CIS 341 Final Examination
May 2020

# SOLUTIONS

## 1. Compilation (20 points total)

**a.** (15 points) Suppose we wanted to add a new "`repeat–until`" loop construct to the Oat language compiler you implemented for the class project. The new syntax is given by extending the Oat grammer as shown below:

$$\begin{array}{rcl} \mathit{stmt} & ::= & \ldots \\ & | & \texttt{repeat}\ \mathit{block}\ \texttt{until(}\mathit{exp}\texttt{)} \end{array}$$

Semantically this construct behaves like a `while` loop, except that the body block is guaranteed to execute at least once; the loop repeats whenever *exp* evaluates to `false` and terminates otherwise. Variables declared in *block* are not in scope inside of *exp*.

For each compiler stage below, indicate whether you would need to modify that stage to implement `repeat–until`. If a stage needs to be changed, briefly say how (no need to fill the boxes!). Aim for changing the *fewest* stages of the existing Oat compiler (as found in homework 5 and 6).

**Lexer:**

☐ no change needed

☒ changes needed:

Add the `repeat` and `until` tokens to the list of keywords.

**Parser:**

☐ no change needed

☒ changes needed:

(OPTIMAL): Extend the grammar to recognize the production above. modify the `stmt` production to return a list of statements and `block` to "flatten" this list. Desugar `repeat` *block* `until(`*exp*`)` into *block*`;while(!(`*exp*`))` *block*. (SUBOPTIMAL): Add a new kind of repeat-until Ast stmt node, parse the production above into that statement.

**Typechecker:**

☒ no change needed

☐ changes needed:

(SUBOPTIMAL): Extend the typechecker to check that "body" is well formed and "exp" is a boolean expression

(continued)

**Frontend:**

☒ no change needed

□ changes needed:

(SUBOPTIMAL): modify the translation to emit basic blocks similar to while, either by "desugaring to while" here, or by handling the logic explicitly.

**Backend:**

☒ no change needed

□ changes needed:

**b.** (5 points) Briefly explain one reason why you might choose a strategy for implementing `repeat–until` that requires changes to more compiler stages (or more extensive changes to those stages) than strictly necessary to provide the functionality. (No need to fill the box! A short sentence or two is enough.)

There are several good answers: (1) to improve error messages for the programmer (not easy if desugaring happens in the parser, (2) to generate smaller code (naive translation to while duplicates the body), (3) to make the compiler itself more maintanable

## 2. Inference Rules and Type Checking (30 points total)

The following questions refer the inference rules given in Appendix A. The language shown in Appendix A supports *generic types* (also known as *polymorphism*). Appendix B includes OCaml code that implements (most of) this type system.[1] Unlike OCaml (but a bit like Java), this language requires the programmer to explicitly bind and instantiate generic types. For example, the generic identity function `id` would be defined as $<\alpha>\lambda(x\!:\!\alpha).\,x$ and it has the generic type $<\alpha>(\alpha \to \alpha)$. Here, $\alpha$ is a *type variable* (usually written as `'a` in OCaml) that stands in for a type, the angle brackets indicate the presence of such a type parameter. To use a generic function like `id`, the programmer must explicitly instantiate the type parameter. For example: `id <bool> true` is a well-formed program.

There are two kinds of judgments: $\Delta \vdash \tau$ `ok` says that the type $\tau$ is well-scoped and has free type variable in $\Delta$. The judgment $\Delta; \Gamma \vdash e : \tau$ asserts that the program $e$ is well-formed and of type $\tau$ in the contexts $\Delta$ and $\Gamma$. The new rules for typechecking programs, compared to the simply typed lambda calculus, are $[generalize]$, which introduces a generic program that can mention a fresh type variable, and $[instantiate]$ which can be used to replace a generic type variable with a specific type instance. The $[instantiate]$ rule uses *type substitution*, defined below the inference rules.

