CS152, Spring 2011, Assignment 5
Due: Thursday 14 April 2011, 10:00AM

Last updated: March 30

See also the associated code on the course website.

1. (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 `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.

2. (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 is a closed value[1] of 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 an environment where x and y 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

[1] Recall a closed value has no free variables (and no heap labels).

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. Meta-hint: Feel free to ask for more hints.

3. (Strong Interfaces) This problem investigates several ways to enforce how clients use an interface. The file `stlc.ml` provides a typechecker and interpreter for a simply-typed lambda-calculus. We intend to use `stlc.mli` to enforce that *the interpreter is never called with a program that does not typecheck*. In other words, no client should be able to call `interpret` such that it raises `RunTimeError`. We will call an approach "safe" if it achieves this goal.

   In parts (a)–(d), you will implement 4 different safe approaches, none of which require more than 2–3 lines of code in `stlc.ml`. (Do not change `stlc.mli`.) Files `stlc2.mli` and `stlc2.ml` are for part (e).

   (a) Implement `interpret1` such that it typechecks its argument, raises `TypeError` if it does not typecheck, and calls `interpret` if it does typecheck. This is safe, but requires typechecking a program every time we run it.

   (b) Implement `typecheck2` and `interpret2` such that `typecheck2` raises `TypeError` if its argument does not typecheck, otherwise it adds its argument to some mutable state holding a collection of expressions that typecheck. Then `interpret2` should call `interpret` only if its argument is pointer-equal (Caml's `==` operator) to an expression in the mutable state `typecheck2` adds to. This is safe, but requires state and can waste memory.

   (c) Implement `typecheck3` to raise `TypeError` if its argument does not typecheck, else return a thunk that when called interprets the program that typechecked. This is safe.

   (d) Implement `typecheck4` to raise `TypeError` if its argument does not typecheck, else return its argument. Implement `interpret4` to behave just like `interpret`. This is safe; look at `stlc.mli` to see why!

   (e) Copy your solutions into `stlc2.ml`. Use `diff` to see that `stlc2.ml` and `stlc2.mli` have one small but important change: part of the abstract syntax is mutable.

   For each of the four approaches above, decide if they are safe for `stlc2`. If an approach is not safe, put code in `adversary.ml` that will cause `Stlc2.RunTimeError` to be raised. (See `adversary.ml` for details about where to put this code.)

4. (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 *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 *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 $x \in FV(\mathit{trans}(e))$, 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 *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 $\dfrac{e \to e'}{v\ e \to v\ e'}$.

5. **Challenge Problem:** In class, you saw ML-style type inference (i.e., let-polymorphism) for a small language including constants, functions, applications, and let-bindings. Extend this language with pairs, booleans, and conditionals and implement type inference for this language in ML. (That is, define abstract syntax for expressions and types and write a Caml function that takes an expression that has no type annotations and produces different abstract syntax where every expression is "decorated" with its type.) For an additional challenge, extend the language with "let rec" following ML's restriction that any recursive call of a polymorphic function must instantiate each α with exactly α.