CIS 341 Final Examination
4 May 2018

SOLUTIONS

1. Inference Rules, Types, and Subtyping (21 points total)

The following questions refer the inference rules given in Appendix A.

a. (3 points) According to the rules, which of the following types are *subtypes* of `bool` $\to$ `bool`?

☒ $\bot$
☒ `bool` $\to$ `bool`
☐ $\bot \to$ `bool`
☒ `bool` $\to \bot$
☐ $\bot \to \bot$

b. (3 points) According to the rules, which of the following types are *supertypes* of `bool` $\to$ `bool`?

☐ $\bot$
☒ `bool` $\to$ `bool`
☒ $\bot \to$ `bool`
☐ `bool` $\to \bot$
☐ $\bot \to \bot$

c. (3 points) The term $\lambda(x : \bot).\ \lambda(x : \texttt{bool}).\ x$ is *not* well typed in the empty typing context. In which inference rule would type checking fail? (Choose one)

☐ $[\mathit{var}]$ ☐ $[\mathit{true}]$ ☐ $[\mathit{false}]$ ☒ $[\mathit{lam}]$ ☐ $[\mathit{app}]$

d. (3 points) The term $\lambda(x : \texttt{bool} \to \texttt{bool}).\ (z\ x)$ is *not* well typed in the empty typing context. In which inference rule would type checking fail? (Choose one)

☒ $[\mathit{var}]$ ☐ $[\mathit{true}]$ ☐ $[\mathit{false}]$ ☐ $[\mathit{lam}]$ ☐ $[\mathit{app}]$

e. (9 points) The term $\lambda(x : \bot).\ (x\ \ x)$ *is* well typed. Demonstrate that fact by completing the following derivation tree. Label the use of each instance of an inference rule with its name (in brackets), and fill in the missing types in the remaining boxes.

$$\dfrac{\dfrac{\dfrac{x{:}\bot \in x{:}\bot}{x{:}\bot \vdash x : \bot}\,[\mathit{var}] \qquad \dfrac{x{:}\bot \in x{:}\bot}{x{:}\bot \vdash x : \bot}\,[\mathit{var}] \qquad \dfrac{}{\vdash \bot <: \bot \to \bot}\,[\mathit{bot}]}{x{:}\bot \vdash x\ x : \bot}\,[\mathit{app}]}{\cdot \vdash \lambda(x{:}\bot).\,(x\ x) : \bot \to \bot}\,[\mathit{lam}]$$

2. Compilation (25 points total)

Some languages (like C) have `break` and `continue` statements. `break` causes control to jump out of the most closely syntactically enclosing loop, whereas `continue` jumps immediately to the start of the most closely enclosing loop. Both constructs *must* only be used inside some loop. In this question we will examine how to implement `break` and `continue` in OAT (only for `while` loops, we'll ignore `for` loops for simplicity).

The following OAT code demonstrates both `break` and `continue`. This program will exit the loop when `x` reaches the value 4 and it will never print anything:

```
int foo() {
  var x = 1;
  while (true) {
    if (x > 3) {
      break;
    } else {
      x = x + 1;
      continue;
      print_string("Will never print");
    }
    print_string("Will also never print");
  }
  return x;
}
```

a. (4 points) Appendix B contains the LL code that we would expect to obtain by compiling the OAT program listed above using a very simple modification to the exiting OAT compiler that we explore next. The unconditional jump (`br`) instructions that correspond to the uses of `break` and `continue` have been left out, but they are marked with `LOCATION` flags. To which label should each branch to?

`LOCATION 1` should branch to label __________`_post3`__________

`LOCATION 2` should branch to label __________`_cond5`__________

b. (12 points) Appendix C contains an excerpt of the existing parts of the OAT frontend that will need to be modified to implement `break` and `continue`. We also extend the OAT abstract syntax as shown to include the new statement forms. One first cut at implementing `break` and `continue` is to make just a few minor changes to the existing compiler, which will generate the code from Appendix B. (We will explore improvements on the next page.)

