CS143 Final
Spring 2017

- Please read all instructions (including these) carefully.
- There are 6 questions on the exam, all with multiple parts. The exam is intended to be a 2 hour exam, but you have the full 3 hours to take it.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: ____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: ____

Problem	Max points	Points
1	20	
2	30	
3	20	
4	15	
5	25	
6	10	
TOTAL	120	

1. **Garbage Collection** (20 points)

Assume that in its steady state a program P allocates data at a constant positive rate, but the size of the reachable data is always a constant b bytes.

(a) Assume P's memory is managed using Mark and Sweep garbage collection. If we run P on a machine with n bytes of memory, and then again on an otherwise identical machine with $2n$ bytes of memory, do we expect P to run faster, slower, or about the same speed on the machine with the larger memory? Explain your answer. Assume $n > 10b$.

Answer: About the same. Since the total memory is much larger than the reachable data, the mark-and-sweep collections are dominated by the time sweeping through memory, $O(n)$. When you double the memory size, the collections take about twice as long; however, they happen with about half the frequency.

(b) Assume P's memory is managed using Stop and Copy garbage collection. If we run P on a machine with n bytes of memory, and then again on an otherwise identical machine with $2n$ bytes of memory, do we expect P to run faster, slower, or about the same speed on the machine with the larger memory? Explain your answer. Assume $n = 3b$.

Answer: Faster. Each collection will take time $O(b)$. When $n = 3b$, the application only has $1.5b$ memory available for it to use, meaning only $0.5b$ for uncollected, unreachable data; when $n = 6b$, it has $3b$ memory available to it, which means $2b$ for uncollected, unreachable data. Therefore with $n = 6b$ memory collections happen with about $1/4$ the frequency but the same duration.

(c) Assume P's memory is managed using reference counting and that all of P's data structures are acyclic. If we run P on a machine with n bytes of memory, and then again on an otherwise identical machine with $2n$ bytes of memory, do we expect P to run faster, slower, or about the same speed on the machine with the larger memory? Explain your answer. Assume n is only slightly larger than b.

Answer: About the same. Reference counting on acyclic data structures always frees memory immediately when it becomes unreachable and thus the program will never have more than b bytes in use. Therefore increasing the available memory should have no effect on performance.

2. **Flow Analysis** (30 points)

(a) Assume that *locks* have two operations: `lock(l)` and `unlock(l)`, where `l` is an identifier (a variable name) of type `Lock`. We will design a flow analysis for analyzing the behavior of locks. Assume a lock can be in one of four states: *locked* (a lock `l` is always in this state immediately after an operation `lock(l)`), *unlocked* (a lock `l` is always in this state immediately after an operation `unlock(l)`, *any* if we do not know whether the lock is *locked* or *unlocked*, and *don't care* if it doesn't matter whether the lock is locked or unlocked because a program point cannot be executed.

i. What is the appropriate ordering on the four values *locked*, *unlocked*, *don't care* and *any*? **Answer:**

$$any \geq locked, unlocked \geq don't\ care$$

ii. Define a flow analysis transfer function for `lock(l)`: Given the state of `l` before this statement, what is the state of `l` after the statement? Assume there is only one lock `l` (i.e., your solution needs to work only for the single lock `l`).

Answer:

$$C(lock(l), l, out) = \begin{cases} don't\ care & C(lock(l), l, in) = don't\ care \\ locked & otherwise \end{cases}$$

iii. Describe the flow analysis transfer function for `unlock(l)`: Given the state of `l` before this statement, what is the state of `l` after the statement? Assume there is only one lock `l` (i.e., your solution needs to work only for the single lock `l`).

Answer:

$$C(unlock(l), l, out) = \begin{cases} don't\ care & C(unlock(l), l, in) = don't\ care \\ unlocked & otherwise \end{cases}$$

iv. For a statement s and its predecessors $p_1, \ldots, p_n$, describe the state of `l` at the program point before s given the the state of `l` at the program point immediately after all the p_i's.

