CSE 505, Fall 2003, Assignment 3
Due: 13 November 2003, 10:30AM (firm)

Advice: Do some of this assignment before the midterm. In particular, problems 1 and 3 don't require much writing and problem 2 is excellent review.

1. (Evaluation Order)

   (a) (Reverse Engineering) Write an O'Caml function `left_first_app` of type `unit->bool`. The function body should contain (among other things) an expression of the form $e_1\ e_2$. `left_first_app` should return true if O'Caml evaluates e_1 before e_2 (and false if O'Caml evaluates e_2 before e_1). Hint: Use exceptions (or mutable references if you prefer).

   (b) (Thunking to Control Evaluation Order) Write an O'Caml function `which_first` of type `(unit->'a->'b)->(unit->'a)->bool->'b` such that `which_first x1 x2 b` has these properties:

   - If `x1 ()` evaluates to `v1` and `x2 ()` evaluates to `v2`, then it returns the evaluation of `v1 v2`.
   - If `b` is true, then it evaluates `x1()` before `x2()`.
   - If `b` is false, then it evaluates `x2()` before `x1()`.

2. (Closedness as Type Safety) For this problem, let our λ-calculus syntax be $e ::= c \mid x \mid \lambda x.\ e \mid e\ e$ and let our operational semantics be CBV left-to-right, small-step.

   (a) Define a type system with only one type (called ok) with this property: We should be able to derive $\Gamma \vdash e : \mathsf{ok}$ if and only if e contains no constants and all free variables in e are in Γ. (In particular, $\cdot \vdash e : \mathsf{ok}$ implies e has no free variables.)

   (b) Give an expression such that $\cdot \vdash e : \mathsf{ok}$ but e is not well-typed in the simply-typed λ-calculus.

   (c) Prove Type Safety for your type system with respect to the operational semantics.

   (d) If we add the rule $\dfrac{}{\Gamma \vdash c : \mathsf{ok}}$, your type system should no longer be safe. Give an inductive definition of the stuck states a term that type-checks with this new rule could get to. Do *not* treat values (which include constants) as stuck for the purpose of this definition.

3. (Sum-Type Projection) In lecture, we added sum-types and a case expression to the simply-typed λ-calculus. Consider a language that does not have case expressions, but does have conditionals (as in lecture), "sum projections", and "tag tests":

$$e ::= \ldots \mid e.\mathsf{left} \mid e.\mathsf{right} \mid \mathsf{isleft}(e)$$

$$\frac{e \to e'}{e.\mathsf{left} \to e'.\mathsf{left}} \qquad \frac{e \to e'}{e.\mathsf{right} \to e'.\mathsf{right}} \qquad \frac{e \to e'}{\mathsf{isleft}(e) \to \mathsf{isleft}(e')}$$

$$\frac{}{(\mathsf{inl}(v)).\mathsf{left} \to v} \qquad \frac{}{(\mathsf{inr}(v)).\mathsf{right} \to v} \qquad \frac{}{\mathsf{isleft}(\mathsf{inl}(v)) \to \mathsf{true}} \qquad \frac{}{\mathsf{isleft}(\mathsf{inr}(v)) \to \mathsf{false}}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash e.\mathsf{left} : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash e.\mathsf{right} : \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2}{\Gamma \vdash \mathsf{isleft}(e) : \mathsf{bool}}$$

   (a) Prove that this language is not type-safe. (That is, give an expression, show that it type-checks, and show that it gets stuck without reaching a value.)

   (b) Show how we can use conditionals, sum projections, and tag tests to make case expressions a "derived form" (i.e., a macro).

4. (Implementation) The code provided to you defines abstract syntax and a parser for the simply-typed λ-calculus with booleans, constants, addition, the $>$ comparison, and rec (instead of fix). To make parsing and type-checking easier, functions have the concrete form `(fn x:t. e)`. Specfically, they must be surrounded by parentheses, they must have explicit argument types, and the ":" and "." must be present.

   Similarly, recursive functions have the form `(rec f:t x. e)`. This creates a recursive function of type `t`. In `e`, `f` is bound to the function and `x` is bound to the argument.

   Conditionals require parentheses.

   (a) Give a formal typing rule for rec $f\ x.e$ in the simply-typed λ-calculus. (Hint: Extend the context with f and x to type-check e.)

   (b) Define an exponentiation function in this language. That is, give a term of type `int->int->int` such that applying it to `n` and `m` returns n^m.

   (c) Implement the type-checker for this language by changing `Main.typecheck`. Your type-checker should raise an exception if the expression does not type-check (or if it encounters the Closure variant, which is explained below).

   (d) Implement a *large-step environment-based* interpreter by changing `Main.interpret`. Such an interpreter is explained below. Your interpreter should raise an exception if it gets stuck. (But expressions that type-check should not get stuck). You will see that the code for implementing environments is provided to you.

   An environment-based interpreter does not use substitution. Instead, the program state includes an environment, which maps variables to values. To implement lexical scope correctly, functions are not values—they evaluate to *closures* (written $<e, E>$ below). Here is a formal large-step semantics using this approach (omitting booleans, conditionals, $>$, and rec). Pay particular attention to the rule for function application—we evaluate function bodies using the environment in its closure!

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda x.\ e \mid e\ e \mid \textsf{rec}\ f\ x.e \mid <\lambda x.\ e, E> \\ E & ::= & \cdot \mid E, x \mapsto v \\ v & ::= & c \mid <\lambda x.\ e, E> \end{array}$$

$$\frac{}{E; v \Downarrow v} \qquad \frac{}{E; x \Downarrow E(x)} \qquad \frac{}{E; \lambda x.\ e \Downarrow <\lambda x.\ e, E>}$$

$$\frac{E; e_1 \Downarrow <\lambda x.\ e_3, E_1> \qquad E; e_2 \Downarrow v_2 \qquad E_1, x \mapsto v_2; e_3 \Downarrow v}{E; e_1\ e_2 \Downarrow v}$$

   As usual with large-step semantics, evaluation order is not fully specified. (We're CBV, but your implementation can be left-to-right or right-to-left.)

   Formalizing rec without substitution is interesting. What we want is $E;$ rec $f\ x.e \Downarrow <\lambda x.\ e, E'>$ where $E' = E, f \mapsto <\lambda x.\ e, E'>$, which has a circularity. In math, we would take a fix-point over an environment-generator. In O'Caml, we need to either represent our environment with a recursive function or with a circular data structure (which requires mutation). The code given to you shows both approaches. (Note the first approach just pushes the problem off to O'Caml's implementors, making them be the ones who build a circular data structure.)

   You can probably do the homework without understanding most of the previous paragraph.

What to turn in:

- A file called `evalorder.ml` for Problem 1.
- Written/printed solution for Problems 2, 3, and 4a.
- A file called `exponent` for problem 4b.
- A modified `main.ml` for Problem 4c and d.

Note: I encourage you to post interesting test programs (that don't implement exponentiation) to cse505.