Fill in the blanks below so that the resulting compiler supports `break` and `continue`. We have allowed for you to add additional arguments to either or both of `cmp_stmt` and `cmp_block` (and we'll assume that calls to these functions that aren't shown below have been adjusted accordingly):

```
let rec cmp_stmt (tc : TypeCtxt.t) (c:Ctxt.t) (rt:Ll.ty) (stmt:Ast.stmt node)

                ____________(lexit:Ll.lbl option) (lcond:Ll.lbl option)____________
                : Ctxt.t * stream =
  match stmt.elt with

  | Ast.Break -> ____________(match lexit with | None -> failwith "err"____________

                 _________________________| Some l -> T (Br l))_________________________

  | Ast.Continue -> ____________(match lcond with | None -> failwith "err"____________

                    _________________________| Some l -> T (Br l))_________________________

  | Ast.While (guard, body) ->
     let guard_ty, guard_op, guard_code = cmp_exp tc c guard in
     let lcond, lbody, lpost = gensym "cond", gensym "body", gensym "post" in

     let body_code = cmp_block tc c rt body _______(Some lpost) (Some lcond)_______ in
     c, []
        >:: T (Br lcond)
        >:: L lcond >@ guard_code >:: T (Cbr (guard_op, lbody, lpost))
        >:: L lbody >@ body_code >:: T (Br lcond)
        >:: L lpost

  | ...  (* other cases omitted *)

and cmp_block (tc : TypeCtxt.t) (c:Ctxt.t) (rt:Ll.ty) (stmts:Ast.block)

                ____________(lexit:Ll.lbl option) (lcond:Ll.lbl option)____________
  : stream =
  snd @@ List.fold_left (fun (c, code) s ->
      let c, stmt_code = cmp_stmt tc c rt s _______________lexit lcond_______________ in
      c, code >@ stmt_code
    ) (c,[]) stmts
```

If you modified the function arguments to `cmp_body`, specify how it would be invoked "at the top level" (i.e. when compiling a function body, which is by definition not in any while loop):

```
  ...
  let block_code = cmp_block tc c ll_rty body _______________None None_______________ in
  ...
```

c. There are two related issues with adding `break` and `continue` as sketched above. One issue is that the generated instruction stream may not be well-formed: it can contain two block terminators in a row, as illustrated by lines 12 and 13 of the code in Appendix B and labels can be interspersed with other code without proper block label entry point. Another issue is that both of these new statements might create a lot of unreachable code, as illustrated by lines 19 – 21.

i. (3 points) After filling in the missing `LOCATION` instructions to implement `break` and `continue`, there will more dead code in the Appendix B example besides that mentioned above. Where is it? (Give your answer as a range of line numbers x–y:)

22–25

ii. (6 points) Briefly (!) describe how you would modify the OAT frontend to deal with these issues. *Possibile parts of a good answer include the following points:*

- Change the body of `cmp_block` to not use `List.fold_left` and to instead stop iterating when it encounters a `break` or `continue`. This will eliminate a lot of the dead code, but not the "double label" or the "dead merge block" problem.
- Change the stream to blocks algorithm to drop a partially defined block. That is, when converting the stream to a CFG, if a terminator is encountered before one is expected, rather than raising an exception, instead it can simply drop the block being generated and start a new block. This will handle the "double label" and much of the dead code, but not the "dead merge block" problem.
- Run a reachability dataflow analysis to identify blocks whose labels are never used and drop those blocks from the CFG. This will deal with the "dead merge block" problem, but not the other two issues.

3. Data-flow Analysis (30 points total)

A major convenience in modern programming languages is automatic memory management. In languages like OCaml, tuples and records are just values that appear in expressions, and it is up to the compiler to allocate these objects in memory appropriately. In languages like Java and OAT, the programmer has to be more explicit in allocating objects through the `new` keyword, but does not have to think about *freeing* or deallocating them.

An object allocated on the heap is a set of locations in the heap that holds the object's contents. To free an object is to mark its locations as being available, so that subsequent allocations (through `new`) can use them.

The OAT compiler currently compiles the allocation of OAT objects, i.e. involving the `new` keyword, to an external function call to `oat_malloc` that handles the actual allocation on the heap. But objects are actually never freed, so long-running OAT programs that create new objects will eventually crash!

A full solution involves linking a runtime garbage collector to compiled programs. But one way to partially alleviate this problem is for the compiler to distinguish between objects that are "freeable" versus those that "may escape" when the function returns. Intuitively, an object within a function does not escape (i.e. is freeable) if no part of it can be accessed from any other function, whether directly through a pointer or indirectly by following other pointers, once the function returns. Freeable objects can be deallocated before the function exits.

