Prof. Dr. Jürgen Giesl

Carsten Otto

Notes:

- To solve the programming exercises you can use the Prolog interpreter **SWI-Prolog**, available for free at `http://www.swi-prolog.org`. For Debian and Ubuntu it suffices to install the `swi-prolog` package. You can use the command "`swipl`" to start it and use "`[exercise1].`" to load the facts from file **exercise1.pl** in the current directory.
- Solve these exercises in **groups of three**! For other group sizes **less points** are given!
- The solutions must be handed in **directly before (very latest: at the beginning of)** the exercise course on Wednesday, 08.05.2013, in lecture hall **AH 2**. Alternatively you can drop your solutions into a box which is located right next to Prof. Giesl's office (this box is emptied **a few minutes before** the exercise course starts).
- Please write the **names** and **immatriculation numbers** of all (three) students on your solution. Also please staple the individual sheets!

Exercise 1 (Programming in Prolog): (6 + 2 + 1 + 1 + 4 = 14 points)

Important: In addition to handing in the solution on paper, please also mail your the solutions for this exercise to **lp13-hiwis@i2.informatik.rwth-aachen.de**. Indicate your immatriculation numbers in the subject of the mail and inside the Prolog file.

In this exercise we implement some Prolog predicates that help with substituting variables in terms and formulas. In this setting, terms and formulas are built using the following function symbols:

symbol	meaning
`forall(Var, Sub)`	`Var` is the name of the bound variable, `Sub` is the subformula
`neg(Sub)`	`Sub` is the negated subformula
`and(SubA, SubB)`	`SubA` and `SubB` are the two subformulas
`funcPred(Name, Args)`	`Name` is the name of the function or the predicate symbol, `Args` is a list containing the arguments
`variable(Var)`	`Var` is the name of the variable

In a similar way, we could define the symbols `exists` and `or`, which are handled likewise. To save some unnecessary work, these symbols are not used in this exercise (the resulting program can be extended easily, though).
As an example, the formula $(\forall X\ \mathtt{f}(Z, X)) \wedge \neg \mathtt{g}(Y)$ can be written as the Prolog term

```
and(forall(x, funcPred(f, [variable(z), variable(x)])), neg(funcPred(g, [variable(y)])))
```

You may not use any predefined Prolog predicates in this exercise (unless allowed explicitly)!

Lists

Lists in Prolog are represented by terms built over the signature $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{\mathtt{[]}\}$ and $\Sigma_2 = \{\mathtt{.}\}$. The symbol `[]` denotes the empty list, while the term `.(X,XS)` denotes the list starting with the element `X` (called the head of the list) and having the list `XS` as the remaining list (called the tail of the list). Thus, a list in Prolog containing the three numbers 2, 3 and 5 would be written as `.(2,.(3,.(5,[])))`. This is the standard representation internally used by Prolog.
However, Prolog also knows a more comfortable way to write lists. The term `.(X,XS)` can also be written as `[X|XS]`. With this representation, the above list is written as `[2|[3|[5|[]]]]`. To save brackets, the

Lists (cont)

representation can be shortened by just enumerating elements in the order they appear in the list: `[2,3,5]`. While this list representation is easier to use for humans, it is equivalent to the internal representation used in Prolog. You can use both representations and even mix them within one Prolog program.
As an example for an algorithm working on lists, we write a binary predicate `hasLast` in Prolog where `hasLast(XS, X)` is true iff `X` is the last element of the list `XS`:

```
hasLast([X], X).
hasLast([X|XS], Y) :- hasLast(XS, Y).
```

The following solution is also correct, but uses the less readable list representation.

```
hasLast(.(X, []), X).
hasLast(.(X, XS), Y) :- hasLast(XS, Y).
```

a) To prepare for the following exercise parts, we need predicates that help us deal with lists. Implement each of the following predicates in Prolog:

predicate	**true iff**	**example evaluating to true**
`contained(X,XS)`	`X` is contained in the list `XS`	`contained(a,[b,a])`
`notContained(X,XS)`	`X` is not contained in the list `XS`	`notContained(a,[b,c])`
`app(XS,YS,ZS)`	appending the lists `XS` and `YS` results in the list `ZS`	`app([a,b],[c],[a,b,c])`
`removeDuplicates(XS,YS)`	only the elements of the list `XS` appear in `YS`, each exactly once	`removeDuplicates([a,b,a,c], [a,b,c])`
`union(XS,YS,ZS)`	`ZS` contains the elements of `XS` and `YS`, but each exactly once	`union([a,b,c],[b,a,d], [a,b,c,d])`
`remove(XS,X,ZS)`	`ZS` is the list resulting by removing all occurrences of the element `X` from the list `XS`	`remove([a,b,c,b],b,[a,c])`

Hints:

- Make use of predicates you already defined, for example the predicate `removeDuplicates` is very useful for `union`.
- To test if `a` and `b` are different, you should use the predefined predicate `\=`. Example: `a \= b` is true.

b) Write a predicate `variables(Formula, Variables)` that computes a list of variable names used in terms and formulas built using only the constructs `neg`, `and`, `funcPred`, and `variable`. As an example,

```
variables(and(funcPred(f, [variable(x), funcPred(g, [])]), funcPred(h,
                                [variable(x)])), [x])
```

is true.

