CIS 341 Final Examination
4 May 2017

SOLUTIONS

1. **Typechecking** (14 points total)

In this problem we will consider the typechecking rules for a simple OCaml-like programming language that has built-in support for *sum types*. To aid your intuition, we could implement sum types in OCaml using the following type declaration:

```
type ('a, 'b) sum = Left of 'a | Right of 'b
```

Our new language will use the notation $\tau_1+\tau_2$ as syntax for a built-in type that acts like `(`τ_1`,`τ_2`) sum` does in OCaml. We create values of type $\tau_1 + \tau_2$ using the `Left` and `Right` constructors, and we inspect such values using `match`, whose semantics is the same as in OCaml.

The grammar and (most of the) typechecking rules for this language is given in Appendix A.

a. (4 points) Which of the following terms are well-typed according to the rules shown in the Appendix? (Mark *all* correct answers.)

- ☐ `let x = 3 in (let x = 4 in x)`
- ☒ `let x = (let x = 4 in x) in x`
- ☐ `let x = (let y = 4 in x) in y`
- ☒ `let x = 3 in Left x`

b. (4 points) Recall that in our course compiler, we implemented the typechecker by viewing the judgment $\Gamma \vdash e : \tau$ as an OCaml function that takes in (OCaml representations of) Γ and e as inputs and produces (the OCaml representation of) τ as the output. That same strategy can no longer be used for this toy language, even with just the rules shown in the appendix. Briefly explain why not.

Answer: The rules for left and right would have to *guess* an appropriate other half of the sum type, but that isn't uniquely determined, so there is no function. For example, there is not a unique type that can be assigned to the term `Left` 3.

c. (6 points) Complete the blanks in the rule below to create a sound (and correct) inference rule for the `match` expression. Assume that the operational behavior of `match` is the same as in OCaml. Some of the blanks might need to be left empty. It is OK if variables bound in the match branches shadow occurrences from an outer scope.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x_1 : \tau_1, \Gamma \vdash e_1 : \tau \qquad x_2 : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{match}\ e\ \texttt{with Left}\ x_1 \rightarrow e_1\ \texttt{|}\ \texttt{Right}\ x_2 \rightarrow e_2 : \tau}\ [match]$$

2. Subtyping (15 points total)

For the next few problems, suppose that we are working with a programming language with subtyping and that there are types `A`, `B`, and `C` such that `B <: A` and `C <: A`. Further suppose that for all types `t`, `t <: Object`.

Mark *all* correct answers. There may be zero, or more than one.

a. Which of the following are sound *subtypes* of the function type `A` → `B`?

- ☐ `Object` → `Object`
- ☐ `A` → `C`
- ☒ `Object` → `B`
- ☐ `C` → `A`

b. Which of the following are sound *supertypes* of the mutable array type `B array`?

- ☐ `A array`
- ☒ `B array`
- ☐ `C array`
- ☒ `Object`

c. Suppose the programming language has conditional expressions and first-class functions (as in OCaml) in addition to subtyping. What is the most precise type that can be given to the following expression, assuming that `b` is a boolean and `new B()` creates a fresh object of type `B`?

```
if b then (fun (a:A) -> new  B()) else (fun (c:C) -> c)
```

- ☐ `A` → `Object`
- ☐ `A` → `A`
- ☒ `C` → `A`
- ☐ `C` → `Object`

d. Suppose that we create a language (like Swift) that makes a type distinction between types `A`, whose values are *definitely not null* references to `A` objects, and *possibly null* types `A?`, whose values are either references to `A` objects or `null`. Which direction should the sound subtyping relation go?

- ☒ `A <: A?`
- ☐ `A? <: A`
- ☐ there can be no sound subtyping relation between `A` and `A?`

3. **Object Compilation** (12 points)

Appendix B contains the LLVM code for the type definitions generated by compiling the source program written in a hypothetical object-oriented language that structures the class hierarchy using *single inheritance*.

a. Fill in each blank to correctly describe the source program that compiles to the given LLVM code.