More specifically, we say that a pointer `%ptr` (to an allocated object) is freeable at the exit of a function `@f` if:

- No part of the object is accessible by following pointers from any arguments.
- No part of it is accessible from any returned value.
- No pointer to it (or a subcomponent) is ever assigned to a global.
- No pointer to it (or a subcomponent) is ever passed to another function via a `call`.

These conditions are sufficient to ensure that no other function can access any part of the object, so it is safe to free it upon function exit. In this problem, we will define a dataflow analysis that identifies freeable pointers in an LL program.

We will define the analysis in two steps. First, we define an "accessibility" relation, which approximates the set of pointers that are accessible from a given starting pointer. Then we will use the notion of accessibility to define freeability, following the rules above.

Note: For simplicity, assume for the rest of this question pertains to a variant of the LL language that **does not** support global definitions.

(There are no questions on this page.)

a. (5 points) First, we look at some example programs to determine what the final behavior of the analysis should be. For each of the following programs, check the box to indicate whether `%ptr1` is *freeable* (according to the explanation above).

i.
```
define void @ptr_example1(i64 %arg) {
  %ptr1 = call i64* @oat_malloc(i64 1)
  store i64 %arg, i64* %ptr1
  ret void
}
```
☒ `%ptr1` is freeable
☐ `%ptr1` is *not* freeable

ii.
```
define void @ptr_example2(i64** %arg) {
  %ptr1 = call i64* @oat_malloc(i64 1)
  store i64* %ptr1, i64** %arg
  ret void
}
```
☐ `%ptr1` is freeable
☒ `%ptr1` is *not* freeable

iii.
```
define i64* @ptr_example3() {
  %ptr1 = call i64* @oat_malloc(i64 1)
  ret i64* %ptr1
}
```
☐ `%ptr1` is freeable
☒ `%ptr1` is *not* freeable

iv.
```
define i64* @ptr_example4() {
  %ptr1 = call i64* @oat_malloc(i64 1)
  %x = alloca i64*
  store i64* %ptr1, i64** %x
  %copy = load i64*, i64** %x
  ret i64* %copy
}
```
☐ `%ptr1` is freeable
☒ `%ptr1` is *not* freeable

v.
```
define i64* @ptr_example5(i64 %arg) {
  %ptr1 = call i64* @oat_malloc(i64 1)
  store i64 %arg, i64* %ptr1
  %ptr2 = call i64* @oat_malloc(i64 1)
  %tmp = load i64, i64* %ptr1
  store i64 %tmp, i64* %ptr2
  ret i64* %ptr2
}
```
☒ `%ptr1` is freeable
☐ `%ptr1` is *not* freeable

Next, we tackle the first part of the analysis—the accessibility relation. Intuitively, we are trying to (conservatively) identify when it is possible to reach the memory location pointed to by a uid `%u` by following pointers starting from `%v`. Let us write `%v` $\rightsquigarrow$ `%u`, when `%u` is accessible from `%v` in this sense. (For your reference, Appendix E includes an example LLVM program plus its accessibility information.)

We can compute accessibility using an instantiation of the iterative dataflow analysis framework that we used for HW6. The *facts* computed by our accessibility analysis are maps m that associate each uid `%v` of the function with a *set* of uids that might be reachable (at that point in the control-flow graph) by traversing pointers starting from `%v`. Because every uid is reachable from itself, we'll assume that our implementation of maps ensures that $\%v \in m(\%v)$ for every map m.

b. (4 points) The goal of the analysis is to compute a map, m_A that *soundly approximates* the accessibility relation. In this context, what does "soundly approximates" mean? Choose one and briefly explain why:

[✓] $m_A(\%v) \supseteq \{\%u \mid \%v \rightsquigarrow \%u\}$

☐ $m_A(\%v) \subseteq \{\%u \mid \%v \rightsquigarrow \%u\}$

Why?

Answer: We plan to use the accessibility information to approximate when it is safe to free a pointer. Therefore, to be conservative, we want to permit more things to be considered accessible than might actually be possible. This means that we might not free some data that is not actually accessible, but we will never free data that is accessible.

Recall that we need to specify the *join* $\sqcup$ (or *combine*) operation and *flow functions* F_I for these facts to fully define the dataflow analysis.

c. (4 points) Which of the following should we use as the *join* function for two maps? Briefly explain why.

