CSE P505, Winter 2009, Assignment 4
Due: Thursday March 5, 2009, 5:00PM

Last updated: March 1 (fix typo in turn-in instructions)

- This assignment covers type-checking, subtyping, and using abstract types to enforce strong interfaces.
- For problems 1 and 2, see also `prob1.ml` and `lang.pdf`.
- Problem 3 is independent of the other problems. See also `stlc.mli`, `stlc.ml`, `stlc2.mli`, `stlc2.ml`, and `adversary.ml`.
- We have provided a Makefile that creates two programs, `prob1` and `prob3`.
- Turn in your solution via the "Turn-in" link on the course website. Include `prob1.ml`, `stlc.ml`, `stlc2.ml`, `adversary.ml`, and a file for your answer to problem 2. If you do the first challenge problem, include another file. If you do the second challenge problem, include your answer in `prob1.ml`. Do not modify the `.mli` files.
- Understand the course policies on academic integrity (see the syllabus) and challenge problems.

1. In `prob1.ml`, complete the functions `subtype` and `typecheck` to provide a type-checker for the language described in `lang.pdf`. For `subtype t1 t2`, return true if and only if `t1` is a subtype of `t2`. For `typecheck`, raise the `DoesNotTypecheck` exception or return the type of the expression. The exception carries a string: you may find it useful to use different strings for different errors, but the strings will not be graded.

   For subtyping:

   - The only subtype of `IntT` or `BoolT` is itself.
   - Subtyping for arrow types is as usual.
   - Subtyping for records includes:
     - Width and permutation as in class.
     - Depth: Sound yet expressive depth subtyping will depend on a field's access modifier. The correct rules are for you to figure out.
     - Subtyping based on different access modifiers. Again, the correct rules are left to you.

   For typechecking:

   - The "natural" typechecking rules are mostly left to you. See `lang.pdf`.
   - `If(e1,e2,e3)` typechecks if `e1` has a subtype of `BoolT`[1] and one of `e2` and `e3` has a subtype of the other. The supertype of these two types is the type of the whole expression. (This is the rule in languages like Java; see the challenge problems.)
   - `RecordV e` should not typecheck because it should not appear in source programs.
   - For `Get` and `Set`, be sure to consult the access modifier.
   - Call `checkType` on every explicit type in the program. The result is `()`, but it may raise an exception. For `RecordE`, be sure the field names are unique. Together, these two checks ensure no record type will ever have repeated fields.

   Hints:

   - For subtyping, do not include code for reflexivity and transitivity. In other words, create a straightforward algorithm that *is* reflexive and transitive, though that may not be obvious.
   - The sample solution for `subtype` is about 15 lines.

[1] It happens that means `e1` has exactly type `BoolT`, but in general we always allow subtypes.

- The typechecker does not need a subsumption rule. Instead you just use `subtype` in any place subsumption may be needed. For example, in an applicaton, see if the argument has a subtype of the type the function expects.
- The sample solution for `typecheck` is about 65 lines.

2. Consider these two functions in our language (only their types differ):

```
Lam("x",RecordT[("l",IntT,Read)],
    RecordE[("l1",Get(Var("x"),"l")); ("l2",Var("x"))])
Lam("x",RecordT[("l",IntT,Both)],
    RecordE[("l1",Get(Var("x"),"l")); ("l2",Var("x"))])
```

   In English, describe:

   - A situation where using the first function would typecheck but the second would not
   - A situation where using the second function would typecheck but the first would not
   - How to use *bounded polymorphism* to extend our type system to overcome this code duplication — A few sentences is plenty, we do not need a full language design and description of the type system

   Hint: Bounded polymorphism for *types* is one topic in Lecture 8, but here think about access modifiers.

3. 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 a mutable global data structure 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. **(Challenge Problems)** In problem 1, our typing rule for `If(e1,e2,e3)` is quite restrictive. A better rule might allow that if `e2` has type `t2` and `e3` has type `t3`, then the whole expression has type `t4` where `t4` is *the least common supertype* of `t2` and `t3`. By definition, the least common supertype would be a type `t4` such that for all types `t5` at least one of the following holds:

   - `t5` is not a supertype of `t2`
   - `t5` is not a supertype of `t3`
   - `t5` is a supertype of `t4`

   One minor problem for our language is that due to permutation least common supertypes for our language are not always unique, so we should more properly say ***a** least common supertype.* One major problem for our language is that least common supertypes do not always exist even when there are common supertypes.[2]

   (a) Give an example of two types `ta` and `tb` for which there are common supertypes but no least common supertypes. Give two types `tc` and `td` such that (1) `tc` and `td` are supertypes of `ta` and `tb`, (2) neither `tc` nor `td` is a subtype of the other, and (3) there is no type that is a supertype of `ta` and `tb` and a subtype of `tc` and `td`.

   Hint: You can do this with just record types, access modifiers, and `IntT` — no function types needed.

   (b) If we ignore the `Read` and `Write` access modifiers, i.e., assume our language has only the `Both` modifier, then least common supertypes do exist whenever two types have a common supertype. Given this assumption, implement a function `least_common_supertype` of type `typ -> typ -> typ option`, returning `None` if the types have no common supertype, else `Some t` where `t` is a least common supertype.

   Hint: You will also have to implement a function that computes greatest common subtypes.

[2] By the way, Java and C# also have this problem but for different reasons.