Constructive Logic (15-317), Fall 2012
Assignment 7: Logic Programming in Prolog

Carlo Angiuli (`cangiuli@cs`)

Out: Friday, October 26, 2012
Due: Thursday, November 1, 2012 (before class)

The purpose of this assignment is to familiarize you with logic programming as a computational interpretation for proof search. You will see how you can use the power of built-in backtracking and unification to concisely implement some familiar algorithms.

Your solutions must include Prolog code for `split`, `merge`, `mergesort`, and `infer`, as well as any auxiliary predicates you defined. Your work should be submitted via AFS by copying your code to the directory

```
/afs/andrew/course/15/317/submit/<userid>/hw07
```

where <`userid`> is replaced with your Andrew ID.

1 Running Prolog

To run Prolog, execute

```
/afs/andrew/course/15/317/bin/runprolog
```

from any Andrew machine. Alternatively, you may download and install a copy locally from `http://www.gprolog.org/`, but please test your code a final time on an Andrew machine to ensure it works there, as that is what we will use to grade.

You can load a file `foo.pl` at the Prolog prompt by typing

```
?- [foo].
```

Issue queries by typing predicates at the prompt as you have seen in class; if Prolog offers more solutions, you can see them by typing `;` and ignore them by pressing enter.

2 Mergesort (15 points)

Let `L1@L2` indicate the concatenation of the lists `L1` and `L2`.

Task 1 (3 pts)**.** Implement a predicate `split(L,L1,L2)` which holds exactly when `L1` and `L2` *evenly partition* the list `L`, that is, when `L1@L2` is a permutation of `L`, and `L1` and `L2` differ in length by at most one.

Task 2 (3 pts)**.** Implement a predicate `merge(L1,L2,L)` for sorted lists of integers `L1`, `L2`, and `L`, which holds exactly when `L` is a sorted permutation of `L1@L2`.

Task 3 (9 pts)**.** Implement a predicate `mergesort(L1,L2)` operating over two lists of integers. Your predicate should use the aforementioned primitives to implement mergesort; `mergesort(L1,L2)` should hold exactly when `L2` is a sorted permutation of `L1`.

3 Type inference (25 points)

We can implement symbolic algorithms such as type checking and evaluation in Prolog almost as easily as we can specify them on paper. Consider the proof term assignment for natural deduction that we have been using all semester. In Prolog, we could specify the syntax of terms as follows:

```
term(?x).    % where x is a Prolog atom
term(unit).
term(lam(?x,M))
  :- term(M).
term(app(M,N))
  :- term(M), term(N).
term(pair(M,N))
  :- term(M), term(N).
term(fst(M))
  :- term(M).
term(snd(M))
  :- term(M).
term(inl(M))
  :- term(M).
term(inr(M))
  :- term(M).
term(case(M,?x,N,?y,P))
  :- term(M), term(N), term(P).
```

Here, we use `?x` to represent a variable x, `lam(?x.M)` to represent the term $\lambda x.M$, and `case(M,?x,N,?y,P)` to represent $\mathsf{case}(M, x.N, y.P)$. We represent conjunction by /\, disjunction by \/, implication by =>, and truth by `top`.

```
% Infix notation
:- op(840, xfy, =>).   % implies, right assoc
:- op(830, xfy, \/).   % or, right assoc
:- op(820, xfy, /\).   % and, right assoc
:- op(800,  fy, :).    % has-type, prefix
:- op(800,  fy, ?).    % variable, prefix
```

Figure 1: Prolog starter code for `infer`.

When we looked at proof terms in class, we annotated lambdas by the type of the bound variable; in practice, we can actually *infer* possible types for the variable by looking at the rest of the term. The term $\lambda x.(\pi_1\ x)$ is encoded `lam(?x, fst(?x))`. It must be assigned a type which unifies with `(A /\ B) => A`; that is, it could have type `(A /\ B) => A`, but it would also validly typecheck as `(tt /\ tt) => tt`.

Task 4 (20 pts)**.** Implement a predicate `infer(G, M, A)` that holds whenever term `M` has type `A` under context `G`. (*Hint*: Make sure your implementation is robust with respect to alpha-renaming!)

The file `infer.pl` contains some infix operator declarations relevant to the assignment (see Figure 1).

Task 5 (5 pts)**.** Given a term `M` with no free variables, how may we invoke `infer` to infer the type of `M`? Use this to infer the type of $\lambda x.x\ x$, and explain the behavior you see in terms of what happens in Prolog's proof search. (Put your answers in a comment.)