☐ $(m_1 \sqcup m_2)(\%v) = m_1(\%v) \cap m_2(\%v)$

[✓] $(m_1 \sqcup m_2)(\%v) = m_1(\%v) \cup m_2(\%v)$

Why?

Answer: This analysis is computing accessibility information along *any* path through the control flow graph, so we want to take the union of the possibilities—if an accessibility relation holds along one possible path, we want to include it in the results.

The flow function $F_I(m)$ depends on the instruction I. For most of the LL instructions, and all of the block terminators, the flow function is just the identity $F_I(m) = m$, because they don't create new aliasing of pointers. Setting aside `call` and `store` for now, the instructions I that manipulate pointers (and so might affect the accessibility relation) are:

- `%u = bitcast W* %w to T*`
- `%u = getelementptr U, U* %w, ...`
- `%u = load T*, T** %w`

All of these instructions introduce new accessibility relations because they make `%u` an alias for a pointer accessible from `%w`. They have the *same* flow function $F_I(m) = m'$, where m' is defined in terms of m and the uids `%u` and `%w` appearing in I as shown below:

$$m'(\%\mathtt{v}) = \begin{cases} m(\%\mathtt{v}) \cup m(\%\mathtt{u}) \cup m(\%\mathtt{w}) & \text{when } \%\mathtt{w} \in m(\%\mathtt{v}) \text{ or } \%\mathtt{u} \in m(\%\mathtt{v}) \\ m(\%\mathtt{v}) & \text{otherwise} \end{cases}$$

Intuitively, this transfer function says that if `%v` $\rightsquigarrow$ `%u` or `%v` $\rightsquigarrow$ `%w` before I, then `%v` $\rightsquigarrow$ `%x` for any `%x` reachable by either `%u` or `%w` after I.

d. (4 points) Fill in the blanks below to complete the transfer function $F_I(m) = m'$ for the case where $I =$ `store T* %u, %T** w`. Your solution should be *sound*, but as precise as possible.

$$m'(\%\mathtt{v}) = \begin{cases} m(\%\mathtt{v}) \cup \underline{\quad m(\%u) \quad} & \text{when} \underline{\quad \%\mathtt{w} \in m(\%\mathtt{v}) \quad} \\ m(\%\mathtt{v}) & \text{otherwise} \end{cases}$$

f. (4 points) Briefly explain what transfer function you would use for the instruction $I =$
`%u = call T* (U1 arg1, ..., Un argN)`

Answer: We want to say that, due to a function call, any of the pointers accessible by any of the arguments might potentially become accessible to one another and the output `%u`, so we conservatively merge all of the accessibility of all of those things together.

The mathematical answer is:

$$m'(\%\mathtt{v}) = \begin{cases} m(\%\mathtt{v}) \cup m(\%\mathtt{u}) \cup \left(\bigcup_{\mathtt{argj}} m(\%\mathtt{argj})\right) & \text{when } \%\mathtt{argi} \in m(\%\mathtt{v}) \text{ for any } \%\mathtt{argi} \text{ or } \%\mathtt{u} \in m(\%\mathtt{v}) \\ m(\%\mathtt{v}) & \text{otherwise} \end{cases}$$

Recall that this (forward) dataflow analysis associates facts with the outgoing CFG edges. After completing the accessibility analysis describe above, we can compute a final map m_A that summarizes all of the accessibility information of from the whole control-flow graph. Let $rets$ be the set of `ret` instructions in the CFG.

$$m_A = \bigsqcup_{I \in rets} out[I]$$

g. (4 points) Why is the dataflow analysis described above guaranteed to terminate?

Answer: There are only finitely many uid identifiers in a program, so the number of possible maps from uids to sets of uids is bounded (if there are N uids, then the number of possible lattices is something like $(2^N)^N$). At each stage of the analysis, the maps only get coarser (i.e. we always add some elements to one of the sets and we never remove them), so we can do that only a finite number of times before the algorithm will terminate.

