Midterm II Solution
CS164, Spring 2014

April 7, 2014

- Please read all instructions (including these) carefully.
- **This is a closed-book exam. You are allowed a one-page handwritten cheat sheet.**
- Write your name, login, and SID.
- There are TODO pages in this exam and 3 questions, each with multiple parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 15 minutes to work on the exam.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. **Do not** use any additional scratch paper.
- Solutions will be graded on correctness and ***clarity***. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.
- No electronic devices are allowed, including **cell phones** used merely as watches. Silence your cell phones and place them in your bag.

LOGIN: ______________________________

NAME: ______________________________

SID: ______________________________

Problem	Max points	Points
1	45	
2	35	
3	20	
Sub Total	100	

1 Type System

(a) Explain a relationship between values and types. **[4 points]**

Type is a set of values, as well as a set of operations on those values.

(b) Define soundness of a type system. **[4 points]**

Type system is sound == "program P is well typed => for all expression E in P, runtime_type(E) is subtype of static_type(E)".

(c) What is an advantage of a statically typed language? **[4 points]**

No runtime type error / free from runtime type check / compile time bug detection

(d) What is a limitation of a statically typed language? **[4 points]**

some valid program may be rejecter / less flexible / slow prototyping

(e) Exactly one of the following COOL subtyping rules is sound for all classes C. Circle the sound rule. **[4 points]**

$C \leq \text{SELF\_TYPE}_C$ $\qquad\qquad$ $\text{SELF\_TYPE}_C \leq C$

The following COOL type rule is unsound and the rule has a single bug.

$$\frac{O,M,C \vdash e_1 : Bool \qquad O,M,C \vdash e_2 : T_2 \qquad O,M,C \vdash e_3 : T_3 \qquad T_2 \leq T_3}{O,M,C \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T_2}$$

(f) Illustrate the bug with a simple program. **[5 points]**

(if false then 1 else new Object) + 1

(g) Fix the type rule. Explain your answer in one or two sentences. **[6 points]**

Fix 1: use lub(T2, T3) as a final type, and remove T2 <= T3
Fix 2: flip T2 <= T3 to T3 <= T2
Fix 3: use T3 as a final type.

Explain: The main problem is that the type system unsoundly propagates "more specific" type. All three solutions will fix the type system to propagate a sound (general) type.

The following COOL type rule is unsound. The rule has one bug.

$$\frac{O[T/x],M,C \vdash e_1 : T_1 \qquad O[T/x],M,C \vdash e_2 : T_2 \qquad T_1 \leq T}{O,M,C \vdash \texttt{let } x : T \leftarrow e_1 \texttt{ in } e_2 : T_2}$$

(h) Illustrate the bug with a simple program. **[5 points]**

let x:Object <- new Object in let:Int x <- x+1 in x

(i) Fix the type rule. Explain your answer in one or two sentences. **[4 points]**
Fix: remove [T/x] from the first hypothesis.

Explain: This will prevents type system from propagating incorrect type information about the let bounded variable to the initialization expression.

The COOL language supports SELF_TYPE as a mechanism to bring a flexibility by using static type information available at method dispatch time.

(j) The following COOL program doesn't type check. Fix the program using SELF_TYPE. **[5 points]**

```
class A {
	copy(): A { new A };
	a:A <- (new A).copy();
};

class B inherits A {
	b:B <- (new B).copy();
};
```

Solution: either

```
	copy(): SEFL_TYPE { new SELF_TYPE };
```

or

```
	copy(): SELF_TYPE { self };
```

2 Code Generation

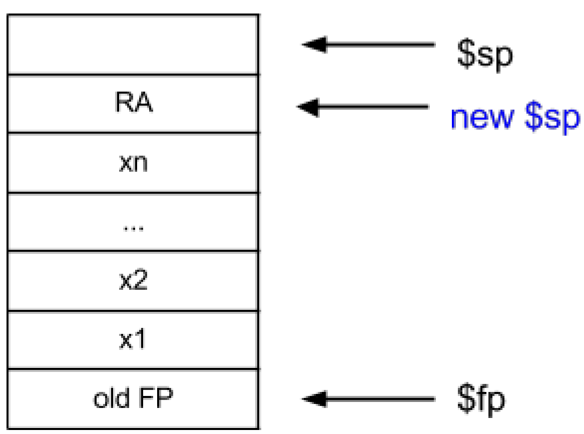
Recall the following MIPS instructions:

- `sw $a0 n($sp)` - store the value of the accumulator at address `n + $sp`
- `lw $ra n($sp)` - load the return address with the value stored at address `n + $sp`
- `addiu $sp $sp n` - adjust the value of the stack pointer by `n`
- `move $a0 $t1` - move the content of `$t1` into `$a0`
- `jal f` - jump to address f and store the address of the next instruction in register `$ra`
- `jr $ra` - jump to address stored in register `$ra`

Imagine a compiler that uses a variation of the calling convention that we used in class. We give below the code that this compiler generates for a function definition. On entry to a function the return address is in `$ra` and on exit the result value is in `$a0`.

```
cgen( def f(x1, x2) = e ) =
        push $ra
        cgen(e)
        lw $ra 4($sp)
        addiu $sp $sp 4
        j $ra
```

Below we give a diagram of the stack where a function and the function body (the `cgen(e)` above) is being evaluated.