**a.** (3 points)
☒ True or □ False:
There is a derivation of the following judgment according to the inference rules:

$$\beta \vdash <\alpha>(\alpha \to \beta)\ \texttt{ok}$$

**b.** (3 points)
□ True or ☒ False:
There is a derivation of the following judgment according to the inference rules:

$$\cdot \vdash <\alpha><\alpha>(\alpha \to \alpha)\ \texttt{ok}$$

**c.** (5 points) According to this type system, what type can be filled in for '?' in the judgment below to obtain a well-typed program? (Choose one.)

$$\cdot;\cdot \vdash <\alpha>\lambda(x\!:\!\alpha).\ <\beta>\lambda(y\!:\!\beta).\ x\ \ :?$$

□ $<\alpha><\beta>\alpha \to \beta \to \alpha$

□ $<\alpha><\beta>\alpha \to \beta \to \beta$

☒ $<\alpha>\alpha \to <\beta>\beta \to \alpha$

□ $<\alpha>\alpha \to <\beta>\beta \to \beta$

□ This term is ill-typed.

[1] NOTE: One way to answer the questions in this problem is to complete the implementation (see part **f.**) and run the code on appropriate test cases. However, these questions should be answerable *without* resorting to doing that. Should you want it, the code of Appendix B is available in the supplementary file `tcpoly.ml` available on the course web page and Piazza.

**d.** (5 points) According to this type system, what type can be filled in for '?' in the judgment below to obtain a well-typed program? (Choose one.)

$$\cdot;\cdot \vdash \lambda(f\!:\!\texttt{<}\alpha\texttt{>}\ \alpha \to \texttt{bool}).\,(f\ \texttt{<bool>}\ \texttt{true})\ :\ ?$$

☒ $(\texttt{<}\alpha\texttt{>}\alpha \to \texttt{bool}) \to \texttt{bool}$

□ $\texttt{<}\alpha\texttt{>}(\alpha \to \texttt{bool} \to \texttt{bool})$

□ $(\texttt{<}\alpha\texttt{>}\alpha \to \texttt{bool}) \to \alpha$

□ $\texttt{<}\alpha\texttt{>}(\alpha \to \texttt{bool} \to \alpha)$

□ This term is ill-typed.

**e.** (4 points) Polymorphism is a strong property. One could rigorously prove (although we won't get that detailed here) that there is no closed program $e$ such that the following judgment is provable.

$$\cdot;\cdot \vdash e : \texttt{<}\alpha\texttt{>}(\texttt{bool} \to \alpha)$$

Briefly give an explanation as to why there can be no such $e$. (Hint: think about what it would take to be able to implement such a generic function.)

Since we can instantiate $\alpha$ with any type, such a function would be a generic algorithm that can construct a value of any type at all, given just a boolean as input. As an even more extreme case, the generic type $\texttt{<}\alpha\texttt{>}\alpha$ could be instantiated to any type too and it would have to create a value of that type using no inputs!

**f.** (10 points) Appendix B contains OCaml code for (most of) an implementation of the type checking algorithm described by the rules in Appendix A. (This code is an adaptation of the lambda calculus code from lecture, and is also similar to the typechecker you implemented for HW5.) The `typecheck` function is missing the cases for the new expressions. Fill in the blanks below to complete their implementations. If you need to raise an error, use `failwith` and give a (somewhat) descriptive message. (The code in each case is at most a few lines long.)

```
let rec typecheck (delta:typ_environment) (gamma:environment) (e:exp) : tp =
  begin match e with
   (* other cases from Appendix B *)

  | Gen(alpha, e1) ->
      if contains alpha delta then failwith "type variable aliased"
      else GenT(alpha, typecheck (alpha::delta) gamma e1)

  | Ins(e1, t) ->
      let _ = wft delta t in
      begin match (typecheck delta gamma e1) with
        | GenT(alpha, t1) -> tsubst t1 t alpha
        | _ -> failwith "tried to instantiate non-generic type"
      end
```

## 3. Dataflow Analysis (36 points total)

Dataflow analysis is most often used by compilers for discovering facts that enable optimization, but it has broad applicability. In this problem we consider how dataflow analysis can be adapted to help improve software *security* by identifying data values that might become "tainted" by untrustworthy sources. Such an analysis might help discover security flaws in which input is inappropriately used—for instance if the user-supplied data is interpreted as part of an SQL query string or as part of a shell command, maliciously constructed inputs could release or corrupt database information, delete files, or worse.