Finally, we can complete the freeability analysis for the control flow graph for some function

```
define T @f(T1 %arg1, ... , TN %argN) { ... }
```

We first compute the accessibility map m_A for the CFG of `f`. We can then say that a pointer `%ptr` created by a call to `oat_malloc` is freeable if, for all `%w` $\in m_A$(`%ptr`), `not(escapes(%w))`. Here, the function `escapes(%w)`, which also uses m_A, does a pass through the CFG for `f` and returns `true` if `%w` occurs in any instruction I of the forms:

`%v = call T g(...W* %w...)` and similarly for `void calls` (*i.e.*, `%w` appears as a function argument)
`store W* w, W** u`, where %u $\in m_A$(%arg) for some %arg
`ret W* w`

Claim: if m_A is a sound approximation to accessibility, then this freeability analysis will identify a pointer as freeable only if it is safe to free.

(This problem continues on the next page.)

h. (5 points) As we observed, this is a *conservative* analysis—even if you have "perfect" accessibility information (so this question does not depend on how you answer parts **d** and **f**). Briefly describe (no need to write the code, though code is fine too) how you could extend the code for the following LL program such that `%ptr1` would *not* be identified as freeable, yet it would nevertheless be memory safe to call `free(%ptr1)` before returning from the function.

Answer: We are conservative about what happens when a function is called, so, even if the escaping pointer is never actually used (in which case it would be safe to free it), our analysis will say that it cannot be freed. The code below illustrates such a program, where `@foo` completely ignores its argument, so `%ptr1` could actually be freed at the end of `@conservative`.

```
define i64* @foo(i64* %ptr1) {
  %new = call i64* @oat_malloc(i64 1)
  ret i64* %new
}

define i64* @conservative(i64 %arg) {
  %ptr1 = call i64* @oat_malloc(i64 1)
  %ptr2 = call i64* @foo(i64* %ptr1)
  ret i64* %ptr2
}
```

4. Control-flow Analysis (16 points total)

The following questions concern the following control-flow graph, where the nodes numbered 1–6 represent *basic blocks* and the arrows denote control-flow edges.

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] & \\
2 \arrow[dr] & & 3 \arrow[dl] \\
 & 4 \arrow[ul, bend left] \arrow[d] & \\
 & 5 \arrow[uur, bend right] \arrow[d] & \\
 & 6 \arrow[u, bend right] &
\end{tikzcd}$$

a. (4 points) It is straightforward to create a LLVMLite program (i.e. a `.ll` file, as in our project compiler) whose basic blocks form the graph shown above. Which of the following are true statements about *all* such LL programs whose graphs have this shape? (Mark all true statements.)

- ☐ The program has one conditional branch (`cbr`) instruction.
- ☒ The program has no return (`ret`) instructions.
- ☐ The program has three `call` instructions.
- ☒ The program as a unique entry block.

b. (4 points) Draw the dominator tree for the control flow graph:
Solution:

$$\begin{tikzcd}
 & 1 \arrow[dl] \arrow[dr] \arrow[dd] & \\
2 & & 3 \\
 & 4 \arrow[d] & \\
 & 5 \arrow[d] & \\
 & 6 &
\end{tikzcd}$$

c. (4 points) Recall that a *natural loop* is a strongly-connected component with a unique entry node (the header) that is the target of a back edge. Which of the following sets of nodes are natural loops of this graph? For each set, fill in the blank with the number of its loop header, or write "NANL" for "not a natural loop".

$\{2, 4\}$	NANL
$\{3, 4, 5\}$	NANL
$\{5, 6\}$	5
$\{2, 3, 4, 5, 6\}$	NANL

d. (4 points) There is no source OAT program that, when compiled, results in an LLVMLite program whose control-flow graph has the depicted shape. Briefly explain why.

Answer: OAT programs have well-nested loops and conditionals and no "goto" or direct "jump" operations. This means that it is not possible to write a program that jumps into the "`then`" or "`else`" branch of a conditional from outside the conditional block itself. Therefore the back edges from $4 \to 2$ and $5 \to 3$ are not possible to express in a source OAT program.

