Exercise 1

Generics and Assignments

Language with Generics and Lots of Type Annotations

Simple language with this syntax

types: T ::= Int | Bool | T => T | [Ident]T

expressions: E ::= ident

| **{ val** ident:T = ident(ident); E **}**

| **{ val** ident:T = {**(**Ident**:**T**)**=>E}; E **}**

Initial environment

plus : Int => (Int => Int) one : Int two : Int

less : Int => (Int => Boolean)

A generic type [Ident]T can be used as T[Ident:=T'] , which we call an *instance* of [Ident]T

Example: { **val** f: [A](A => A) = {(x:A)=>x};

(type checks) { **val** x : Int = f(one);

1) give rules { **val** above : Int => Bool = less(one);

2) type check { **val** outcome : Bool = above(two);

{ **val** res : Bool = f(outcome); res }}}}}

Add Assignment Statements

expressions: E ::= ident

| **{ var** ident:T = ident(ident); E **}**

| **{ var** ident:T = {**(**Ident**:**T**)**=>E}; E **}**

| **{** ident = ident; E **}**

Use same rules for ‘var’ as for ‘val’

Give rule for assignment statement that are as permissive as possible, but sound

Try to Type Check These Examples

1)

```
{ var f: [A](A => A) = {(x:A)=>x}
  var p : Int => Int = plus(one)
  f = p
  var above : Int => Bool = less(one)
  var outcome : Bool = above(two)
  var res : Bool = f(outcome);   res }}}}}
```

2)

```
{ var f: [A](A => A) = {(x:A)=>x}
  var p : Int => Int = plus(one)
  p = f
  var x : Int = p(one)
  var above : Int => Bool = less(x)
  var outcome : Bool = above(two)
  var res : Bool = f(outcome);   res }}}}}
```

Subtyping

- Suppose we wish to introduce subtyping rule into the previous system
- There should be some sort of subtyping relation between [Ident]T and its instance T[Ident:=T'] .
  Which type should be subtype of which?

Exercise 2

Computing Stack Depth

Control-Flow Graph with Bytecodes

- Consider assignment statements
- Right hand sides have only
  - integer binary minus (-)
  - integer multiplication (*)
  - local variables
- We compile them into iload,istore,isub,imul
- Consider sequence of such statements as a simple control-flow graph (a line)

CFG for Expression. Stack Depth

Assume x,y,z are in slots 1,2,3

Statement

$$x = x*(x - y*z - z*x)$$

Compute for how many integers stack content increased after every point, relative to initial size

Design data-flow analysis for CFG containing these bytecode instructions that maintains interval of possible increases of stack sizes (stack depth), starting from entry

Define analysis domain as arbitrary intervals of stack size, and give transfer functions for iload,istore,isub,imul.

Run Analysis on This Program

```
x = y
while (x > 0) {
  x = y – x – z
}
```

What is the maximal number of steps for such analysis for:

- arbitrary graphs with these bytecodes
- graphs obtained by compiling Tool programs

(if the analysis is extended with all bytecode instructions)

Constant Stack Depth

- Consider CFGs with bytecodes, generated from Tool
- Suppose we wish to store local variables in the memory and temporary variables in registers
- Give an upper bound on the number of registers and a way to convert bytecodes

  iload,istore,isub,imul

  into instructions that use registers

Exercise 3

Pattern Matching

Matching on Polymorphic Lists

Suppose we have language with these types:

Int, Bool

List[T] for any type T

(T1,T2) if T1,T2 are types

Patterns have types:

true,false : Bool integer constant : Int Nil[T]:List[T]

(x :: xs) : List[T] where x:T and xs:List[T]

(x,y) : Tx, Ty where x:Tx, y:Ty

Consider expressions containing the above constructs as expressions, in addition to the expression of the kind:

e match { case p1 => e1; … ; case pN => eN }

Give type checking rules for this language, and pay attention to introducing and removing fresh variables into the scope. Type rules should prevent the use of unbound variables.