We will develop (a simplified version of) this analysis in the context of the LLVM IR you are familiar with from class (ignoring `getelementptr` for simplicity).

**a.** (6 points) The first step is to construct a lattice of dataflow facts. For simplicity, our analysis will track, for each UID of the program, a *taint level* draw from the set $\mathit{Level} = \{T, U\}$, where $T$ stands for "may be tainted" and $U$ for "definitely untainted". We order them as $T \sqsubseteq U$. Note that $T \sqcap U = T$.

An element of our dataflow lattice is a (finite) map $m : \mathit{UID} \rightarrow \mathit{Level}$. Let us write $\{\texttt{\%x} \mapsto U,\ \texttt{\%y} \mapsto T\}$ for the map that sends UID `%x` to label $U$ and `%y` to the label $T$ (and we'll use similar notations that let us treat maps as sets of key–value bindings). We'll use the notation $m_1(\texttt{\%u})$ to lookup the label associated with `%u` in $m_1$. Maps are ordered *pointwise*, and we write $m_1 \sqsubseteq m_2$ to mean that map $m_1$ is less-than-or-equal-to map $m_2$ (overloading the $\sqsubseteq$ symbol).

Suppose $m = \{\texttt{\%x} \mapsto U,\ \texttt{\%y} \mapsto T\}$. Which of the following relations hold? (Mark all that apply.)

☒ $\{\texttt{\%x} \mapsto U,\ \texttt{\%y} \mapsto T\} \sqsubseteq m$

□ $\{\texttt{\%x} \mapsto U,\ \texttt{\%y} \mapsto U\} \sqsubseteq m$

□ $\{\texttt{\%x} \mapsto T,\ \texttt{\%y} \mapsto U\} \sqsubseteq m$

☒ $\{\texttt{\%x} \mapsto T,\ \texttt{\%y} \mapsto T\} \sqsubseteq m$

□ $\{\texttt{\%y} \mapsto U\} \sqsubseteq m$

☒ $\{\texttt{\%y} \mapsto T\} \sqsubseteq m$

**b.** (2 points) The "meet" of two lattice elements, written $m_1 \sqcap m_2$, is defined as the map such that for all UIDs `%u`, $(m_1 \sqcap m_2)(\texttt{\%u}) = m_1(\texttt{\%u}) \sqcap m_2(\texttt{\%u})$. For distinct UIDs `%x` and `%y` we also want:

$$\{\texttt{\%x} \mapsto \ell_1\} \sqcap \{\texttt{\%y} \mapsto \ell_2\} = \{\texttt{\%x} \mapsto \ell_1, \texttt{\%y} \mapsto \ell_2\}$$

With these definitions, we can consider UID's *not appearing* in a map $m$ to have which label? That is, for any $\texttt{\%u} \notin \mathit{dom}(m)$, it is as though $m(\texttt{\%u}) = X$. What is $X$?

□ $X$ is $T$

☒ $X$ is $U$

Given the setup above, we extend our lattice elements (label maps) $m$ so that literals appearing in the program are considered to be untainted, so we have, for example, $m(\texttt{3}) = U$ (and similarly for all other `i64` literals). This means that for an LLVM IR operand `op`, which is either a UID or a literal, we can write $m(\texttt{op})$ for its taint level.