5. Object Compilation (15 points total)

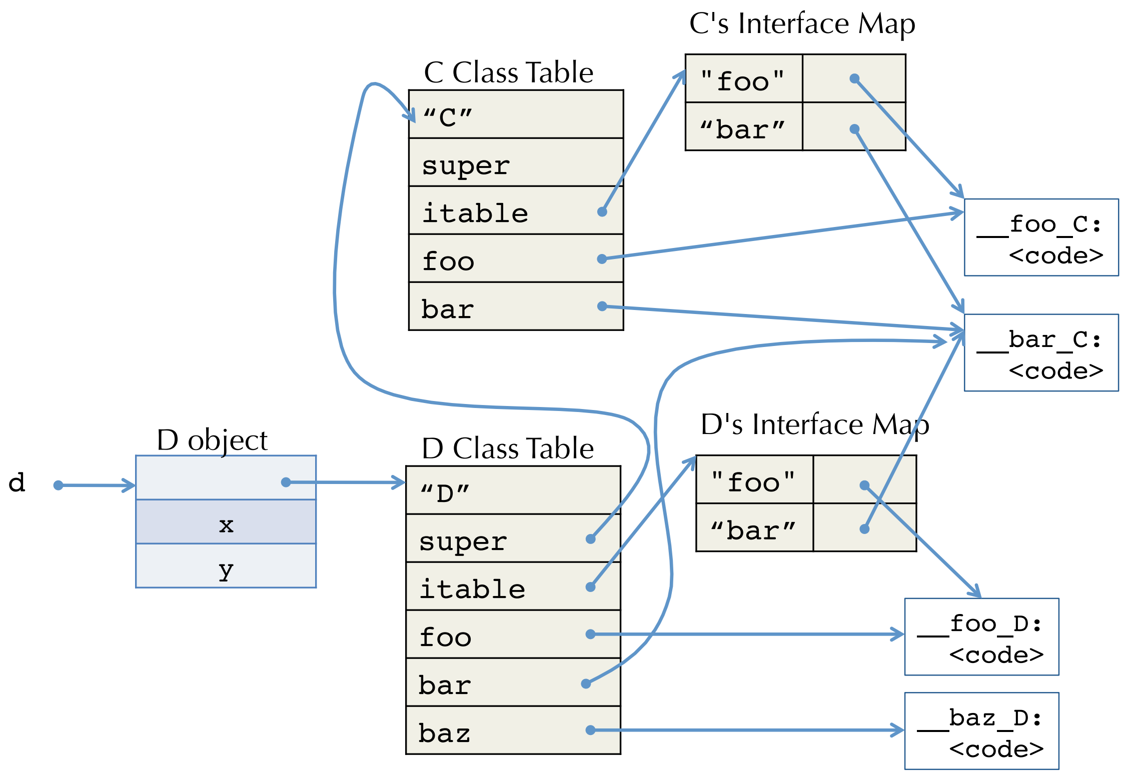
These questions refer to the Java code given in Appendix D.

Pick a strategy for compiling Java's multiple inheritance that is suitable for use with *separate compilation*—so class `C` could be compiled without referring to class `D`. (Of course `D` will need to know something about `C`, since `D` inherits from it.)

a. (3 points) Briefly (!) explain your compilation strategy.

Answer: There are several strategies. Here we will pick the common "use a level of indirection" with an `ITable` that can be searched to find the functions. Dispatch via an interface searches the `ITable`. Dispatch via a class uses the object's usual dispatch vector, which can be arranged by traversing the class tree to assign dispatch vector slots compatible with "width" subtyping. In this strategy, the compiler assigns each method (non-overloaded) method a unique identifier. We'll write `"foo''`, `"bar"`, and `"baz"` for them.

b. (8 points) Draw a picture of the class tables, an instance of the object constructed by the statement `D d = new D();`, and any other relevant dynamic state required by your implementation strategy. (Omit any fields or methods contributed by `Object`.)



c. (4 points) Suppose that a high-performance Java to native x86 compiler is claimed to support dynamic dispatch via interfaces with the same performance as dispatch via a class. For example: the method-dispatch overhead imposed by each of the two calls to `foo` in the following code would have *exactly* the same run-time cost (of course the bodies of the two method calls might perform differently):

```
static void callThem(I i, D d) {
  i.foo(); // dispatch to foo via an interface
  d.foo(); // dispatch fo foo via a class
}
```

Which of the following properties *must* this Java compiler have? (mark all that apply)

☐ Both calls to `foo` in the above code are *inlined* into the code for `callThem`.

☒ The compiler uses *whole-program compilation*, and so has access to all of the classes that can ever possibly be used.

☐ The dynamic dispatch implementation uses an *inline cache* to accelerate the calls.

☐ The compiler uses hashing to compute the layout of its dispatch vectors.