Answer: $C(s, l, in)$ is the least upper bound in the lattice of the program points following $p_1, \ldots, p_n$. Equivalently stated, given $x_1, \ldots, x_n$ where $x_i = C(p_i, l, out)$ we can compute $C(s, l, in)$ as follows:

- If $x_1 = x_2 = \cdots = x_n = \mathit{don't\ care}$, then $C(s, l, in) = \mathit{don't\ care}$.
- If some x_i are `don't care` and the others are `locked`, then $C(s, l, in) = \mathit{locked}$.
- If some x_i are `don't care` and the others are `unlocked`, then $C(s, l, in) = \mathit{unlocked}$.
- Otherwise, $C(s, l, in) = \mathit{any}$.

v. Locks should observe the following restriction: applications of `lock(l)` and `unlock(l)` must alternate: there cannot be two `lock`s or `unlock`s of the same lock in a row. How can you use the results of the flow analysis for a lock `l` to issue a warning when a program *might* violate this restriction and an error when a program *must* violate this restriction?

Answer:
For a statement $s = \mathit{lock}(l)$ we generate a warning if $C(s, l, in) = \mathit{any}$ and an error if $C(s, l, in) = \mathit{locked}$.
For a statement $s = \mathit{unlock}(l)$ we generate a warning if $C(s, l, in) = \mathit{any}$ and an error if $C(s, l, in) = \mathit{unlocked}$.

(b) Fill in the blanks in the body of `foo` with variable names or constants such that the following requirements are met:

- The return value of `foo` is 10.
- `c` has a constant value, but the combination of the constant propagation and constant folding optimizations presented in lecture cannot determine that `c` is a constant.
- The assignment `x = 1` can be proven to be dead, and this is the *only* dead assignment.

You should assume the omitted predicates marked with * are sometimes true and sometimes false. Otherwise, they play no role in this problem. Fill in liveness information at the points requested. The only result live on exit from the procedure is the return value. Assume when the program runs that every assignment statement in the program executes at least once, that all variables are undefined until assigned, and that the missing predicates are such that no undefined reads occur.

Answer: Note that *d*, *e*, and *f* would be reported live by a liveness analysis. Yet we can manually deduce that these values are dead. We will accept either answer.

```
void foo() {
  int a, b, c, d, e, f, x;
  x = 1;
  // LIVE VARIABLES: d, e, f
  //
  if * {
    a = _7_;
    b = _0_;
  } else {
    a = _0_;
    b = _7_;
  }
  c = _a_ + _b_;
  // LIVE VARIABLES: c, d, e, f
  //
  while *
    if * {
      if * {
        d = e;
      } else {
        e = _f_;
      }
    } else {
      f = 3;
    }
  return d + c;
}
```

3. **Operational Semantics** (20 points)

Recall the proposal to add an array type to Cool in Written Assignment 3 and the immutable variant that was chosen to ensure consistent subtyping. In this problem, we will specify an operational semantics for constructing and manipulating immutable arrays. The Cool grammar is extended as follows:

$$\begin{aligned} \text{oel} &::= \text{expr},\ \text{oel} \mid \epsilon \\ \text{el} &::= \text{oel expr} \\ \text{expr} &::= \text{newarray TYPE[el} \mid \epsilon\ ] \\ &\mid \text{map(ID : TYPE} \rightarrow \text{expr}_1,\ \text{expr}_2) \end{aligned}$$

Note that the square brackets in the array constructor enclose a list of expressions that are evaluted to initialize the elements of the array. In defining the operational semantics, use $[v_1, v_2, \ldots, v_n]$ to denote an *array value* containing elements v_1 to v_n. For example $S[[v_1, \ldots, v_n]/l]$ updates the store S at location l to hold the array value $[v_1, \ldots, v_n]$.

(a) Give a Cool operational semantics rule for the array constructor. The initialization expressions are evaluated left to right. You may use $[e_1, e_2, \ldots, e_n]$ to denote the list of initialization expressions.

Answer:

$$\frac{so, S_i, E \vdash e_i \mapsto v_i, S_{i+1} \text{ for } 1 \leq i \leq n}{so, S_1, E \vdash \text{newarray TYPE}[e_1, \ldots, e_n] \mapsto [v_1, \ldots, v_n], S_{n+1}}$$

(b) The map operator creates a new array. The ith element of the new array is the result of evaluating the map expression expr$_1$ with the identifier named in the map bound to the ith element of the old array expr$_2$. The elements $[v_1, \dots, v_n]$ of the new array are initialized in left-to-right order. Give a Cool operational semantics rule for map.