- `A`'s parent in the class hierarchy is `Object`
- `B`'s parent in the class hierarchy is `Object`
- `C`'s parent in the class hierarchy is `A`

b. Check one box in each case below to make the sentence true of the source program that compiles to the given LLVM code.

- Class `B` ☒ *introduces* ☐ *overrides* ☐ *inherits* ☐ *doesn't have* a method called `print`.
- Class `B` ☒ *introduces* ☐ *overrides* ☐ *inherits* ☐ *doesn't have* a method called `foo`.
- Class `B` ☐ *introduces* ☐ *overrides* ☐ *inherits* ☒ *doesn't have* a method called `bar`.
- Class `C` ☐ *introduces* ☐ *overrides* ☒ *inherits* ☐ *doesn't have* a method called `print`.
- Class `C` ☐ *introduces* ☒ *overrides* ☐ *inherits* ☐ *doesn't have* a method called `foo`.
- Class `C` ☒ *introduces* ☐ *overrides* ☐ *inherits* ☐ *doesn't have* a method called `bar`.

c. How many fields does an object of (source) type `C` have? 2

d. Let `c` be an object of (source) type `C` and consider the legal method call `c.bar(c)`. True or False: compiling this call requires a `bitcast` to implement at the LLVM level.

☒ True

☐ False

4. Optimization (12 points)

Consider the following function written in Oat notation.

```
int foo(int n, int[] a, int[] b) {
  var sum = 0;
  for(var i = 0; i < n; i = i + 1; ) {
    a[i] = a[i] + b[2];
    sum = sum + a[i];
  }
  return sum;
}
```

Now consider this *incorrectly* "optimized" version, which tries to hoist the expression `b[2]` outside of the loop:

```
int foo_opt(int n, int[] a, int[] b) {
  var sum = 0;
  var tmp = b[2];
  for(var i = 0; i < n; i = i + 1; ) {
    a[i] = a[i] + tmp;
    sum = sum + a[i];
  }
  return sum;
}
```

There are at least *two* different reasons why the proposed "optimization" is incorrect, one having to do with aliasing, and one that does not involve aliasing.

a. (2 points) First complete the code below to give a set of inputs that *use* aliasing to demonstrate that the calls `foo(n,a,b)` and `foo_opt(n,a,b)` have different behaviors:

```
var a = new int[] {0, 1, 2, 3};
var n = 4;
var b = a;
```

b. (4 points) Now complete the code snippet to give a set of inputs that does not *use* aliasing, yet still demonstrates that the calls `foo(n,a,b)` and `foo_opt(n,a,b)` have different behaviors:

```
var a = new int[] {0, 1, 2, 3};
var n = 0;
var b = new int[] {};
```

c. (True or False) Inlining function definitions always improves running time performance.

☐ True [✓ ☒ False]

d. (True or False) Removing dead code cannot impact the running time of a program.

☐ True [✓ ☒ False]

e. We have seen that a source C (or Oat) variable declaration like the one shown on the left below will typically first be translated to the LLVM code shown on the right, which uses `alloca`:

```
int x = 0;                     %_var_x = alloca i64
                               store i64 0, i64*  %_var_x
                               /* here */
```

Which of the following LLVM instructions, if it appeared at the point marked `/* here */` in the LLVM code, would *prevent* alloca promotion of `%_var_x` (i.e. replacing its uses with uids; also known as SROA optimization)? There may be zero or more than one answer.

☐ `%y = load i64* %_var_x`

[✓ ☒ `store i64* %_var_x, i64** %_var_y`]

[✓ ☒ `%y = call i64* foo(i64* %_var_x)`]

☐ `store i64 %_var_y, i64* %_var_x`

5. **Data-flow Analysis** (34 points)

Warning: read all of this problem before tackling the earlier parts—the later parts give you clues to the earlier ones. There are no questions to answer on this page!