6. Optimization Miscellany (13 points total)

a. (8 points) So-called "peephole" optimizations simplify code by looking at a short sequences of instructions and replacing them equivalent but shorter code sequences. Each of the following at the x86 snippet "templates" can be replaced by a shorter instruction sequence. Assuming that `LOC`, `LOC1`, and `LOC2` refer to register or memory operands (not immediate values), suggest an equivalent one instruction replacement, or write "delete" if the snippet can be deleted without affecting the program behavior. We have done the first one for you.

```
0.    subq $8, %rsp             pushq LOC
      movq LOC, (%rsp)

1.    addq $0, LOC              delete

2.    movq LOC1, LOC2           movq LOC1, LOC2
      movq LOC2, LOC1

3.    popq  LOC                 movq (%rsp), LOC
      pushq LOC

4.    pushq LOC                 delete
      popq  LOC
```

b. (5 points) Briefly describe the purpose of tracking *move-related* edges when doing graph-coloring based register allocation. Give an example of which LL IR instruction would most benefit from this technique.

Answer: Move-related edges are a heuristic that can be used to improve register allocation. The idea is to remember when one identifier is moved to another and, if possible, to assign them the same color (i.e. the same register), which would mean that the move could be eliminated. Move-related identifiers can be forced to use the same register by "coalescing" their nodes in the interference graph.

The LLVM `bitcast` instruction compiles directly to a `movq` a the x86 level, so it would benefit from this optimization. The `call` instruction needs to take into account the register conventions, so it can also benefit. On x86, arithmetic instructions such as `add` can also benefit if one of the argument UIDs is allocated to the same register as the destination UID.

CIS341 Final Exam 2018 Appendices

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

APPENDIX A: Subtyping Inference Rules

Consider a variant of the typed lambda calculus whose types τ and terms e are generated by the following grammars:

$$\tau \ ::= \ \texttt{bool} \mid \bot \mid \tau_1 \to \tau_2$$

$$e \ ::= \ x \mid \texttt{true} \mid \texttt{false} \mid \lambda(x\!:\!\tau).\, e \mid e_1\, e_2$$

The subtyping relation for this language is given by the following collection of inference rules:

$$\boxed{\vdash \tau_1 <: \tau_2}$$

$$\frac{}{\vdash \texttt{bool} <: \texttt{bool}}\ [bool] \qquad \frac{}{\vdash \bot <: \tau}\ [bottom] \qquad \frac{\vdash \tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\vdash \tau_1 \to \tau_2 <: \tau_3 \to \tau_4}\ [fun]$$

The term type checking relation is given by the following collection of inference rules, where Γ is the typing context that associates variables with their types. Recall that we write $x\!:\!\tau \in \Gamma$ to mean that Γ maps x to type τ and we write $x \notin \Gamma$ to mean that x is not associated with any type in Γ. The empty context is written as "$\cdot$".

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

$$\frac{x\!:\!\tau \in \Gamma}{\Gamma \vdash x : \tau}\ [var] \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [true] \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ [false]$$

$$\frac{x \notin \Gamma \quad x\!:\!\tau_1, \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \lambda(x\!:\!\tau_1).\, e_2 : \tau_1 \to \tau_2}\ [lam] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \vdash \tau_1 <: \tau_2 \to \tau}{\Gamma \vdash e_1\, e_2 : \tau}\ [app]$$

APPENDIX B: LLVM Code for Break and Continue

```
define i64 @foo() {
  %_x1 = alloca i64
  store i64 1, i64* %_x1
  br label %_cond5
_cond5:
  br i1 1, label %_body4, label %_post3
_body4:
  %_x6 = load i64, i64* %_x1
  %_bop7 = icmp sgt i64 %_x6, 3
  br i1 %_bop7, label %_then18, label %_else17
_then18:
  ;;  <----------------- LOCATION 1
  br label %_merge16
_else17:
  %_x9 = load i64, i64* %_x1
  %_bop10 = add i64 %_x9, 1
  store i64 %_bop10, i64* %_x1
  ;;  <----------------- LOCATION 2
  %_str14 = getelementptr [17 x i8], [17 x i8]* @_str_arr13, i32 0, i32 0
  call void @print_string(i8* %_str14)
  br label %_merge16
_merge16:
  %_str20 = getelementptr [22 x i8], [22 x i8]* @_str_arr19, i32 0, i32 0
  call void @print_string(i8* %_str20)
  br label %_cond5
_post3:
  %_x22 = load i64, i64* %_x1
  ret i64 %_x22
}
```

