CSE505 Concepts of Programming Languages, Assignment 5
Due: Thursday 8 March 2012, 11:00PM

1. (System F and parametricity)

   (a) Give 4 values v in System F such that:
   - $\cdot;\cdot \vdash v : \forall\alpha.(\alpha * \alpha) \to (\alpha * \alpha)$
   - Each v is not equivalent to the other three (i.e., given the same arguments it may have a different behavior).

   For *one* of your 4 values, give a full typing derivation.

   (b) Give 6 values v in Caml such that:
   - v is a closed term of type `'a * 'a -> 'a * 'a` *or a more general type*. For example, `'a * 'b -> 'a * 'b` is more general than `'a * 'a -> 'a * 'a` because there is a type substitution that produces the latter from the former (namely `'a` for `'b`).
   - Each v is not totally equivalent to the other five.
   - None perform input or output.

   (c) Consider System F extended with lets, mutable references, booleans, and conditionals all with their usual semantics and typing:

   $e ::= c \mid x \mid \lambda x{:}\tau.\ .e \mid e\ e \mid \Lambda\alpha.\ e \mid e\ [\tau] \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{ref}\ e \mid !e \mid e := e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{true} \mid \mathsf{false}$

   Unsurprisingly, if v has no free variables, no heap labels, and type $\forall\alpha.(\alpha \to \mathsf{bool} \to \mathsf{bool})$, then $v\ [\tau]\ x\ y$ and $v\ [\tau]\ z\ y$ always produce the same result in any environment where x, y, and z are bound to values of appropriate types. Surprisingly, there exists closed values v of type $\forall\alpha.((\mathsf{ref}\ \alpha) \to (\mathsf{ref}\ \mathsf{bool}) \to \mathsf{bool})$ such that in some environment, $v\ [\tau]\ x\ y$ evaluates to true but $v\ [\tau]\ z\ y$ evaluates to false. Write down one such v and explain how to call v (i.e., what x, y, and z should be bound to) to get this surprising behavior. Hints: This is tricky. Exploit aliasing. You can try out your solution in ML (you do not need any System F features not found in ML), but do not use any ML features like pointer-equality.

2. (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.

For problems 3 and 4, do *not* use mutable references, mutexes, condition variables, fork-join, etc. All you need is thread-creation and the Concurrent ML primitives provided in the `Event` module. Download `hw5code.tar` from the course website. Remember to compile `hw5.ml` via `ocamlc -vmthread -o hw5 threads.cma hw5.ml` or similar. Also note Caml terminates when the main thread terminates, so it is sometimes convenient when testing to block the main thread (e.g., by calling `Thread.yield` in an infinite loop).

3. (Concurrent ML) Implement the first commented-out interface in `hw5.mli`. Define type `barrier` and functions `new_barrier` and `wait`. If a barrier is created by `new_barrier` i and a thread makes one of the first $i-1$ calls to `wait` with that barrier, then the thread should block until the i^{th} call, at which point all i threads should proceed. A thread that calls `wait` with a barrier after there have been i calls with that barrier should block forever.

   How to do it: `new_barrier` should return a channel that a newly created thread receives on. `wait` should send on this channel another channel and then receive on the channel it sends. (It does not matter what is sent on this channel; `()` is a fine choice.) The newly created thread should "remember" how many waiters there are and what channels to send on after the last arrives. The thread can then terminate. You do not need `choose` or `wrap`.

4. (Concurrent ML) Implement the second commented-out interface in `hw5.mli`. This interface is for shared/exclusive locks (better known as readers/writer locks). Function `new_selock` creates a new lock. Functions `shared_do` and `exclusive_do` take a lock and a thunk and run the thunk. An implementation is correct if for each lock lk:

   - If there are any thunks that have not completed, then at least one of these thunks gets to run.
   - While a thunk passed to `exclusive_do` lk runs, no other thunk passed to `exclusive_do` lk or `shared_do` lk runs.
   - If there are no uncompleted exclusive thunks, then the uncompleted shared thunks get to run in parallel.

   (a) A correct solution to this problem has been given to you, but it is much longer and more complicated than necessary. In it, the "server" maintains explicit lists of waiting thunks. Reimplement the interface without these lists. Instead just use a simpler protocol that relies on the fact that CML allows multiple threads to block on the same channel, effectively forming an implicit queue. Hint: You still need `choose` and `wrap`, but you need fewer code cases and fewer total messages.

   (b) Suppose a call to `shared_do` lk blocks before a call to `exclusive_do` lk. For the two implementations of the interface (the one given to you and your simpler one), describe the states under which we are *sure* the call to `shared_do` lk will unblock before the call to `exclusive_do` lk. Explain your answer, which can be different for the two implementations. Note `choose` is nondeterministic.

5. **Challenge Problem:** (Recursive types) We did not study recursive types, in which a type definition can refer to itself, but a formalism is quite concise:

   - In the type $\mu\alpha.\tau$, the type variable α stands for the entire type $\mu\alpha.\tau$. For example, with tuples and sums, a type for binary trees where internal nodes hold ints and leaves hold strings could be $\mu\alpha.\ \mathsf{string} + (\mathsf{int} * \alpha * \alpha)$.
   - To make type-checking easy, we can use "roll" and "unroll" coercions as seen below.

   While the primary motivation for recursive types is to allow for recursive data structures in a typed language, the full power of $\mu\alpha.\tau$ turns out to be more. In this problem, you will 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 $\mathit{trans}(e)$ type-checks: If $\Gamma(x) = \mu\alpha.\alpha \to \alpha$ for all $x \in FV(\mathit{trans}(e))$, then $\cdot;\Gamma \vdash \mathit{trans}(e) : \mu\alpha.\alpha \to \alpha$. (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 $\mathit{trans}(e)$ preserves meaning: If $e \to e'$ then $\mathit{trans}(e) \to^2 \mathit{trans}(e')$ (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 $\dfrac{e \to e'}{v\ e \to v\ e'}$.

What to turn in:

- Hard-copy (written or typed) answers to problems 1, 2, 4b, and optionally problem 5.
- Caml file `hw5.ml` problems 3 and 4a.

Follow the dropbox link on the course website (homework section), follow the "Homework 5" link, and upload your files. If you do not have an electronic copy of your non-code answers in a standard format, you can turn in these problems in Adrian's grad-student mailbox or give them to him directly.