Homework #4

CS231

Due by the end of the day on February 20. You should submit one file: `hw4.pdf`.

Remember that you are encouraged to work in pairs on homework assignments. See the course syllabus for details. Also remember the course's academic integrity policy. In particular, you must credit other people and other resources that you consulted. Again, see the syllabus for details.

1. Consider the simply-typed lambda calculus augmented with pairs and tagged unions from Section 3 of the cheat sheet. Assume we also include three *uninterpreted* types in the syntax of types, denoted `A`, `B`, and `C`. Demonstrate that the following theorems are valid in constructive propositional logic by providing a term that has the associated type, under an empty type environment (recall that ¬`T` is shorthand for `T → False`):

   (a) Associativity of conjunction:
   `((A ∧ B) ∧ C) → (A ∧ (B ∧ C))`

   (b) Associativity of disjunction:
   `((A ∨ B) ∨ C) → (A ∨ (B ∨ C))`

   (c) Distributivity of implication over disjunction:
   `((A ∨ B) → C) → ((A → C) ∧ (B → C))`

   (d) Transposition:
   `(A → B) → (¬B → ¬A)`

2. In this problem you will augment the simply typed lambda calculus integers, booleans, the unit value, and `let` (see the cheat sheet) with a new term `while` $\texttt{t}_1$ `do` $\texttt{t}_2$ for loops. The semantics of this construct is the normal one for such loops: while $\texttt{t}_1$ evaluates to `true` we repeatedly execute $\texttt{t}_2$. We assume that $\texttt{t}_2$ is just executed for its side effects, so it (and the entire loop) should have type `True`.

   Define the small-step operational semantics for this new term. *Hint: It will be easiest to define the operational semantics using a single rule.* Also define a typing rule for the new term.

3. Suppose the language from the previous question is augmented with the `letrec` expression from the cheat sheet. Show that now `while` need not be a primitive in the language by implementing a function `whileFun` such that any term `while` $\texttt{t}_1$ $\texttt{t}_2$ can instead be expressed as

   `whileFun (function x:Unit -> `$\texttt{t}_1$`) (function x:Unit -> `$\texttt{t}_2$`)`

   where `x` is not free in $\texttt{t}_1$ or $\texttt{t}_2$. The reason that `whileFun` takes two functions as arguments, instead of directly taking $\texttt{t}_1$ and $\texttt{t}_2$, is to avoid evaluating $\texttt{t}_1$ and $\texttt{t}_2$ to values before passing them to `whileFun`. When functions are used for this purpose they are called *thunks*. So `whileFun` should have the type `(True -> Bool) -> (True -> True) -> True`.

4. Consider adding a type and associated operations for binary trees to our full language from the cheat sheet:

   `t ::= ... | leaf | node(`$\texttt{t}_1$`,`$\texttt{t}_2$`,`$\texttt{t}_3$`) | leftTree t | data t | rightTree t`
   `tv ::= leaf | node(`$\texttt{tv}_1$`, `$\texttt{v}_2$`, `$\texttt{tv}_3$`)`

```
v ::= ... | tv
T ::= ... | T Tree
```

The new term `leaf` represents an empty tree, and `node(t1, t2, t3)` is used to create a non-empty tree with left subtree t_1, data t_2, and right subtree t_3. The three accessor functions `leftTree`, `data`, and `rightTree` access the associated components of a `node`.

The new metavariable `tv` represents tree values, which are themselves a kind of value `v`. For example, a tree with the value 1 at the root, 2 in the left child of the root, and 3 in the right child of the root would be represented as the tree value `node(node(leaf,2,leaf), 1, node(leaf,3,leaf))`. Finally, we have a new type of the form `T Tree`; for example, the tree value above has type `Int Tree`.

(a) Provide the small-step operational semantics of the form $\texttt{t} \longrightarrow \texttt{t}'$ for the new terms.

(b) Provide the typing rules of the form $\Gamma \vdash \texttt{t} : \texttt{T}$ for the new terms.

(c) Are your typing rules *algorithmic*, or would the implementation of a typechecker sometimes have to "guess" types? If the latter, what modifications could you make to the language to make typechecking algorithmic?

(d) Unfortunately your type system is not sound! This is not your fault — the types for trees that I gave you are simply not expressive enough. Demonstrate the unsoundness by providing a counterexample to either the Progress or Preservation theorem (and specify which one).

5. In class we saw the following *type inference* rules for the simply-typed lambda calculus with integer constants and addition (here metavariable `s` represents the same language as `t` but without type annotations on formal parameters, and `S` is the same type language as `T` with the addition of type variables ranged over by metavariable `X`):

$$\frac{\texttt{X fresh}}{\text{tinfer}(\Gamma, \texttt{n}) = (\texttt{X}, \{\texttt{X=Int}\})}$$

$$\frac{\texttt{X fresh} \qquad \text{tinfer}(\Gamma, \texttt{s}_1) = (\texttt{S}_1, C_1) \qquad \text{tinfer}(\Gamma, \texttt{s}_2) = (\texttt{S}_2, C_2)}{\text{tinfer}(\Gamma, \texttt{s}_1 \texttt{ + } \texttt{s}_2) = (\texttt{X}, \{\texttt{X=Int}, \texttt{ S}_1\texttt{=Int}, \texttt{ S}_2\texttt{=Int}\} \cup C_1 \cup C_2)}$$

$$\frac{\texttt{X fresh} \qquad \Gamma(\texttt{x}) = \texttt{S}}{\text{tinfer}(\Gamma, \texttt{x}) = (\texttt{X}, \{\texttt{X=S}\})}$$

$$\frac{\texttt{X fresh} \qquad \texttt{X}_1 \text{ fresh} \qquad \text{tinfer}(\Gamma, \texttt{x:X}_1, \texttt{s}) = (\texttt{S}_2, C)}{\text{tinfer}(\Gamma, \texttt{function x -> s}) = (\texttt{X}, \{\texttt{X=X}_1\texttt{→S}_2\} \cup C)}$$

$$\frac{\texttt{X fresh} \qquad \text{tinfer}(\Gamma, \texttt{s}_1) = (\texttt{S}_1, C_1) \qquad \text{tinfer}(\Gamma, \texttt{s}_2) = (\texttt{S}_2, C_2)}{\text{tinfer}(\Gamma, \texttt{s}_1 \texttt{ s}_2) = (\texttt{X}, \{\texttt{S}_1\texttt{=S}_2\texttt{→X}\} \cup C_1 \cup C_2)}$$

(a) Provide a derivation tree through these rules for the following program, starting from an empty type environment:

```
function f -> function x -> f (f x)
```

(b) Provide a *substitution* (a mapping from type variables to types) that represents a solution to the constraints that you generated in the derivation above, and then explicitly provide the corresponding types for `f`, `x`, and the entire program. (You need not provide the most general solution to the constraints, but see if you can.)

6. Recall the typing rules for `left t`, `right t`, and `match`, which are the terms related to tagged unions — see the cheat sheet.

   Write the corresponding *type inference* rules, of the form tinfer(Γ, s) = (S, C), for these three terms, where C is a set of type equality constraints of the form $\mathtt{S}_1 = \mathtt{S}_2$.