CS152, Spring 2011, Assignment 4
Due: Thursday 31 March 2011, 10:00AM

Last updated: March 21

See also the associated code on the course website.

1. (References and Subtyping) Consider a simply-typed lambda-calculus including mutation (as defined in homework 3), records, and subtyping (as defined in lecture 13). In other words, it has mutable references and immutable records, plus all the subtyping rules considered in lecture. This "combined language" has no subtyping rule for reference types yet (see below).

   (a) Write an inference rule allowing *covariant* subtyping for reference types. Show this rule is *unsound*. To show a rule is unsound, assume the language without the rule is sound (which it is). Then give an example program, show that the program typechecks using the rule, and that evaluating the program can get stuck.

   (b) Write an inference rule allowing *contravariant* subtyping for reference types. Show this rule is *unsound*.

   (c) Write an inference rule allowing *invariant* subtyping for reference types. Invariant subtyping means it must be covariant and contravariant. This rule is sound, but you do not have to show it. However, show that this rule is not *admissable* (i.e., it allows programs to typecheck that could not typecheck before). Keep in mind our language already has reflexive subtyping, so we can already derive $\tau \leq \tau$ for all τ.

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

   - 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 in a type is not signficiant. 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.

   Hint: We have three sound and generally useful subtyping rules for record types. Some of your rules might be very similar to those and others might be analogous but crucially different.

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

3. (Implementing Subtyping) You have been provided an interpreter and typechecker for the language in homework 3, extended with tuples, *explicit* subsumption, and named types. The example program `factorial` uses these new features, but it will not typecheck until you implement subtype checking. Language details:

   - A program now begins with zero or more "type aliases" of the form `type` s `=` τ where `type` is a keyword, s is an identifier, and τ is a type. A type alias makes s a legal type. As for subtyping, $s \leq \tau$ *and* $\tau \leq s$. You may assume without checking that a program's type aliases have no cyclic references (see challenge problem 3(b)) and each alias defines a different type name.
   - The typechecker does *not* allow implicit subsumption. However, if e has type τ and $\tau \leq \tau'$, then the explicit subsumption `(e :` τ'`)` has type τ'. If τ is not a subtype of τ', then `(e :` τ'`)` should not typecheck.
   - Tuple types are written `t1 * t2 ... * tn`. There is no syntax for tuple types with fewer than 2 components even though the interpreter and typechecker support them.
   - Similarly, tuple expressions are written `(e1, e2, ..., en)`.
   - To get a field of a tuple, use `e.i` where `i` is an integer and the fields are numbered left-to-right starting with 1.

   All you need to do is implement the `subtype` function in `main.ml` to support the following:

   - A named type (i.e., type alias) is a subtype of what it aliases and vice-versa.
   - `Int` is a subtype of `Int`.
   - Reference types are invariant as in problem 1(d).
   - Tuple types have width and depth subtyping.
   - Function types have their usual contravariant argument and covariant result subtyping.

   Note: Feel free to use functions from the `List` library to make your solution more concise. Pattern-matching on pairs of types is also very useful.

   Challenge Problems:

   (a) Change `typecheck` to support *implicit* subsumption between type aliases and their definitions (but still require explicit subsumption for all other subtyping).

   (b) Extend your subtype-checker to be sound and always terminate even if the type aliases have cycles in their definitions (e.g., the definition of s_1 uses s_2 and vice-versa; one-type cycles are also a problem). Explain what subtyping you do and do not support in the presence of cycles.

4. (Types for Continuations) Recall how we added first-class continuations to the lambda-calculus with evaluation-context semantics:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathsf{letcc}\ x.\ e \mid \mathsf{throw}\ e\ e \mid \mathsf{cont}\ E \\ v & ::= & \ldots \mid \mathsf{cont}\ E \\ E & ::= & \ldots \mid \mathsf{throw}\ E\ e \mid \mathsf{throw}\ v\ E \end{array}$$

$$\frac{}{E[\mathsf{letcc}\ x.\ e] \to E[(\lambda x.\ e)(\mathsf{cont}\ E)]}$$

$$\frac{}{E[\mathsf{throw}\ (\mathsf{cont}\ E')\ v] \to E'[v]}$$

   Extend the simply-typed lambda-calculus with typing rules for these new constructs. Your rules should be sound and not unreasonably restrictive. Assume we extend the type system with types of the form τ cont. The type τ cont should describe expressions that evaluate to cont E for some E such that $E[v]$ is well-typed for any v with type τ. (We don't care what type $E[v]$ has as long as it has some type.)

   Hint: These three rules are enough given the right hypotheses:

$$\frac{???}{\Gamma \vdash \mathsf{letcc}\ x.\ e : \tau} \qquad \frac{???}{\Gamma \vdash \mathsf{throw}\ e_1\ e_2 : \tau} \qquad \frac{???}{\Gamma \vdash \mathsf{cont}\ E : \tau\ \mathsf{cont}}$$

5. (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 (`problem5/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.

6. **Challenge Problem:** Extend the CPS transformation from lecture 14 to include the translation for pairs and sums as introduced in lecture 11.