In this problem, we will consider a dataflow analysis for *anticipated expressions*, which finds useful facts for performing an optimization called *partial redundancy elimination*. For the purposes of this problem, we will consider a stripped-down version of the LLVM IR that contains only the integer types `i1` and `i64`, pointer types `T*`, and the following instructions / terminators and phi functions:

```
 Instructions:  %x = add i64 op1, op2
                %x = icmp eq i64 op1, op2
                %x = load T* op
                %x = alloca T
                %x = store T op1, T* op2      (%x is a dummy id)
 Terminators:   ret void
                br label %lbl
                br i1 op1, label %lbl1, label %lbl2
Phi functions:  %x = phi T [op1, %lbl1] [op2, %lbl2]
```

We structure these into a control-flow graph as usual.

The *expression* of an instruction of the form `%x = e` is just `e`, though for the purposes of this analysis we treat `alloca` and `store` as defining no expressions, since their behavior operates by side effect (see part (c)).

Definition 0.1 (Anticipated Expression) *We say that an expression* `e` *is* anticipated *at a program point p if* all *paths leading from* `p` *eventually compute* `e` *and* `e` *has the same value at those points as it does at p.*

Intuitively, this analysis is used to find expressions that can potentially be moved earlier in the code. If `e` is computed multiple times, moving its definition earlier can allow the resulting value to be shared among those redundant occurrences—see parts (d) and (e) of this problem for an example.

Computing anticipated expressions is a *backward* dataflow analysis. Recall the generic frameworks for backward iterative dataflow analysis that we discussed in class. Here, `n` ranges over the nodes of the control-flow graph, `succ[n]` is the set of successor nodes, F_n is the *flow function* for the node `n`, and $\sqcap$ is the *meet* combining operator.

```
for all nodes n: in[n] = $\top$, out[n] = $\top$;
repeat until no change {
  for all nodes n:
    out[n]  := $\bigsqcap_{n' \in succ[n]}$ in[n'];
    in[n] := $F_n$(out[n]);
}
```

Recall that the flow functions `Fn` and the $\sqcap$ operator work over elements of a *lattice* $\mathcal{L}$. The anticipated expressions analysis will compute, for each program point p of the control-flow graph, a *set* of expressions that are anticipated at that point, so each $\ell \in \mathcal{L}$ is a set of expressions.

a. (3 points) What should the meet operation $\sqcap$ be for this analysis and (briefly) why?

*Answer:*Intersection: because an expression is anticipated only if it computed on *all* paths from p.

b. (3 points) Why is this dataflow lattice of finite height?

*Answer:*Each set consists of a subset of expressions found in the CFG, which is finite, so the height is at most E where E is the number of distinct expressions appearing in the graph.

c. We can now define the flow functions for each program instruction $\mathtt{n}$ by defining:

$$\mathtt{F_n}(\ell) \quad = \quad (\ell \cup \{\mathtt{e}\}) - \mathtt{kill}(\mathtt{n}) \qquad \text{where } \mathtt{n} \text{ is of the form } \mathtt{\%x\ =\ e}$$

$$\mathtt{F_{(ret\ void)}}(\ell) \quad = \quad \emptyset$$

To define $\mathtt{kill}(\mathtt{n})$, the set of expressions killed by the instruction $\mathtt{n}$, it is useful to have a helper function that computes the set of all expressions that use the identifier $\mathtt{\%x}$:

$$\mathtt{usesOf}(\mathtt{\%x}) = \{\mathtt{e} \mid \mathtt{\%x} \text{ is an operand of } \mathtt{e}\}$$

Once `usesOf` is defined, we specify `kill` as follows:

n	$\mathtt{kill}(\mathtt{n}) =$
`%x = alloca T`	$\mathtt{usesOf}(\mathtt{\%x}) \cup \{\mathtt{alloca\ T}\}$
`%x = store T op1, T* op2`	$\mathtt{usesOf}(\mathtt{\%x}) \cup \left(\bigcup_{\mathtt{\%y} \in \mathtt{mayAlias(op2)}} \mathtt{usesOf}(\mathtt{\%y})\right)$
`%x = e`	$\mathtt{usesOf}(\mathtt{\%x})$

i. (3 points) Which of the following properties do the flow functions satisfy? (Mark all that apply.)

