Graduate Programming Languages, Extra Homework Problems (These All Fit Between Assignments 3 and 5)

Associated Caml files are in `hw_extra_code.tar`.

1. (The CPS Transformation) Extend the CPS transformation from lecture 13 to include the translation for pairs and sums as introduced in lecture 11.

2. (Continuation-Passing Style) In this problem you will reimplement the large-step, environment-based interpreter and the type-checker from homework 3. Your reimplementations should always use a constant amount of stack space regardless of how big a program they evaluate or type-check. To do so, use the idiom of continuation passing. Note that you are manually using continuation-passing style to implement the interpreter and type-checker; you are *not* applying a CPS transformation to the program being type-checked and evaluated.

   - In the provided code, complete the definition of `problem2/interpret`, which should have type `exp -> (exp * heap) option` where the result `Some (v,h)` carries the final value and heap and the result `None` indicates a run-time error occurred. Two cases of the tail-recursive helper function are provided to you. This helper function should never raise an exception: it should return `None` or invoke the continuation it is passed. Hints:
     - There is no reason to use the `Some` constructor in this helper function.
     - It is probably easiest to copy parts of your solution to homework 3 and then modify them.
   - In the provided code, complete the definition of `typecheck`, which should have type `exp -> typ option` where the result `Some typ` carries the type of the entire program and `None` indicates a type-error was found. You need to define a helper function that, like the helper function in part (a), takes a function as an extra argument that serves as a continuation.

3. (Machines and Continuations) You are given an untyped lambda-calculus and part of a low-level abstract machine. The machine uses explicit evaluation contexts and environments, much like the last and most efficient interpreter in lecture 14 (`interp_closure`). The definition of syntax and contexts (`problem3/ast.ml`) is somewhat different to support more easily the fact that, unlike in lecture, we have several types of values.

   (a) Complete the definition of `Main.interp` to support pairs, conditionals, and first-class continuations. You must maintain tail-recursion. Note the kinds of contexts you need are already defined for you in `ast.ml`, along with comments about their purpose. You can do continuations last; the provided testing program (`adder`) doesn't use them.

   (b) Change `Main.allow_halt` such that:

   - It takes an expression e and returns an expression e'.
   - e can have free occurrences of the variable `halt` and call it as a *function* taking one argument, i.e., `halt` e''.
   - If e evaluates to v without ever calling `halt`, then e' evaluates to (true, v).
   - If e evaluates after some number of steps to $E[\texttt{halt}\ v]$, then e' evaluates to (false, v).
   - e' contains e as a subexpression – that is, do not examine e, just wrap it with some outer code.

   Sample solution is 3 lines. Advice: Work out your solution on paper first. Put e in a function that takes `halt` as an argument. Pass this function a function that contains a `throw`.

4. (Sums and Subtyping) Consider a typed λ-calculus with a more flexible version of sum types than considered in lecture:

   - There are an infinite number of constructors, not just A and B. Let C range over constructors. So an example expression is $C_7\ (\lambda x.\ x)$.
   - A single sum type $+\{C_1{:}\tau_1, \ldots, C_n{:}\tau_n\}$ can list any finite number of constructors and the types of the values they carry. So one example type would be $+\{C_3{:}\mathsf{int}, C_7{:}\mathsf{int} \to \mathsf{int}, C_2{:}\mathsf{int}\}$. Like in Caml, the order of constructors is not significant. Unlike in Caml, we are using structural typing and different types can use the same constructors (with possibly different types they carry).
   - As you should expect, a match expression can have any finite number of branches, with a different constructor for each branch. Informally (it can be formalized), a match expression has type τ if (1) the matched expression has type $+\{C_1{:}\tau_1, \ldots, C_n{:}\tau_n\}$, (2) for each C_i in the type there is a branch of the form $C_i\ x_i \to e_i$ where e_i has type τ assuming x_i has type τ_i.
   - The typing rule for constructor expressions can just be:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash C\ e : +\{C\ \tau\}}$$

     If that seems odd, read on.

   Come up with three sound and generally useful *subtyping rules* for these sum types and *justify informally* why each rule is sound. Write the rules formally.

   Note: We already have rules like reflexivity and transitivity. Your rules should specifically deal with the new sum types.

