CS 164 Programming Languages and Compilers — Spring 2019

Discussion Week of 3/18: Prolog and Type Inference

Instructor: Paul N. Hilfinger   GSIs: Nikhil Athreya, Vivant Sakore

The goal of this section is to expose you to logic programming and type inference.

1 Prime Numbers and List Reversal in Prolog

1. **Prime Numbers.** We'll start by looking at prime numbers. Write a function `prime(X)` in Prolog that takes a number `X` and returns true if it is prime and false if not. You may find the starter code file `primes.pl` on the course website useful.
2. **List Reversal.** Write a list reversal predicate in Prolog. We are intentionally not giving you the function signature. Come up with your own.

Answer:

1.
```
add(A, B, C) :- C is A + B.
lt(A, B) :- A < B.
div(A, B, C) :- C is A / B.
divisible(X, Y) :- div(X, Y, Z), integer(Z).

# We would like to write a predicate, composite(X),
# that checks if a number X is composite (that is, whether it
# can be represented as Y * Z for Y > 1, Z > 1).
# To do this, we create a helper predicate, composite(X, Y), that checks if
# X can be divided by Y, or any number greater than Y but smaller than X / 2.

composite(X, Y) :- Y > 1, divisible(X, Y).
composite(X, Y) :- lt(Y, X / 2), composite(X, Y+1).

# Now, our task is much simpler.
# Define composite(X) using the two argument version of composite.
composite(X) :- X > 2, composite(X, 2).
prime(X) :- not(composite(X)).
```

2.
```
lrevaux([], A, A).
lrevaux([X | Y], A, R) :- lrevaux(Y, [X | A], R).
```

```
lrev(X, R) :- lrevaux(X, [], R).
```

Example Trace for List Reversal

```
lrev[[1, 2, 3], R] :- lrevaux[[1,2,3],[], R] :-
lrevaux[[1| [2, 3]], [], R] :- lrevaux[[2, 3], [1], R] :-
lrevaux[[2 | [3]], [1], R] :- lrevaux[[3], [2, 1], R] :-
lrevaux[[3 | []], [2,1], R] :- lrevaux[[], [3,2,1], R] :-
lrevaux[[], [3,2,1], [3,2,1]] ⟹ R = [3,2,1]
```

2 Type System for a Toy Language

Recall the `typeof` predicate from lecture.

```
defn(I, T, [def(I, T) | _]).
defn(I, T, [def(I1, _) | R]) :- dif(I, I1), defn(I, T, R).
typeof(X, T, Env) :- defn(X, T, Env).
```

1. Translate these Prolog lines into English. Be precise.

   Answer: I is defined to have type T if the environment list starts with such a definition, or if I isn't the same as the identifier in the first def but matches the next suitable definition further down the list.

Now we will gradually build up a subset of the following grammar with typing rules. Then we'll experiment with type inference with this grammar.

```
e :  ID | INT
  | [ ((e ,)* e)?  ] /* list */
  | lambda ( ID , e )
  | e '+' e
  | e '<<' e
  | e '//' e
  | cast(e,e)
```

Begin with two typing rules. The starter code is also available online.

```
typeof(X, int, _) :- integer(X).
typeof(X, T, Env) :- defn(X, T, Env).
```

2. Write a typing rule such that the type of an empty list is unbounded (e.g `[_]`).

   Answer: `typeof([], [_], _).`

3. Write a typing rule such that the type of a list is `[T]`, where all list elements are of type `T`.

   Answer: `typeof([E | R], [T], Env) :- typeof(E, T, Env), typeof(R, [T], Env).`

4. Write a typing rule such that the type of a lambda is `T1->T2`, where `T2` is the return type and `T1` is the type `X` is bound to within the body of the lambda.

   Answer: `typeof(lambda(X, E), T1->T2, Env) :- typeof(E, T2, [def(X, T1) | Env]).`

5. Write a typing rule such that the type of `L + R` is `int` when `L` and `R` are both of type `int`.

   Answer: `typeof(L + R, int, Env) :- typeof(L, int, Env), typeof(R, int, Env).`

6. Write a typing rule such that the type of `L // R` is `[T]` when `L` and `R` are both of type `[T]`.

   Answer: `typeof(L // R, [T], Env) :- typeof(L, [T], Env), typeof(R, [T], Env).`

7. Write a typing rule such that the type of `L << R` is `T2` when `L` is of type `T1 -> T2` and `R` is of type `T1`.

   Answer: `typeof(L << R, T2, Env) :- typeof(R, T1, Env), typeof(L, T1->T2, Env).`

8. Write a typing rule such that the type of `cast(L,R)` is `T1 -> T2` when `L` is of type `T1` and R is of type `T2`.

   Answer: `typeof(cast(L, R), T1 -> T2, Env) :- typeof(L, T1, Env), typeof(R, T2, Env).`

   At this point we could enter our rules into Prolog and have it infer types for our programs. For the next few questions, try solving for `T` manually.

   (a) `typeof(f << g, T, [def(f, int->[int]), def(g, int)]).`
   Answer: `T = [int].`

(b) `typeof(lambda(x, x // x) << [1], T, []).`
Answer: `T = [int].`

(c) `typeof(lambda(x, x + x) << [1], T, []).`
Answer: false.

(d) `typeof(lambda(x, x + x) << 1, T, []).`
Answer: int.

(e) `typeof(lambda(x, x + x) + 1, T, []).`
Answer: false.

(f) `typeof(lambda(x, x // x) << [lambda(x, x // x)], T, []).`
Answer: `[[T1]->[T1]].`

Reasoning:

The type of the RHS of the function application is `[[T]->[T]]`. The type of the LHS of the function application is `[T]->[T]`. Consequently, the ultimate type of the expression is `[[T]->[T]]`.

(g) `typeof(lambda(x,lambda(y, cast(x,y))) << [lambda(x,lambda(y, cast(x,y)))], T, []).`
Answer: `(T3->[(T1->(T2->(T1->T2)))]->T3).`

Reasoning:

```
x:  T1
y:  T2
lambda(y,cast(x,y)) :  T2->(T1->T2)
lambda(x, lambda(y,cast(x,y))) :  T1->(T2->(T1->T2))
[lambda(x, lambda(y,cast(x,y)))] :  [T1->(T2->(T1->T2))]
```

Observe that we have already computed the type of the LHS of the function application: it can be written as `T3->(T4->(T3->T4))`. Note the type we are ultimately looking for is the RHS of this type, e.g. `T4->(T3->T4)`. Observe that `T3` is `[T1->(T2->(T1->T2))]`, since this is the type of the expression on the right of the function application.