Take care that each variable name is contained at most once in the list of the second argument!

Hints:

- It may be useful to define and use a predicate that computes the variables contained in a list of formulas.

c) Extend the predicate `variables` from the previous exercise part so that it is also able to deal with formulas constructed using `forall`. Here, `variables(Formula, Variables)` should only compute the *free* variables in `Formula`.

As an example, `variables(F, [z, y]` is true where `F` is the long example formula shown at the beginning of the whole exercise. The variable `x` is not part of the result because it only occurs at bound positions inside the formula.

d) In the last two parts of this exercise, we consider substitutions. A substitution is a list of pairs, where one component of each pair is a variable and the other component is the term that the variable is replaced with. We represent substitutions using a list where each pair of the substitution is encoded just as two elements of a list. As an example the substitution $[X/f, Y/Z]$ is represented as `[x, funcPred(f,[]), y, variable(z)]`. Note that in this representation, we only mention the names of the replaced variables `x` and `y` (instead of the representation using the function symbol `variable`). The variables are substituted by terms. So to specify that `y` is substituted by `z`, we use the term `variable(z)`.

You may assume that each variable is only replaced once, so `[x, funcPred(f, []), x, funcPred(g, [])]` is not a valid substitution.

Implement a predicate `notInRange(Subst, Variable)` that is true if the variables in the domain of the substitution are only replaced with terms that do not contain `Variable`. As an example, `notInRange([x, funcPred(f, []), y, variable(x)], x)` is false (because `x` occurs in the term `variable(x)`).

e) Finally, implement a predicate `substitute(Formula, Substitution, Result)`. The term resp. formula in `Result` is the result of applying the substitution `Substitution` to `Formula`.

Here, you need to take care that bound variables are not substituted (implementing another predicate might be helpful). Furthermore, the predicate is false if we substitute with a term containing a free variable which will be bound in `Result`. Definition 2.1.8 part (4) gives more details on these two aspects.

Example 1:

```
substitute(and(funcPred(f, [variable(x)]), funcPred(f, [variable(y)])),
           [x, funcPred(h, []), y, variable(x)],
           and(funcPred(f, [funcPred(h, [])]), funcPred(f, [variable(x)])))
```

is true.

Example 2:

```
substitute(forall(x, funcPred(f, [variable(x), variable(y)])),
                  [x, funcPred(h, []), y, variable(x)],
                  forall(x, funcPred(f, [variable(x), variable(x)])))
```

is false (because when replacing `variable(y)` with `variable(x)` we replace with a term that contains the bound variable `x`).

Exercise 2 (Herbrand model): (4 points)

Let

$$\begin{aligned}\varphi =& \mathsf{p}(\mathsf{a},\mathsf{b},\mathsf{a})\\ &\wedge \forall X,Y,Z\ \mathsf{p}(X,Y,Z) \rightarrow \mathsf{p}(\mathsf{f}(X),\mathsf{g}(X,X),X)\\ &\wedge \neg\forall X\ \mathsf{p}(\mathsf{f}(X),\mathsf{g}(X,X),X)\end{aligned}$$

be a formula over the signature (Σ,Δ) with $\Sigma_0 = \{\mathsf{a},\mathsf{b}\}, \Sigma_1 = \{\mathsf{f}\}, \Sigma_2 = \{\mathsf{g}\}$ and $\Delta_3 = \{\mathsf{p}\}$. Here, we use $\mathsf{f}^i(\mathsf{a})$ as an abbreviation for $\underbrace{\mathsf{f}(\ldots(\mathsf{f}}_{i \text{ times}}(\mathsf{a}))\ldots)$.

a) Prove that φ is satisfiable.

b) Give a Herbrand model for φ or show why no such model exists.

c) Transform φ into a corresponding formula ψ in Skolem normal form (that is satisfiability-equivalent to φ).

d) Give a Herbrand model for ψ or show why no such model exists.

e) Are φ and ψ equivalent (cf. Definition 2.2.1)? Explain your answer.

Exercise 3 (Gilmore's algorithm): (5 points)

Consider the following logic program

```
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
plus(0,Z,Z).
```

and the query

```
? - plus(s(0),s(0),s(s(0))).
```

Using Gilmore's algorithm show that the formulas φ_1 and φ_2 corresponding to the logic program entail the formula φ corresponding to the query (i.e., $\{\varphi_1, \varphi_2\} \models \varphi$).