☒ Monotonicity: $\ell_1 \sqsubseteq \ell_2 \Rightarrow \mathtt{F_n}(\ell_1) \sqsubseteq \mathtt{F_n}(\ell_2)$

☒ Distributivity: $\mathtt{F_n}(\ell_1 \sqcap \ell_2) = \mathtt{F_n}(\ell_1) \sqcap \mathtt{F_n}(\ell_2)$

☒ Consistency: $\mathtt{F_n}(\ell) \sqsubseteq \top$

ii. (4 points) The definition of the set $\mathtt{kill}(\mathtt{n})$ depends on an *alias analysis*. One conservative, but sound alias analysis would be to say that *any* uid of pointer type may alias *any* other such uid. Briefly describe one way to improve the alias information for this LLVM subset.
Answer: Use the fact that two distinct occurrences of `alloca` can never alias.

d. (9 points) Consider applying the anticipated expressions analysis to the control-flow graph below.

```
void foo( i1 %c, i64 %a, i64 %b)  {
entry:
  %t1 = alloca i64
  %d1 = store i64 0, i64* %t1
  br i1 %c, label %then, label %else
then:
  %x1 = add i64 %a, %b
  %d2 = store i64 %x1, %t1
  br label %merge
else:
  br label %merge
merge:
  %x2 = add i64 %a, %b
  %x3 = load i64* %t1
  %x4 = add i64 %x2, %x3
  %d3 = store i64 %x4, %t1
  ret void
}
```

i. What is the lattice element computed by the dataflow analysis for `in[else]`, i.e. the lattice element at the incoming edge of the `else` block? (select one)

- ☐ ∅
- ☐ { `add i64 %a, %b` }
- [✓ { `add i64 %a, %b,  load i64* %t1` }]
- ☐ { `add i64 %a, %b,  load i64* %t1,  add i64 %x2, %x3` }

ii. What is the lattice element computed by the dataflow analysis for `in[then]`, i.e. the lattice element at the incoming edge of the `then` block? (select one)

- ☐ ∅
- [✓ { `add i64 %a, %b` }]
- ☐ { `add i64 %a, %b,  load i64* %t1` }
- ☐ { `add i64 %a, %b,  load i64* %t1,  add i64 %x2, %x3` }

iii. What is the lattice element computed by the dataflow analysis for `in[entry]`, i.e. the lattice element at the incoming edge of the `entry` block? (select one)

- ☐ ∅
- [✓ { `add i64 %a, %b` }]
- ☐ { `add i64 %a, %b,  load i64* %t1` }
- ☐ { `add i64 %a, %b,  load i64* %t1,  alloca i64` }

e. An expression e is *redundant* if it is computed multiple times along one path of program execution (and each time it is computed it yields the same value). A redundant expression e can be moved anywhere it is anticipated, which may allow rewriting the program to eliminate the redundant computations. For example, in the program from part (d) the expression `add i64 %a, %b` is redundantly computed when the "`then`" branch is taken.

