Constructive Logic (15-317), Spring 2022
Assignment 8: Prolog

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

Submit to Gradescope by Thursday, March 24, 2022, 11:59 pm

For this homework, you will be submitting both written and coding solutions to Gradescope:

- `coloring.pl` (your coding solutions to graph coloring)
- `infer.pl` (your coding solutions to type inference)
- `hw.pdf` (your written solutions)

1 Mode Checking

For each of the following problems, you will be given a predicate and desired modes for that predicate. Determine if each mode is well-moded for the given predicate. If so, provide the modes for the other predicates that are necessary for our predicate to be well-moded and prove that it is well-moded for the given modes. If not, explain why the desired mode does not work for the given predicate.

For example, consider the following Prolog program for `mult`. In it, `plus`(M, N, P) holds when $M + N = P$.

```
mult(z, N, z).
mult(s(M), N, Q) :-
  mult(M, N, P),
  plus(P, N, Q).
```

In the case that the desired mode is `mult`$(+, +, -)$, we can prove that this is well-moded by induction on the structure of the rules that if the values of the first two arguments of `mult` are known then the third argument will be known in case the search succeeds, as long as `plus`$(+, +, -)$ is well-moded.

Clause 1: For the clause `mult`(z, N, z), we know z and N from the first and second argument. The third argument will be z, meaning the clause holds.

Clause 2: For the clause `mult`$(s(M), N, Q)$, we know s(M) and N from the first and second arguments in the clause. This means that we also know M and can apply the induction hypothesis to conclude that we know Q. Since `plus`$(+, +, -)$ is well-moded, that means we will know Q, meaning the clause holds.

In the case that the desired mode is `mult`$(-, -, +)$, we know that it cannot be well-moded because in the second clause, we would need to show that `mult`$(-, -, -)$ is also well-moded, which we cannot do.

Task 1 (9 points)**.** Consider the following Prolog program for `subsetsum`. In it, `subset`(L, S) holds when S is a subset of L and `sum`(L, Acc) holds when all the elements in L sum to Acc.

```
subsetsum(List, Sum, Subset) :-
  subset(List, Subset),
  sum(Subset, Sum).
```

a. `subsetsum`$(+, -, +)$

b. `subsetsum`$(-, +, -)$

c. `subsetsum`$(+, -, -)$

Task 2 (9 points)**.** Consider the following Prolog program for `mergesort`. In it, `split`$(L, L1, L2)$ holds when $L1$ concatenated with $L2$ is a permutation of L and the size of $L1$ and $L2$ differ by at most 1 and `merge`$(L, L1, L2)$ holds when L is the sorted permutation of $L1$ concatenated with $L2$.

```
mergesort([], []).

mergesort([A | []], [A | []]).

mergesort(Unsorted, Sorted) :-
  split(Unsorted, UHalf1, UHalf2),
  mergesort(UHalf1, SHalf1),
  mergesort(UHalf2, SHalf2),
  merge(SHalf1, SHalf2, Sorted).
```

a. `mergesort`$(+, -)$

b. `mergesort`$(-, +)$

c. `mergesort`$(+, +)$

2 Coloring maps

Graph coloring is an interesting problem in graph theory. A graph coloring is an assignment of colors to each vertex such that no two adjacent vertices have the same color. Of particular interest is a coloring using a minimum number of colors; this number is called the *chromatic number* of the graph. The four-color theorem states that any planar graph[1] can be colored using at most four colors. The theorem was proved in 1976 using a computer program, and has caused much controversy (is a computer proof really a proof?). It has since been formally verified using the Coq theorem prover in 2005.

As a consequence of this theorem, any map can be colored with at most four colors such that no adjacent regions have the same color. This is because every map can be represented by a planar graph, with one vertex for each region, and an edge between two vertices if and only if their corresponding regions are adjacent.

Consider, for example, Australia's map in Figure 1. Observe that this map uses more colors than necessary, although this might make it more visually appealing.

[1] A graph that can be drawn on the plane with no crossing edges.

