, you may need to rename some variables (in the literature on resolution and unification, this is called *standardizing apart*) before you can apply the `mgu` function.

You should submit only the module `Resolution.hs` to Themis.

Exercise 4: Answering Queries

Make a module `Query.hs` that exports only one function:

```
answerQueries :: Program -> String
```

The input of this function is the representation of a `uProlog` program, and the output should be a string representation of the answers produced by the `uProlog` interpreter as a complete string (including newlines).

For example, we extend the `uProlog` program that is given in the introduction of this lab with the following queries:

```
?-male(john).
?-male(sue).
?-male(peter).  % returns no because peter is not knwon at all
?-parent(john,X).
?-parent(X,Y).
```

If we feed this program to the function `answerQueries` then the output should be the string

```
"yes\nno\no\X<-[sam,sue]\n(X,Y)<-[(jane,sue),(john,sam),(john,sue)]\n"
```

Note that lists in this output string must be (lexicographically) sorted.

You should submit only the module `Query.hs` to Themis.