- **i.** (4 points) Redundant expressions are typically moved as *late* as possible while still reducing (or keeping the same) the number of times the expression is computed along each path. Briefly explain why they are moved "as *late* as possible" and *not* "as *early* as possible". (Hint: think about the impact on code generation.)
  Answer: Moving the expression as early as possible increases the live range of its operands and thus causes more interference for subsequent register allocation.
- **ii.** (8 points) Fill in the blanks below to eliminate the redundant expression `add i64 %a, %b` from `foo`. Follow the "as late as possible" policy described above . *You may not need to add code at all of the blanks.*

```
void foo( i1 %c, i64 %a, i64 %b)  {
entry:
  %t1 = alloca i64
  %d1 = store i64 0, i64* %t1
  ____________(blank)____________
  br i1 %c, label %then, label %else
then:
  ____________%x1 = add i64 %a, %b____________
  %d2 = store i64 %x1, %t1
  br label %merge
else:
  ____________%x0 = add i64 %a, %b____________
  br label %merge
merge:
  ____________%x2 = phi i64 [%x1, %then] [%x0, %else]____________
  %x3 = load i64* %t1
  %x4 = add i64 %x2, %x3
  %d3 = store i64 %x4, %t1
  ret void
}
```

6. Control-flow Analysis (21 points)

Consider the following control-flow graph $\mathbb{G}$ with nodes labeled A through G and edges labeled 1 through 11. Node A is the entry point.

$$\begin{tikzcd}
& {} \arrow[d] & \\
& A \arrow[dl, "1"'] \arrow[dr, "2"] & \\
B \arrow[d, "3"] \arrow[dd, bend right=60, "6"'] & & C \arrow[d, bend right, "4"'] \\
D \arrow[d, bend right, "7"'] \arrow[uur, "11"'] & & E \arrow[u, bend right, "5"'] \arrow[ddl, "9"] \\
F \arrow[u, bend right, "8"'] \arrow[dr, "10"] & & \\
& G &
\end{tikzcd}$$

a. (6 points) Draw the dominance tree for the control-flow graph $\mathbb{G}$, making sure to label nodes appropriately (there is no need to label the edges).

$$\begin{tikzcd}
& & {} \arrow[d] & \\
& & A \arrow[dl] \arrow[d] \arrow[dr] & \\
& B \arrow[dl] \arrow[d] & G & C \arrow[d] \\
D & F & & E
\end{tikzcd}$$

b. (6 points) For each *back edge* e of $\mathbb{G}$, identify the set of nodes appearing e's *natural loop*. Each answer should be of the form "e, $\{nodes\}$" where e is a back edge and $nodes$ is the set of nodes that make up the loop.

5, {C, E}
11, {A, B, D, F}

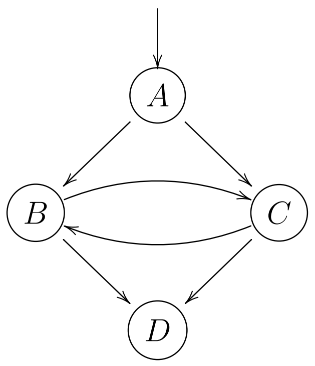
c. (3 points) Which nodes are in the dominance frontier of the node `D`?

{A,F}

d. (3 points) Which nodes are in the dominance frontier of the node `F`?

{D, G}

e. (3 points) The following control-flow graph is *irreducible*: it contains a cycle but has no back edges and hence no natural loops.



Can such a graph arise at the LL level when compiling Oat programs as found in HW5? If "yes", give an example, if "no" briefly explain why not.

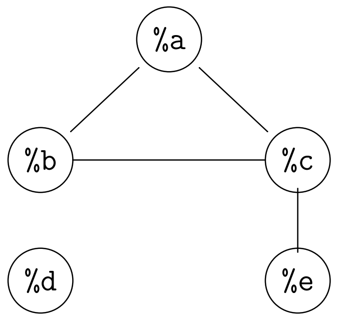
Answer: No: Oat cannot generate this cfg because its control-flow constructs are well-nested. It does not have `goto` or any other means of transferring control from one branch of a conditional to another.