Answer:

$$\frac{\begin{array}{c} so, S_1, E \vdash e_2 \mapsto [v_1, \dots, v_n], S_2 \\ l = newloc(S_2) \\ E' = E[l/id] \\ so, S_{i+1}[v_i/l], E' \vdash e_2 \mapsto v_i', S_{i+2} \text{ for } 1 \leq i \leq n \end{array}}{so, S_1, E \vdash \text{map}(\text{ID} : \text{TYPE} \rightarrow e_1, e_2) \mapsto [v_1', \dots, v_n'], S_{n+2}}$$

Another possible answer: Note that v_i is technically a value and not an expression and therefore one cannot initialize a let binding with v_i. However, we accept this answer anyway.

$$\frac{\begin{array}{c} so, S_1, E \vdash e_2 \mapsto [v_1, \dots, v_n], S_2 \\ so, S_{i+1}, E \vdash \text{let ID} : \text{TYPE} \leftarrow v_i \text{ in } e_1 \mapsto v_i', S_{i+2} \text{ for } 1 \leq i \leq n \end{array}}{so, S_1, E \vdash \text{map}(\text{ID} : \text{TYPE} \rightarrow e_1, e_2) \mapsto [v_1', \dots, v_n'], S_{n+2}}$$

4. **Runtime Environments** (15 points)

(a) Recall that a Cool object's in-memory representation is always preceded a word of memory containing "-1". What is the purpose of this extra value in memory in a Cool implementation?

Answer: We accepted two solutions. One is that it's used to detect memory corruption errors at runtime. Another is that it is used to maintain temporary state for the garbage collector; we only accept this answer for full credit if an example is given.

Identifying this as the garbage collector tag is not worth any credit, nor is it true that the cool runtime uses this to determine the bounds of objects.

(b) Consider an extension to Cool that adds arrays but does not do any checking whether array access are in bounds. How might an attacker execute arbitrary code by overrunning an array allocated in the *heap*? (Assume that the stack is so far away from the heap that it cannot be reached by this overrun.)

Answer: By overrunning an array one can override the dispatch pointer of some object. The new dispatch pointer has to point to a new attacker-supplied dispatch table which contains the address in memory of code the attacker wishes to execute.

5. **Multiple Inheritance** (25 points)

Consider the following two class definitions:

```
class P1 {
    x: Int <- (new IO).in_int();
    f1(): Int { x + 10 };
    f2(): Int { x + 20 };
};
```

```
class P2 {
    y: Int <- (new IO).in_int();
    f3(a: Int): Int { y + a };
    f1(): Int { y + 10 };
};
```

In this problem, we ignore methods inherited from `Object`. The object layouts for the two classes are thus as follows:

P1 class tag
4
P1 dispatch ptr
x

P1 object

P1.f1
P1.f2

P1 dispatch tbl

P2 class tag
4
P2 dispatch ptr
y

P2 object

P2.f3
P2.f1

P2 dispatch tbl

(a) Complete the MIPS assembly for the method body of `P2.f3` below by filling in the missing values. One value is needed for each blank ____. Assume that:

- `self` is passed in `$a0`,
- the argument `a` is the only value passed on the stack,
- the stack grows towards lower addresses,
- the result of a method is returned in register `$a0`, and
- the callee pops method arguments off of the stack.

```
P2.f3:
    lw $a0 12($a0)
    jal Object.copy
    lw $t1 12($a0)
    lw $t2 4($sp)
    lw $t2 12($t2)
    add $t1 $t1 $t2
    sw  $t1 12($a0)
    addiu $sp $sp 4
    jr $ra
```

We now introduce *multiple inheritance* to Cool. Consider the definition for class `C`, which inherits both `P1` and `P2`:

```
class C inherits P1, P2 {
    f1(): Int { 42 };
};
```

Here, class `C` overrides method `f1` while it inherits `f2` and `f3` from two different parents. The memory layout for class `C` is as follows:

`C` class tag
8
`C[P1]` dispatch ptr
x
`C` class tag
4
`C[P2]` dispatch ptr
y

`C` object

`C.f1`
`P1.f2`

`C[P1]` dispatch tbl

`P2.f3`
`C.f1`

`C[P2]` dispatch tbl

To guarantee backwards compatibility, the memory layouts and method implementations for `P1` and `P2` are unchanged.