5. (Recursive types) In this problem, we show that a typed lambda-calculus with recursive types and *explicit roll and unroll coercions* is as powerful as the untyped lambda-calculus. We give this language the following syntax, operational semantics, and typing rules (where for the sake of part (c) we allow evaluation of the right side of an application even if the left side is not yet a value):

$$\begin{array}{rcl} \tau & ::= & \alpha \mid \tau \to \tau \mid \mu\alpha.\tau \\ e & ::= & x \mid \lambda x{:}\tau.\ e \mid e\ e \mid \mathsf{roll}_{\mu\alpha.\tau}\ e \mid \mathsf{unroll}\ e \\ v & ::= & \lambda x{:}\tau.\ e \mid \mathsf{roll}_{\mu\alpha.\tau}\ v \end{array}$$

$$\frac{e \to e'}{e\ e_2 \to e'\ e_2} \qquad \frac{e \to e'}{e_1\ e \to e_1\ e'} \qquad \frac{e \to e'}{\mathsf{roll}_{\mu\alpha.\tau}\ e \to \mathsf{roll}_{\mu\alpha.\tau}\ e'} \qquad \frac{e \to e'}{\mathsf{unroll}\ e \to \mathsf{unroll}\ e'}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{}{\mathsf{unroll}\ (\mathsf{roll}_{\mu\alpha.\tau}\ v) \to v}$$

$$\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash e_1\ e_2 : \tau_1}$$

$$\frac{\Delta;\Gamma \vdash e : \tau[(\mu\alpha.\tau)/\alpha]}{\Delta;\Gamma \vdash \mathsf{roll}_{\mu\alpha.\tau}\ e : \mu\alpha.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \mu\alpha.\tau}{\Delta;\Gamma \vdash \mathsf{unroll}\ e : \tau[(\mu\alpha.\tau)/\alpha]}$$

   (a) Define a translation from the pure, untyped, call-by-value lambda-calculus to the language above. Naturally, your translation should preserve meaning (see part (c)) and produce well-typed terms (see part (b)). Use $\mathit{trans}(e)$ to mean the result of translating e. You just need to write down how to translate variables, functions (notice the target language has explicit argument types), and applications. The translation must insert roll and unroll coercions exactly where needed. The key trick is to make sure every subexpression of $\mathit{trans}(e)$ has type $\mu\alpha.\alpha \to \alpha$.

(b) Prove this theorem, which implies that if e has no free variables, then $trans(e)$ type-checks: If $\Gamma(x) = \mu\alpha.\alpha \to \alpha$ for all $x \in FV(trans(e))$, then $\cdot;\Gamma \vdash trans(e) : \mu\alpha.\alpha \to \alpha$. (If the theorem is false, go back to part (a) and fix your translation.)

(c) Prove this theorem, which, along with determinism of the target language (not proven, but true), implies that $trans(e)$ preserves meaning: If $e \to e'$ then $trans(e) \to^2 trans(e')$ (notice the 2!). (If the theorem is false, go back to part (a) and fix your translation.) Note: A correct proof will require you to state and prove an appropriate lemma about substitution.

(d) Explain briefly why the theorem in part(c) is false if we replace $\dfrac{e \to e'}{e_1\ e \to e_1\ e'}$ with $\dfrac{e \to e'}{v\ e \to v\ e'}$.

6. (Data Races) Consider these two code fragments in a shared-memory multithreaded language:

   - `while(f()) { y = (x+1)*g(); }`
   - `int t = x+1; while(f()) { y = t * g(); }`

   Note that, like in C/C++/Java, the variable `t` is thread-local. Assume functions `f` and `g` are known to not read or write global integers `x` or `y` (nor `t`) and to not perform any synchronization, but nothing else is known about them statically.

   (a) Sketch how, given an operational semantics for the language, you could prove that it would be meaning preserving to replace the first fragment with the second one *if there is only one thread.*

   (b) There are situations with multiple threads where replacing the first fragment with the second one introduces data races into a previously data-race-free program. Fully describe such a situation. (Therefore, in a language in which data races allow more behaviors, which might include "catch-fire semantics," this transformation would be illegal.) Note: A situation need not be something we expect programmers to do often.

   (c) Come up with a slightly more sophisticated transformation for the first fragment such that:

   - If the original program has no data races, then neither will the transformed program.
   - It has the same single-threaded semantics.
   - Each time the transformed program fragment executes, `x+1` is evaluated no more than once.