7. Register Allocation (10 points)

a. (5 points) Consider the following straight-line LLVM code.

```
%c = call i64 init()
%a = add i64 %c, %c
%b = add i64 %c, %c
%e = add i64 %a, %b
%d = add i64 %c, %e
ret i64 %d
```

Label the vertices of the following *interference graph* (where the solid lines denote interference edges) so that it corresponds to the code above. Each label %a–%e should be used once. (There may be more than one correct solution.)



b. (2 points) What is the minimal number of colors needed to color the graph?

3

c. (3 points) Briefly explain the purpose of *coalescing* nodes of the interference graph.

Answer: Coalescing forces two nodes to be given the same color (and hence the same register). It is useful when one temporary is moved into another because the resulting move instruction can then be eliminated.

CIS341 Final Exam 2017 Appendices

(Do not write answers in the appendices. They will not be graded)

APPENDIX A: Typechecking Sum Types

Grammar for a simple expression-based language with sum types:

$$\begin{array}{lcll}
\tau & ::= & & \text{Types} \\
& \mid & \texttt{int} & \\
& \mid & \tau_1 + \tau_2 & \\
\\
e & ::= & & \text{Expressions} \\
& \mid & x & \quad\text{variables} \\
& \mid & n & \quad\text{integer constants} \\
& \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 & \quad\text{local lets} \\
& \mid & \texttt{Left}\ e & \quad\text{sum constructors} \\
& \mid & \texttt{Right}\ e & \\
& \mid & \texttt{match}\ e\ \texttt{with}\ \texttt{Left}\ x_1 \rightarrow e_1\ \texttt{|}\ \texttt{Right}\ x_2 \rightarrow e_2 & \quad\text{case analysis} \\
& \mid & \texttt{(}e\texttt{)} & \\
\\
\Gamma & ::= & & \text{Typechecking Contexts} \\
& \mid & \cdot & \\
& \mid & x{:}\tau, \Gamma &
\end{array}$$

Typechecking expressions is defined, in part, by the following inference rules. Recall that the notation $x : \tau \in \Gamma$ means that x occurs in the context Γ at type τ and $x \notin \Gamma$ means that x is not bound in Γ.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}\ [\mathit{var}] \qquad\qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ [\mathit{int}]$$

$$\frac{x \notin \Gamma \quad \Gamma \vdash e_1 : \tau_1 \quad x{:}\tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ [\mathit{let}]$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{Left}\ e : \tau_1 + \tau_2}\ [\mathit{left}] \qquad\qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{Right}\ e : \tau_1 + \tau_2}\ [\mathit{right}]$$

APPENDIX B: Example LLVM Code for an OO Program

The following LLVM code was obtained by compiling a source program that uses single-inheritance (as in Java, but without interfaces). We have omitted the definitions of the method bodies (i.e., we omit the definitions of `@_foo_A`, `@_print_A`, etc.).

```
%Object = type { %_class_Object* }
%_class_Object = type {  }

%A = type { %_class_A*, i64 }
%_class_A = type { %_class_Object*, void (%A*)*, i64 (%A*, %C*)* }

%B = type { %_class_B*, i64, i64 }
%_class_B = type { %_class_Object*, void (%B*)*, i64 (%B*, %B*)* }

%C = type { %_class_C*, i64, i64 }
%_class_C = type { %_class_A*, void (%A*)*, i64 (%C*, %C*)*, %A* (%C*, %A*)* }

@_vtbl_Object = global %_class_Object {  }

@_vtbl_A = global %_class_A { %_class_Object* @_vtbl_Object,
                              void (%A*)* @_print_A,
                              i64 (%A*, %C*)* @_foo_A
                            }

@_vtbl_B = global %_class_B { %_class_Object* @_vtbl_Object,
                              void (%B*)* @_print_B,
                              i64 (%B*, %B*)* @_foo_B
                            }

@_vtbl_C = global %_class_C { %_class_A* @_vtbl_A,
                              void (%A*)* @_print_A,
                              i64 (%C*, %C*)* @_foo_C
                              %A* (%C*, %A*)* @_bar_C
                            }
```