# CSE 505, Fall 2008, Assignment 4
# Due: Tuesday 25 November 2008, 5:00PM

Last updated: November 10

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 11). 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) **Challenge Problem:** Extend your type-safety proof from homework 3 to encompass this new language (i.e., with records, subsumption, and the subtyping rules from lecture).

   (b) Write down 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 (as proven in the previous problem). Then give an example program, show that your example typechecks using the rule, and that evaluating the program can get to a stuck state.

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

   (d) Write down 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 $\tau$.

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 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 any context where $x$ and $y$ are bound to values of appropriate types. Surprisingly, there exists closed values $v$ of type $\forall\alpha.((\mathsf{ref}\ \alpha) \to (\mathsf{ref\ bool}) \to \mathsf{bool})$ such that in some environment, $v\ [\tau]\ x\ y$ evaluates to $\mathsf{true}$ but $v\ [\tau]\ z\ y$ evaluates to $\mathsf{false}$. 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. Meta-hint: Feel free to ask for more hints.

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

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` $s$ `=` $\tau$ where `type` is a keyword, $s$ is an identifier, and $\tau$ 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 the second challenge problem) and each alias defines a different type name.
   - The typechecker does *not* allow implicit subsumption. However, if $e$ has type $\tau$ and $\tau \leq \tau'$, then the explicit subsumption `(e :` $\tau'$`)` has type $\tau'$. If $\tau$ is not a subtype of $\tau'$, then `(e :` $\tau'$`)` 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: The sample solution is about 15 lines long. Pattern-matching on pairs of types helps keep things concise as does using a couple functions defined in the `List` library.

   **Challenge Problems:**

   - Change `typecheck` to support *implicit* subsumption between type aliases and their definitions (but still require explicit subsumption for all other subtyping).
   - 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. (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 `Stlc.RunTimeError` to be raised. (See `adversary.ml` for details about where to put this code.)

**What to turn in:**

- Hard-copy (written or typed) answers to problems 1 and 2.
- Caml source code in `main.ml`, `stlc2.ml`, and `adversary.ml` for problems 3 and 4.

Email your source code to Jonathan as `firstname-lastname-hw4.tgz` or `firstname-lastname-hw4.zip`. The code should untar/unzip into a directory called `firstname-lastname-hw4`. Hard copy solutions can be emailed in a pdf file, put in Jonathan's grad student mailbox, or given to him directly.