Homework #3

CS231

Due by the end of the day on October 26. You should submit one PDF file.

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. 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 `Unit`.

   Define the small-step operational semantics for this new term. Your semantics may use the sequencing derived form $\texttt{t}_1\texttt{;}\,\texttt{t}_2$ if you desire. Also define a typing rule for the new term.

$$\frac{}{\texttt{while t}_1\texttt{ do t}_2 \longrightarrow \texttt{if t}_1\texttt{ then (t}_2\texttt{ ; while t}_1\texttt{ do t}_2\texttt{) else ()}}$$

$$\frac{\Gamma \vdash \texttt{t}_1\texttt{:Bool} \qquad \Gamma \vdash \texttt{t}_2\texttt{:Unit}}{\Gamma \vdash \texttt{while t}_1\texttt{ do t}_2\texttt{:Unit}}$$

2. Suppose the language from the previous question is augmented with the `letrec` expression. Show that now `while` need not be a primitive in the language by defining it as a syntactic sugar: define its operational semantics with a single rule with no premises, where the stepped-to expression does not employ the `while` syntax. You can assume that $\texttt{t}_1$ and $\texttt{t}_2$ do not contain any uses of `while`. Your new term need not evaluate in perfect step-by-step synchrony with the original term but should have the same behavior.

   `letrec while = function u:Unit -> if` $\texttt{t}_1$ `then (`$\texttt{t}_2$`;while()) else () in while()`

3. Functional languages rely heavily on *pattern matching*, which is a declarative way to test the structure of a value, give names to its pieces, and perform different actions based on that structure. For example, as we've seen, OCaml includes an expression of the form `match t with` $\texttt{p}_1$ `->` $\texttt{t}_1$ `|` $\cdots$ `|` $\texttt{p}_n$ `->` $\texttt{t}_n$. The semantics is as follows: The term `t` is evaluated to a value `v`. The result of the whole expression is then the value of $\texttt{t}_i$, where i is the smallest index such that `v` *matches* the pattern $\texttt{p}_i$. Any variables declared in $\texttt{p}_i$ are in scope within $\texttt{t}_i$ and so may be referred to there.

   In this problem you will formalize a variant of OCaml's `match` expression. For simplicity we'll consider only a subset of the possible patterns and we'll assume that the `match` expression always has exactly two cases. Consider the simply-typed lambda calculus augmented with numbers and pairs. We'll augment this language with the `match` expression (not to be confused with the `match` expression we have defined specifically for tagged unions, which is a special case of this one), along with the following syntax for pattern matching:

$\texttt{t} ::= \ldots \mid \texttt{(match t with } \texttt{p}_1 \texttt{ => } \texttt{t}_1 \texttt{ | } \texttt{p}_2 \texttt{ => } \texttt{t}_2\texttt{)}$
$\texttt{p} ::= \texttt{\_:T} \mid \texttt{x:T} \mid \texttt{n} \mid \texttt{(}\texttt{p}_1\texttt{,}\texttt{p}_2\texttt{)}$

Here `_:T` is the *wildcard* pattern, which any value matches. The associated type should be ignored at run time but used for static typechecking. The `x:T` pattern is also matched by any value but additionally declares the variable `x` and binds it to that value, again ignoring the declared type. A numeric value pattern `n` is matched only by that same numeric value. A pair pattern $\texttt{(}\texttt{p}_1\texttt{,}\texttt{p}_2\texttt{)}$ is matched by any pair value $\texttt{(}\texttt{v}_1\texttt{,}\texttt{v}_2\texttt{)}$ whose two component values respectively match the corresponding component patterns. I encourage you to play more with OCaml's `match` expression to get some more intuition on this great language feature.

(a) Provide rules for a new judgment of the form $\vdash \texttt{v}$ *matches* $\texttt{p} \Rightarrow \texttt{E}$, which formalizes the run-time semantics of pattern matching. The judgment should be provable when value `v` matches pattern `p`. The result of a successful match is an *environment* `E`, which is a finite function (or equivalently a set of pairs) from variables to values. For example, the result of matching the value `((1,2),(3,4))` against the pattern $\texttt{((}\texttt{x}_1\texttt{:Int,}\texttt{y}_1\texttt{:Int),(}\texttt{x}_2\texttt{:Int,}\texttt{y}_2\texttt{:Int))}$ is the environment $\{(\texttt{x}_1, 1),(\texttt{y}_1, 2),(\texttt{x}_2, 3),(\texttt{y}_2, 4)\}$. This environment provides the values of the variables declared in a pattern, which may be referred to in the corresponding body term within a `match` expression. You may assume that a variable name is never used more than once within a given pattern.

$$\frac{}{\vdash \texttt{v} \textit{ matches } \texttt{(\_:T)} \Rightarrow \emptyset}$$

$$\frac{}{\vdash \texttt{v} \textit{ matches } \texttt{(x:T)} \Rightarrow \{\texttt{(x,v)}\}}$$

$$\frac{}{\vdash \texttt{n} \textit{ matches } \texttt{n} \Rightarrow \emptyset}$$

$$\frac{\vdash \texttt{v}_1 \textit{ matches } \texttt{p}_1 \Rightarrow \texttt{E}_1 \qquad \vdash \texttt{v}_2 \textit{ matches } \texttt{p}_2 \Rightarrow \texttt{E}_2}{\vdash \texttt{(}\texttt{v}_1\texttt{,}\texttt{v}_2\texttt{)} \textit{ matches } \texttt{(}\texttt{p}_1\texttt{,}\texttt{p}_2\texttt{)} \Rightarrow \texttt{E}_1 \cup \texttt{E}_2}$$

