Homework # 9 **SOLUTION**
due April 29, 13:00

This assignment is due after Easter break. As well as SASyLF, you will need to use the `fullref` and `fulluntypedref` checkers (either Scala or OCaml). Please turn in all parts by email to `scmalte@inf.ethz.ch` as three separate files: `algo.slf`, `varinc.f` and `object.f`.

1 Reading

Please read Chapters 16 and 18 in the textbook. If this homework assignment's § 3 makes no sense to you, please read Chapter 18 (again).

2 Algorithmic Subtyping

Proof the results about algorithmic subtyping: Proposition 16.1.5, Theorem 16.2.4, Theorem 16.2.5 in the given skeleton file.

3 Object Encodings

1. Extend the counter class (from 18.6) with a new "class" of variable counter class, with a new method `setinc` that sets the step size which is used by a new definition for `inc` that increments by the current step size. The code should use ascription (reminder, see 11.4) so that the record types are given useful names. Use the `fullref` checker to check your code. Here is a simple test case:

```
vc1 = newVarCounter(2);
vc2 = newVarCounter(3);
inc3 vc1;
inc3 vc2;
vc1.get unit;
vc2.get unit;
vc1.setinc(1);
inc3 vc1;
inc3 vc2;
vc1.get unit;
vc2.get unit;
```

   The last two numbers printed should be 10 and 19 (if you start your counters at 1).

   Place your code (including the definitions of `Counter`, `CounterRep` and related functions following the style used in the textbook) in a file varinc.f. Note that open recursion is not needed for this program.

```
plus = fix (lambda plus:Nat -> Nat -> Nat.
              lambda x:Nat . lambda y:Nat .
                if iszero x then y else succ(plus (pred x) y));

Counter = {get:Unit -> Nat, inc:Unit -> Unit};
CounterRep = {x: Ref Nat};

counterClass = lambda r:CounterRep.
      {get = lambda _:Unit . !(r.x),
       inc = lambda _:Unit. r.x := succ(!(r.x))} as Counter;

newCounter =
  lambda _:Unit. let r = {x = ref 1} in counterClass r;

inc3 = lambda c :Counter . (c.inc unit; c.inc unit; c.inc unit);

VarCounter = {get:Unit -> Nat, inc:Unit -> Unit, setinc: Nat -> Unit};
VarCounterRep = {x: Ref Nat, s: Ref Nat};

varCounterClass = lambda r:VarCounterRep .
      let super = counterClass r
      in { get = super.get,
           inc = lambda _:Unit. r.x := plus (!(r.x)) (!(r.s)),
           setinc = lambda x:Nat. r.s := x } as VarCounter;

newVarCounter =
  lambda n:Nat .
    varCounterClass {x = ref 1, s = ref n};
```

2. The approach to open recursion given in the textbook puts `self` in the environment, which in principle means we copy the method bodies for each object. Mainstream OO languages instead pass `self` as a parameter to the method. Thus calling (say) the `set` method of counter c, one would write `(c.inc) c` assuming we have a counter such as:

```
c = { get = lambda self . !(r.x) ,
      set = lambda self . lambda i . r.x := i,
      inc = lambda self . (self.set) self (succ ((self.get) self)) };
```

   Redo the "open recursion through self" examples from 18.10 using this approach (classes SetCounter and InstrSetCounter). Use fulluntypedref to test your implementation. You will be syntactically required to specify types for lambda-bound variables, but the types are ignored. Define `type SelfType = Unit;` and then use this for the type of `self`. Put your results in a file object.f.

```
SelfType = Unit;
SetCounter = { get : SelfType -> Nat,
               set : SelfType -> Nat -> Unit,
               inc : SelfType -> Unit };
SetCounterRep = {x : Ref Nat };

setCounterClass =
   lambda rep: SetCounterRep.
          {get = lambda s:SetCounter . !(rep.x),
   set = lambda s:SetCounter . lambda n:Nat . (rep.x) := n,
           inc = lambda s:SetCounter . (s.set) s (succ ((s.get) s))}
as SetCounter;

makeCounter =
  lambda i:Nat . setCounterClass {x = (ref i)};

c = makeCounter(3);
(c.get) c;
(c.inc) c;
(c.get) c;

InstrSetCounter = { get : SelfType -> Nat,
                    set : SelfType -> Nat -> Unit,
                    inc : SelfType -> Unit,
                    accesses : SelfType -> Nat };
InstrSetCounterRep = {x : Ref Nat, a : Ref Nat};

instrSetCounterClass =
  lambda rep : InstrSetCounterRep .
    let super = setCounterClass rep in
      { get = super.get,
        set = lambda s:InstrSetCounter .
                lambda i:Nat .
                  (super.set s i;
                    (rep.a) := succ (!(rep.a))),
        inc = super.inc,
        accesses = lambda s:InstrSetCounter . !(rep.a)}
      as InstrSetCounter;

makeInstrSetCounter =
  lambda i :Nat . instrSetCounterClass { x = ref i, a = ref 0 };

i = makeInstrSetCounter 3;
(i.inc) i;
(i.get) i;
(i.accesses) i;
(i.set) i 10;
(i.get) i;
(i.accesses) i;
```

3. Of course `Unit` is the wrong type, that's why we are using an untyped checker. Cite one published reference on how to type "self" in object-oriented programming. Describe the proposal in two sentences.

Kim B. Bruce, Luca Cardelli, Benjamin C. Pierce. Comparing object encodings. *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science Volume 1281, 1997, pp 415-438

The most complex encoding they compare, the ORBE encoding defined:

```
ORBE(I) = Rec(X) Some(Y<:X) Y * (Y->I(Y))
```

This uses recursive types, bounded existental types and higher ordered types (passing `I`, a type constructor, as a parameter). The self-type is `Y`.

Homework 9.5 **SOLUTION**
due April 29, 13:00

This assignment is an optional assignment. If you do this assignment, the score can replace a lower score on an earlier homework. If you do worse than all previous homeworks, it will be ignored. (In other words, doing it cannot hurt your grade in this course.)

Send the completed `fj.slf` by email to `scmalte@inf.ethz.ch` and bring the written answers to §6 to lecture.

4 Reading

Please read Chapter 19.

5 Soundness of FJ

Complete the SASyLF proofs in the skeleton file (`fj.slf`) to prove that FJ is sound. FJ uses algorithmic typing rather than T-Sub. This means that the preservation theorem must be permitted to compute a *subtype* for the result. On the other hand, the canonical forms lemma doesn't need to worry about subtyping any more.

6 Discussion

Even in the absence of interfaces and conditional expressions, Java (and C++, C#, Scala) has problems with subsumption (T-Sub) because of static overloading resolution. Give a simple Java program that would type-check (and run) if subsumption were permitted for any expression, but that does not type-check in standard Java unless up-casts are added (in the place where T-Sub is used). Note that up-casts never fail. Hint: Your example will need to use overloaded functions where the overloadings have different result types. (You may use C++, C# or Scala if you know these languages better than Java.)

```
public class Main {
  public static void main(String[] args) {
    Main m = new Main();
    int result = m.foo((Object)m); // without upcast, this code won't compile
    System.out.println("The answer is " + result);
  }

  public int foo(Object x) {
    return x.hashCode();
  }

  public String foo(Main m) {
    return "Hello";
  }
}
```