Next we must consider the *flow functions* $F_n$ for each instruction $n$ of the LLVM IR. The idea here is that a computation that uses (possibly) tainted inputs produces a (possibly) tainted output — to be *sound* our flow functions should be conservative approximations. Some instructions' flow functions are easy to describe. For example, the bitcast instruction (shown in the table below) simply extends the map $m$ to include the newly defined UID `%ans` with the same taint level as the incoming `%ptr` argument.

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = bitcast ty1* %ptr to ty2*` | $m \sqcap \{\texttt{\%ans} \mapsto m(\texttt{\%ptr})\}$ |

For arithmetic operations, we want to conservatively propagate taint: if either input is tainted, so is the result. Below we define the flow function for `add`; the other arithmetic operations (`sub`, `mul`, etc.) and `imcp` use the same flow function.

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = add i64 op1, op2` | ? |

**c.** (3 points) What map should we fill in for the ? in the table above? (Choose one.)

☐ $\{\texttt{\%op1} \mapsto m(\texttt{\%op2})\}$

☐ $\{\texttt{\%ans} \mapsto m(\texttt{\%op1}) \sqcap m(\texttt{\%op2})\}$

☐ $\{\texttt{\%op1} \mapsto m(\texttt{\%ans}) \sqcap m(\texttt{\%op2}))\}$

☐ $m \sqcap \{\texttt{\%op1} \mapsto m(\texttt{\%op2})\}$

☒ $m \sqcap \{\texttt{\%ans} \mapsto m(\texttt{\%op1}) \sqcap m(\texttt{\%op2})\}$

☐ $m \sqcap \{\texttt{\%op1} \mapsto m(\texttt{\%ans}) \sqcap m(\texttt{\%op2}))\}$

Because the `alloca` instruction has no input operands its flow-function is straightforward, and there are *two* possible sound answers:

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = alloca typ` | $m \sqcap \{\texttt{\%ans} \mapsto T\}$ *or* $m \sqcap \{\texttt{\%ans} \mapsto U\}$ |

**d.** (2 points) Which one is more precise?

☐ $F_n(m) = m \sqcap \{\texttt{\%ans} \mapsto T\}$ is more precise.

☒ $F_n(m) = m \sqcap \{\texttt{\%ans} \mapsto U\}$ is more precise.

Defining the flow functions for `call` instructions is more challenging. In general, whether or not the result of a call to a function `f(arg1, ..., argN)` is tainted depends on how the code in `f` uses its arguments (and whether they are tainted) and also on whether `f` gets tainted data from elsewhere (perhaps by calling some other function or by accessing a tainted global). Here we simplify considerably and define the flow functions for `call` as shown below, with the functions `tainted_input` and `sanitize` being treated specially:

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = call i64 tainted_input()` | $m \sqcap \{\texttt{\%ans} \mapsto T\}$ |
| `%ans = call i64 sanitize(i64 op)` | $m \sqcap \{\texttt{\%ans} \mapsto U\}$ |
| `%ans = call ty f(op_1, ..., op_n)` | $m \sqcap \{\texttt{\%ans} \mapsto \bigsqcap_i m(\texttt{op\_}i)\}$ |

**e.** (8 points) Which of the following statements characterize the assumptions we are making by using the definition above? (Mark all that apply.)

☒ The (known) function `i64 tainted_input()` takes no arguments and produces a tainted `i64` value (modeling “untrusted user input”).

☒ The (known) function `i64 sanitize(i64 op)` returns an untainted result no matter what its input is (modeling “validation of user input”).

☒ For (unknown) function `f`, we assume that `f` produces a tainted result only if *at least one* of its inputs are tainted.

□ For (unknown) function `f`, we assume that `f` produces a tainted result only if *all* of its inputs are tainted.