(b) Now add rules to the judgment $\texttt{t} \longrightarrow \texttt{t'}$ to define the run-time behavior of the `match` expression. These rules can refer to the judgment defined above. They can also use the notation $\nvdash \texttt{v}$ *matches* `p` to denote the fact that there is no `E` such that $\vdash \texttt{v}$ *matches* $\texttt{p} \Rightarrow \texttt{E}$ can be derived. The rules should get stuck if no pattern matches.

$$\frac{\texttt{t} \longrightarrow \texttt{t'}}{\texttt{match t with } \texttt{p}_1 \texttt{ => } \texttt{t}_1 \texttt{ | } \texttt{p}_2 \texttt{ => } \texttt{t}_2 \longrightarrow \texttt{match t' with } \texttt{p}_1 \texttt{ => } \texttt{t}_1 \texttt{ | } \texttt{p}_2 \texttt{ => } \texttt{t}_2}$$

$$\frac{\vdash \texttt{v} \textit{ matches } \texttt{p}_1 \Rightarrow \{\texttt{(}\texttt{x}_1\texttt{,}\texttt{v}_1\texttt{)},\cdots,\texttt{(}\texttt{x}_m\texttt{,}\texttt{v}_m\texttt{)}\}}{\texttt{match v with } \texttt{p}_1 \texttt{ => } \texttt{t}_1 \texttt{ | } \texttt{p}_2 \texttt{ => } \texttt{t}_2 \longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_1]\cdots[\texttt{x}_m \mapsto \texttt{v}_m]\texttt{t}_1}$$

$$\frac{\nvdash \texttt{v} \textit{ matches } \texttt{p}_1 \qquad \vdash \texttt{v} \textit{ matches } \texttt{p}_2 \Rightarrow \{\texttt{(}\texttt{x}_1\texttt{,}\texttt{v}_1\texttt{)},\cdots,\texttt{(}\texttt{x}_m\texttt{,}\texttt{v}_m\texttt{)}\}}{\texttt{match v with } \texttt{p}_1 \texttt{ => } \texttt{t}_1 \texttt{ | } \texttt{p}_2 \texttt{ => } \texttt{t}_2 \longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_1]\cdots[\texttt{x}_m \mapsto \texttt{v}_m]\texttt{t}_2}$$

(c) Define a judgment $\vdash \texttt{p:T} \Rightarrow \Gamma$ which determines the type of a pattern (thereby determining the allowed type of terms that can be matched against the pattern in a `match` expression). The judgment also produces a type environment Γ which records the types of the variables declared in the pattern. For example, the pattern $\texttt{((}\texttt{x}_1\texttt{:Int,}\texttt{y}_1\texttt{:Int),(}\texttt{x}_2\texttt{:Int,}\texttt{y}_2\texttt{:Int))}$ has type `(Int∧Int)∧(Int∧Int)` and produces

the type environment $\{(\texttt{x}_1, \texttt{Int}), (\texttt{y}_1, \texttt{Int}), (\texttt{x}_2, \texttt{Int}), (\texttt{y}_2, \texttt{Int})\}$. Again, you may assume that a variable name is never used more than once within a given pattern.

$$\frac{}{\vdash \texttt{(\_:T)} : \texttt{T} \Rightarrow \emptyset}$$

$$\frac{}{\vdash \texttt{(x:T)} : \texttt{T} \Rightarrow \{\texttt{(x,T)}\}}$$

$$\frac{}{\vdash \texttt{n:Int} \Rightarrow \emptyset}$$

$$\frac{\vdash \texttt{p}_1 : \texttt{T}_1 \Rightarrow \Gamma_1 \qquad \vdash \texttt{p}_2 : \texttt{T}_2 \Rightarrow \Gamma_2}{\vdash \texttt{(p}_1\texttt{,p}_2\texttt{)} : \texttt{T}_1 \wedge \texttt{T}_2 \Rightarrow \Gamma_1 \cup \Gamma_2}$$

(d) Now add rules to the judgment $\Gamma \vdash \texttt{t:T}$ to define static typechecking for the `match` expression. These rules can refer to the judgment defined above. Think carefully about the requirements to ensure that neither the body of one of the `match` cases nor the context that uses the result value from a `match` will get stuck. You do not need to statically prevent a failure to match any patterns, however (which OCaml's type system does prevent, by warning when the patterns in a a `match` are not *exhaustive* for the argument type). Trying examples out in OCaml should be helpful.

$$\frac{\Gamma \vdash \texttt{t:T} \qquad \vdash \texttt{p}_1 : \texttt{T} \Rightarrow \Gamma_1 \qquad \vdash \texttt{p}_2 : \texttt{T} \Rightarrow \Gamma_2 \qquad \Gamma \cup \Gamma_1 \vdash \texttt{t}_1 : \texttt{T}' \qquad \Gamma \cup \Gamma_2 \vdash \texttt{t}_2 : \texttt{T}'}{\Gamma \vdash \texttt{(match t with p}_1 \texttt{ => t}_1 \texttt{ | p}_2 \texttt{ => t}_2\texttt{)} : \texttt{T}'}$$

4. Consider the simply-typed lambda calculus augmented with pairs and tagged unions. 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):

   (a) Transitivity of implication:
   ```
   ((A → B) ∧ (B → C)) → (A → C)
   function p:A → B ∧ B → C -> function x:A -> (snd p) ((fst p) x)
   ```

   (b) Commutativity of disjunction:
   ```
   (A ∨ B) → (B ∨ A)
   function u:A ∨ B -> match u with left x -> right x | right y -> left y
   ```

   (c) A form of distributivity of implication over disjunction:
   ```
   ((A ∨ B) → C) → ((A → C) ∧ (B → C))
   function f:(A ∨ B) → C ->
   (function x:A -> f (left x), function x:B -> f (right x))
   ```