Figure 1: Australia (more colorful than necessary)

Task 3 (15 points)**.** Implement a predicate `color_graph`$(nodes, edges, colors)$ that associates with the graph $(nodes, edges)$ all of the valid 4-colorings of the graph. Submit your implementation in a file named `coloring.pl`.

The predicate `color_graph` should find all valid colorings via backtracking. For efficiency reasons, you may prefer to find all valid colorings without repetition, but we will not be checking this. Once all valid solutions have been found via backtracking, the predicate should fail. You may assume the graph is finite, and your implementation should satisfy the following requirements:

1. You should define a `color/1` predicate with four colors.
2. Assume there are predicates `node/1` and `edge/2` that each take in atoms.
3. In `color_graph/3`, the first parameter is a list of `node/1` terms, the second parameter is a list of `edge/2` terms, and the third parameter is a list of pairs `(a,c)`, which indicates that `node(a)` is colored with `color(c)`.
4. The predicate `color_graph` should have mode `color_graph`$(+nodes, +edges, -coloring)$.
5. Given ground inputs for the first two arguments (*i.e.*, nodes and edges), `color_graph` should always return at least one coloring. (You will not be given an graph that cannot be 4-colored.) If Prolog is asked to backtrack and generate additional solutions, `color_graph` should return all possible colorings.

Your solution does not need to be very long. The reference solution is just 19 lines of Prolog, including the color definitions.

For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by executing:

```
$ ./test_coloring.sh
```

The test script uses another Prolog file (`coloring_tests.pl`) to test your implementation. It employs some Prolog features that you have not been taught. We strongly urge you not to use any of those features in your solution. They are unlikely to help, and are very likely to make your code hard to understand.

3 Type Inference

In Homework 6, you implemented type-checking in SML to check if a given STLC proof term was correct for a given proposition. This time, we will implement type-inference in Prolog, meaning we will be producing a valid proposition for a given proof term.

Task 4 (22 points)**.** Implement type inference for STLC in Prolog. You must define the predicate `infer`$(term, prop)$ that produces a valid proposition for a given proof term, and use the predefined logical operators in the starter code. Proof terms will be in the form of STLC. Submit your implementation in a file named `infer.pl`. Your implementation should satisfy the following requirements:

1. Assume that the predicates and atoms for types are defined and follows:
   - `atomTy/1`, a predicate which takes an atom as a parameter
   - `unitTy`, an atom that represents the unit type
   - `voidTy`, an atom that represents the void type
   - `times/2`, a predicate with both parameters being types
   - `plus/2`, a predicate with both parameters being types
   - `arrows/2`, a predicate with both parameters being types
2. Assume that the predicates and atoms for proof terms are defined as follows:
   - `var/1`, a predicate which takes an atom as a parameter
   - `unit/0`, an atom that represents unit
   - `tuple/2`, a predicate with both parameters being proof terms
   - `fst/1`, a predicate that takes a proof term as a parameter
   - `snd/1`, a predicate that takes a proof term as a parameter
   - `inl/3`, a predicate that takes a proof term as its first parameter and a type as its second and third
   - `inr/3`, a predicate that takes a proof term as its first parameter and a type as its second and third
   - `case/5`, a predicate that takes a proof term as its first, third, and fifth parameter and an atom as the second and fourth parameter
   - `fn/3`, a predicate that takes an atom as its first parameter, a type as its second, and a proof term as its third
   - `app/2`, a predicate with both parameters being proof terms
   - `abort/2`, a predicate with its first parameter being a proof term and its second parameter being a type
3. In `infer/2`, the first parameter is a proof term and the second parameter is a proposition.
4. The predicate `infer` should have mode `infer`$(+term, -prop)$.
5. Given ground input for the first argument (*i.e.*, proof term), `infer` should always return at the most general proposition for the argument.
6. You will likely need to make use of `unify_with_occurs_check/2` to ensure that ill-typed terms are not given types.

For your convenience, we have provided you with a shell script to test your implementation. You can invoke it by executing:

```
$ ./test_infer.sh
```

The test script uses another Prolog file (`inference_tests.pl`) to test your implementation. It employs some Prolog features that you have not been taught. We strongly urge you not to use any of those features in your solution. They are unlikely to help, and are very likely to make your code hard to understand.