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 [A]T can be used as T[A:=T'] ,
which we call an *instance* of [A]T. e.g. Int=>Int instance of [A](A=>A)

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 }}}}}

Here are the interesting cases, involving instantiation and introduction of generic variables

$$\frac{(x, T_1) \in \Gamma \qquad T_1 = [A]T_2 \qquad T = T_3[A := T_3]}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \uplus \{(y, T_y)\} \vdash E \qquad (g, T_1) \in \Gamma \qquad (x, T_2) \in \Gamma \qquad T_1 = [A](T_2' \Rightarrow T_y') \qquad T_2 = T_2'[A := T_3] \quad T_y = T_y'[A := T_3]}{\Gamma \vdash \{\text{val } y : T_y = g(x);\ E\}}$$

Type equation here.

FRESH

$$A \notin FV(\Gamma)$$

$$\frac{\Gamma \uplus \{(g, T_g)\} \vdash E_2 \qquad \Gamma \uplus \{(x, T_x)\} \vdash E_1 : T_1 \qquad T_g = [A](T_x \Rightarrow T_1)}{\Gamma \vdash \{\text{val } g : T_g = \{(x : T_x) \Rightarrow E_1\};\ E_2\}}$$

Adding 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, and still sound

$$\frac{\Gamma \vdash E \qquad (x, T_x) \in \Gamma \qquad (y, T_y) \in \Gamma \qquad T_y = [A]\,T_0 \qquad T_x = T_0[A := T_3]}{\Gamma \vdash \{\, x = y\,;\, E \,\}}$$

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 }}}}}
```

breaks

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 }}}}}
```

works

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?

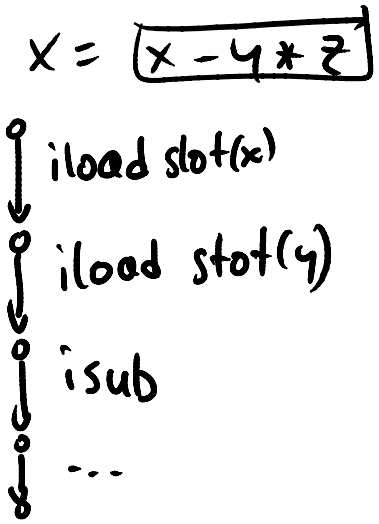
$$[A]T <: T[A:=T_3]$$

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)

Observation: we can use constant propagation to compute places where stack depth is constant

Remark: JVM class loader ensures that stack depth for each progrma poitn is constant, otherwise rejects the bytecode.

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.

Sketch of Solution: Constants

$$\frac{\Gamma \vdash e : T1 \quad \Gamma \vdash_{T1} \{ \text{case } p_i \Rightarrow e_i \} : T \text{ for all } i,\ 1 \le i \le N}{\Gamma \vdash e \text{ match } \{ \text{case } p_1 \Rightarrow e_1; \ldots ; \text{case } p_N \Rightarrow e_N \} : T}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash_{Bool} \{ \text{case true} \Rightarrow e \} : T}$$

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash_{Int} \{ \text{case } K \Rightarrow e \} : T}$$

K is any integer literal, such as 42

Sketch of Solution: Non-nested lists

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash_{List[H]} \{ \text{case Nil}[H] \Rightarrow e \} : T}$$

$$\frac{\Gamma \oplus \{(x,H),(xs,List[H])\} \vdash e : T}{\Gamma \vdash_{List[H]} \{ \text{case } (x::xs) \Rightarrow e \} : T}$$

From Nested Patterns to Tuples

$$\frac{\Gamma \vdash_{(T1,\ldots,Tk,H,List[H])} \{\text{case } (x1,\ldots,xk,p1,p2) \Rightarrow e\} : T \qquad k \geq 0}{\Gamma \vdash_{(T1,\ldots,Tk,List[H])} \{\text{case } (x1,\ldots,xk,(p1::p2)) \Rightarrow e\} : T}$$

Handling Tuples

$$\frac{\Gamma \oplus \{(x1,T1),\ldots,(xn,Tn)\} \vdash e : T \qquad xi \text{ – all variables}}{\Gamma \vdash_{(T1,\ldots,Tn)} \{\text{case } (x1,\ldots,xn) \Rightarrow e\} : T}$$

(also add rules to eliminate constant parts from a tuple)