APPENDIX C: OAT Compiler Code (excerpt)

Existing Frontend Code

```
(* Compile a statement *)
let rec cmp_stmt (tc : TypeCtxt.t) (c:Ctxt.t) (rt:Ll.ty) (stmt:Ast.stmt node)
                 : Ctxt.t * stream =
  match stmt.elt with

  | Ast.While (guard, body) ->
     let guard_ty, guard_op, guard_code = cmp_exp tc c guard in
     let lcond, lbody, lpost = gensym "cond", gensym "body", gensym "post" in
     let body_code = cmp_block tc c rt body in
     c, []
        >:: T (Br lcond)
        >:: L lcond >@ guard_code >:: T (Cbr (guard_op, lbody, lpost))
        >:: L lbody >@ body_code >:: T (Br lcond)
        >:: L lpost

  | ...  (* other cases omitted *)

(* Compile a series of statements *)
and cmp_block (tc : TypeCtxt.t) (c:Ctxt.t) (rt:Ll.ty) (stmts:Ast.block) : stream =
  snd @@ List.fold_left (fun (c, code) s ->
      let c, stmt_code = cmp_stmt tc c rt s in
      c, code >@ stmt_code
    ) (c,[]) stmts
```

Extensions to the AST

```
type stmt =
...
| Break
| Continue
| While of exp node * stmt node list
```

APPENDIX D: Example Java Code

```java
interface I { public void foo(); }

interface J { public void bar(); public void foo(); }

class C implements I, J {
   public int x;

   public void foo() { System.out.println("foo"); }

   public void bar() { System.out.println("bar"); }
}

class D extends C {
   private int y;

   private void baz() { System.out.println("baz"); }

   @Override
   public void foo() { System.out.println("D’s foo"); }
}
```

APPENDIX E: Example of accessibility analysis

This code shows the accessibility information for an example LLVM program. The comments after each statement summarize the accessibility facts at that point in the program. For example, the comment on line 15 says that `%node_ptr` $\rightsquigarrow$ `%node_ptr` and `%node_ptr` $\rightsquigarrow$ `%ptr1` and `%node_ptr` $\rightsquigarrow$ `%next_ptr`.

```
 1 %Node = type { i64, %Node* } ; struct Node { int element; Node? next }
 2
 3 define i64* @ptr_example6(%Node* %arg) {
 4   ; %arg -> {%arg}
 5   %ptr1 = call i64* @oat_malloc(i64 16)
 6   ; %arg -> {%arg},
 7   ; %ptr1 -> {%ptr1}
 8   %node_ptr = bitcast i64* %ptr1 to %Node*
 9   ; %arg -> {%arg},
10   ; %ptr1 -> {%ptr1, %node_ptr},
11   ; %node_ptr -> {%node_ptr, %ptr1}
12   %next_ptr = getelementptr %Node, %Node* %node_ptr, i32 0, i32 1
13   ; %arg -> {%arg},
14   ; %ptr1 -> {%ptr1, %node_ptr, %next_ptr},
15   ; %node_ptr -> {%node_ptr, %ptr1, %next_ptr}
16   ; %next_ptr -> {%next_ptr, %node_ptr, %ptr1}
17   store %Node* %arg, %Node** %next_ptr
18   ; %arg -> {%arg},
19   ; %ptr1 -> {%ptr1, %node_ptr, %next_ptr, %arg},
20   ; %node_ptr -> {%node_ptr, %ptr1, %next_ptr, %arg}
21   ; %next_ptr -> {%next_ptr, %node_ptr, %ptr1, %arg}
22   %next_ptr_of_arg = getelementptr %Node, %Node* %arg, i32 0, i32 1
23   ; %arg -> {%arg, %next_ptr_of_arg},
24   ; %ptr1 -> {%ptr1, %node_ptr, %next_ptr, %arg, %next_ptr_of_arg},
25   ; %node_ptr -> {%node_ptr, %ptr1, %next_ptr, %arg, %next_ptr_of_arg}
26   ; %next_ptr -> {%next_ptr, %node_ptr, %ptr1, %arg, %next_ptr_of_arg}
27   ; %next_ptr_of_arg -> {%arg, %next_ptr_of_arg}
28   ret i64* %ptr1
29 }
```