(a) Draw on the diagram above where does the `$sp` points to when a function call is about to end, right before `j $ra` in the code above. **[4 points]**

(b) Write the code for accessing the *4*th argument **[4 points]**

```
cgen(x4) =
        lw $a0 -16($fp)
```

(c) Write the code for function call. **[21 points]**

```
cgen( f(e1, e2) ) =

        sw  $fp 0($sp)

        addiu $sp $sp -4

        cgen(e1)

        sw $a0 0($sp)

        addiu $sp $sp -4

        cgen(e2)

        sw $a0 0($sp)

        addiu $sp $sp -4

        addiu $fp $sp 12

        jal f

        move $sp $fp

        lw $fp 0($sp)
```

(d) You notice that we push $ra to the stack when setting up the activation record. Explain in one sentence if callee doesn't store $ra on invocation entrance, what kind of modification you need to make to (c)? **[6 points]**

We need to store the content of $ra before jumping to *f* somehow and restore its value after function call terminates.

3 Operational Semantics

(a) Fill in the evaluation derivation trees below using COOL's operational semantics. Every box should be filled in. You do not need to write anything outside the boxes. Where necessary, contextual information appears below the problem. **[5 points]**

$$\dfrac{\dfrac{\boxed{} \qquad \boxed{}}{\boxed{} \vdash \boxed{} : \boxed{}} \qquad \boxed{E(x)=} \qquad \boxed{S'=}}{so, S, E \vdash x \leftarrow y : \boxed{}}$$

Context:

E(x) = l1 E(y) = l2 S(l1) = Int(9) S(l2) = Int(6)

Solution:

$$\dfrac{\dfrac{E(y) = l_2 \qquad S(l_2) = Int(6)}{so, S, E \vdash y : Int(6), S} \qquad E(x) = l_1 \qquad S' = S[Int(6)/l_1]}{so, S, E \vdash x \leftarrow y : Int(6), S'}$$

Consider the following modified operational semantics rule for **while** in COOL.

$$\dfrac{so, S, E \vdash e_1 : Bool(false), S'}{so, S, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int(0), S'}\ [\textsc{Loop-False}]$$

$$\dfrac{\begin{array}{c} so, S, E \vdash e_1 : Bool(true), S_1 \\ so, S_1, E \vdash e_2 : v_2, S_2 \\ so, S_2, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : v_3, S_3 \end{array}}{so, S, E \vdash \texttt{while}\ e_1\ \texttt{loop}\ e_2\ \texttt{pool} : Int(1), S_3}\ [\textsc{Loop-True}]$$

(b) Explain the return value of this new **while** expression in one or two sentences **[5 points]**

The loop expression returns 0 if the loop body is not not executed at all.
Otherwise, the loop expression will return 1.

(c) Now imagine that we want a **while** loop that counts its number of iterations modulo 2. The value of such a **while** loop should be an integer object that holds the number of times the body has been evaluated modulo 2. Give the operational semantics rules for this new **while** expression. **[10 points]**

(Hint: Your answer will contain three cases. The rule given in 3(b) has two cases)

Solution:

$$\frac{so, S, E \vdash e_1 : Bool(false), S'}{so, S, E \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : Int(0), S'}$$

$$\frac{\begin{array}{c} so, S, E \vdash e_1 : Bool(true), S_1 \\ so, S_1, E \vdash e_2 : v_2, S_2 \qquad so, S_2, E \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : Int(0), S_3 \end{array}}{so, S, E \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : Int(1), S_3} \text{[LOOP-TRUE0]}$$

$$\frac{\begin{array}{c} so, S, E \vdash e_1 : Bool(true), S_1 \\ so, S_1, E \vdash e_2 : v_2, S_2 \qquad so, S_2, E \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : Int(1), S_3 \end{array}}{so, S, E \vdash \texttt{while } e_1 \texttt{ loop } e_2 \texttt{ pool} : Int(0), S_3} \text{[LOOP-TRUE1]}$$