CSE 505, Fall 2003, Assignment 4
Due: 25 November 2003, 10:30AM (firm)

1. (Typed Currying)

   (a) In System F (with pairs), provide an e_1 and e_2 such that

   $$\cdot;\cdot \vdash e_1 : \forall\alpha_1.\forall\alpha_2.\forall\alpha_3.((\alpha_1 * \alpha_2) \to \alpha_3) \to \alpha_1 \to \alpha_2 \to \alpha_3$$

   and

   $$\cdot;\cdot \vdash e_2 : \forall\alpha_1.\forall\alpha_2.\forall\alpha_3.(\alpha_1 \to \alpha_2 \to \alpha_3) \to (\alpha_1 * \alpha_2) \to \alpha_3$$

   Hint: e_1 will return a curried version of its input function. e_2 will return an uncurried version of its input. For all e_3, if $e_2\ [\tau_1][\tau_2][\tau_3]\ (e_1\ [\tau_1][\tau_2][\tau_3]\ e_3)$ typechecks, then it is equivalent to e_3. (In other words, e_1 and e_2 are inverses of each other.) These are "free theorems" indicating that any correct solution will have certain behavior.

   (b) In O'Caml, write functions `curry` and `uncurry` equivalent to e_1 and e_2 above. (Note in O'Caml `*` binds tighter than `->`, so `int*int->int` means `(int*int)->int`.)

   (c) In O'Caml, write functions `not_curry` and `not_uncurry` that have the same types as `curry` and `uncurry` but are not equivalent to them. (You may want to give explicit types to the function arguments so the type-checker does not infer more general types, though there are solutions that do not require doing this.)

2. (Picking on Java) This program type-checks and runs:

```
class C {
  public static void f(Object x, Object arr[]) {
    arr[0] = x;
  }
  public static void main(String args[]) {
    Object o = new Object();
    C []   a = new C[10];
    f(o, a);
  }
}
```

   (a) For this program, where does the type-checker use subsumption? From what type to what type? What is Java's subtyping rule for arrays?

   (b) Does this program execute any downcasts when it runs? What happens when it runs?

   (c) Informally, what is the semantics of array-update in Java? (Start your answer with, "Array update takes an array-object a, an index i, and an object o...". Discuss what exceptions might be thrown under what conditions and what occurs if no exceptions are thrown.)

   (d) Is it possible to compile a Java program without run-time type information, even if the program has no downcasts, method overriding, or reflection? (Note that compilation must preserve the behavior you described in the previous question.)

3. (Strong Evaluator Interfaces) For this problem, we put Dan's homework-3 implementations of `typecheck` and `interpret` in a file `impl.ml` and expose only the interface in `impl.mli`. *Do not change `typecheck` or `interpret`, but use them so that you actually write very little code. Do not change `impl.mli`.*

   Our goal is to prevent clients (code in other files) from being able to cause the `Impl.RunTimeError` exception to be raised. We do this by making sure (nonrecursive) calls to `Impl.interpret` are with `mt_env` and `prog` *such that `prog` typechecks*. Part of our implmentation is "safe" if it cannot violate this goal.

   Ignore the files with "2" in their name until part (e).

(a) Implement `interpret1` such that it raises TypeError if its input does not type-check and returns the evaluation of its argument otherwise. This is safe, but requires type-checking a program every time we run it.

(b) Implement `typecheck2` and `interpret2` such that `typecheck2` returns what `typecheck mt_ctxt` does, and `interpret2` returns the evaluation of its argument, but `interpret2` raises TypeError iff a pointer-equal STLC program has not been previously passed to `typecheck2` and successfully type-checked. (You will need mutable state in `impl.ml` to do this. Given programs x and y, compare them with pointer-equality: $x == y$.) This is safe, but requires state and can leak memory.

(c) Implement `typecheck3` to do this: If the program doesn't type-check raise TypeError. If it does type-check, return a thunk that when invoked returns the evaluation of the program that has been type-checked. This is safe.

(d) Implement `typecheck4` to raise TypeError if the program doesn't type-check and return its argument if it does type-check. Implement `interpret4` to behave just like `interpret`. This is safe. (To see why, look at the type of these bindings in `impl.mli`.)

(e) Build "version 2" by copying your solutions from `impl.ml` to `impl2.ml`. Version 2 is different in only these ways: (1) The abstract syntax for application is `Apply(exp ref,exp)`, i.e., the first part is mutable. (2) We don't bother with a lexer or parser.

For each of `interpret1`, `interpret2`, the result of `typecheck3` e, and `interpret4`, do the following: If the function is still safe, explain why. If it is not, put code in `adversary.ml` that will cause `Impl2.RunTimeError` to be raised when calling the function. (Just build abstract syntax "manually" rather than parsing a program. See `adversary.ml` for details.)

What to turn in:

- Written or typed solutions to 1a, 2, and any parts of 3e that are still safe.
- A file `problem1.ml` with bindings for `curry`, `uncurry`, `not_curry`, and `not_uncurry`.
- A file `impl.ml` which has your additions for problems 3a–3d.
- A file `adversary.ml` which has your additions for problem 3e.