For the next parts, assume that `c` is an identifier with static type `C`. The garbage collector is not run unless otherwise specified.

(b) Complete the MIPS assembly for the dispatch `c.f3(10)`. Assume that:

- `c` is not void,
- the address of object `c` is stored in `$s1`,
- an `Int` object with value 10 is located at label `int_const10`,
- your dispatch must work correctly with your code for `P2.f3` from part (a), and
- the stack grows towards lower addresses.

```
    la $t1 int_const10
    sw $t1 0($sp)
    addiu $sp $sp -4
    addiu $a0 $s1 16
    lw $t1 8($a0)
    lw $t1 0($t1)
    jalr $t1
```

(c) Now consider this method definition in class `Main`:

```
foo(b: P2): Int { b.f1() };
```

Consider the dispatch `foo(c)` within class `Main`. Note that `c` is implicitly cast from type `C` to type `P2` by the call. Describe what needs to be done on the caller's side of the calling sequence to ensure the method call executes correctly for any possible value of `c`, including void. You only need to describe any differences with regular Cool (without multiple inheritance); there is no need to write out the assembly code for the calling sequence.

Answer:

- Check if `c` is `void`. If so, pass `void` to `foo`.
- Check the class tag t of `c`.
- If t matches `P2`, then pass `c` unmodified.
- If t matches `C`, then pass `c+16`.
- If t matches a descendent of `C`, then search for the subobject of P2 and pass a pointer to it.

Note that because we said that `c` is of static type `C` it's not necessary to check if the class tag is `P2`, and we also accept solutions that don't handle the case of a subclass of `C`. As a result, it's not necessary to check the class tag to get full credit.

(d) With multiple inheritance, name one way the original Cool garbage collector might fail.

Answer: The garbage collector depends on several invariants that no longer hold. Describing the impact of any of them is satisfactory:

- Objects are preceded by the garbage collection tag -1. Subobjects, like the copy of `P2` contained in `C`, do not have their own garbage collection tag.
- After the class tag, every object contains the size of the entire object that needs to be collected. However, subobjects do not satisfy this.
- Given a pointer to a reachable subobject `P2`, a collector might not follow the pointers to reachable objects in the attributes of the containing object of type `C`.

6. **Semantic Analysis** (10 points)

In this problem, we explore extending Cool with constant values. We implement support for constant values by extending the type system. For every type T in the language, there is a new type **constant** T. We extend the grammar of Cool with the **constant** keyword, which can be applied to any type that appears in a program, except a class declaration.

A value v of type **constant** T can be initialized once in an initialization expression but never *altered*. That means the attributes of v, which are pointers, cannot be assigned after v is initialized. Furthermore, the objects to which attributes of v point cannot be altered. Thus this is a recursive definition. In the case of booleans, integers and strings, their primitive value must not change (although this is already the case in Cool without any extension).

(a) Consider the subtyping properties of **constant** types. Give a short Cool fragment showing that allowing **constant** $T \leq T$ is unsound.

Answer:

```
let x : constant A <- new constant A,
    y : A <- x in
  y.mutate()
```

(b) Would allowing $T \leq$ **constant** T be sound? Explain.

Answer: No. Consider the following program fragment,

```
let x : A <- new A,
    y : constant A <- x in
  x.mutate()
```

In this case, an attribute of the object `y` points to is modified after assignment.

Note that the meaning of constant values here is different than in languages like C++. In C++ a const pointer x only means that x cannot be used to update the object it points to, but nothing more. Here we provide a much stronger definition where a constant value is one that can never be modified, even by other pointers. Thus in C++ it is legal to assign a non-const pointer to a const pointer.