Tracking taint through the LLVM IR `load` and `store` instructions is also a bit subtle. A first, *unsound* attempt at defining flow functions for `load` and `store` is shown below:

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = load ty* op` | $m \sqcap \{\texttt{\%ans} \mapsto m(op)\}$ |
| `store ty op, ty* %ptr` | $m \sqcap \{\texttt{\%ptr} \mapsto m(op)\}$ |

**f.** (5 points) Briefly explain *why* the definition is unsound.

Pointers can be aliased, but updating taint associated with `%ptr` in the `store` instruction doesn't update any of its aliases, so one could write "tainted" information through one pointer and read "untainted" data from an alias.

One option to fix the unsoundness would be to change the flow function for `load` as shown below so that we conservatively assume that any value loaded from memory might be tainted. This isn't great because everything read from memory would be considered to be tainted, leading to lots of "false positives."

| instruction $n$ | $F_n(m) =$ |
|---|---|
| `%ans = load ty* op` | $m \sqcap \{\texttt{\%ans} \mapsto T\}$ |

A more precise option is to instead change the flow function for `store` by making use of the results of a different dataflow analysis.

**g.** (2 points) What other analysis would you use to improve the precision of the flow function for `store`?

alias analysis

**h.** (3 points) Briefly explain how the flow function for `store` can exploit that analysis:

Use the results of alias analysis to modify the `store` flow function to update the taint levels of all possible aliases of `%ptr`. Technically, we have $F_n(m) = m \sqcap \{\texttt{\%a} \mapsto m(op) \mid \texttt{\%a} \in mayalias(\texttt{\%ptr})\}$ `load` can use the flow-function given above part f.

To complete the story, we observe that the flow functions for block terminators simply propagate the current taint map: $F_n(m) = m$ when $n$ is a block terminator. With that, we can plug our taint-tracking fact lattice into the general dataflow fixpoint solver (as we used for HW6) in *forward* mode. The resulting analysis will compute taint maps for each edge of the control flow graph.

Consider the following LLVM IR program that loops a variable number of times, accumulating an answer into the (location) pointed to by `%acc`.

```
define i64 @example(i64 %n) {
  %ctr = alloca i64
  %acc = alloca i64
  store i64 %n, i64* %ctr
  store i64 1, i64* %acc
  br label %start

  ;; ** HERE **
start:
  %2 = load i64, i64* %ctr
  %3 = icmp sgt i64 %2, 0
  br i1 %3, label %then, label %end
then:
  %4 = load i64, i64* %acc
  %5 = call i64 tainted_input()
  %6 = add i64 %4, %5
  store i64 %6, i64* %acc
  %7 = load i64, i64* %ctr
  %8 = sub i64 %7, 1
  store i64 %8, i64* %ctr
  br label %start
end:
  %9 = load i64, i64* %acc
  ret i64 %9
}
```

**i.** (5 points) Let the map $m$ be the fixpoint solution computed for the in-edge of the `start` block, marked as `** HERE **` above. For each UID $u$ below, what is the taint value $m(u)$? [2]

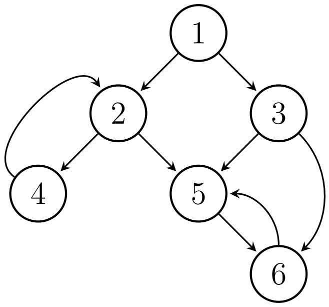
$m(\%\texttt{ctr})$ = ☒ U □ T
$m(\%\texttt{acc})$ = □ U ☒ T
$m(\%2)$ = ☒ U □ T
$m(\%3)$ = ☒ U □ T
$m(\%4)$ = □ U ☒ T

$m(\%5)$ = □ U ☒ T
$m(\%6)$ = □ U ☒ T
$m(\%7)$ = ☒ U □ T
$m(\%8)$ = ☒ U □ T
$m(\%9)$ = ☒ U □ T

[2] Technically, the answer to this question would depend on how you implement the flow functions for `load` and `store` as in part **h.**. Here assume that they are (magically) defined as precisely as possible.

## 4. Control-flow Analysis (16 points total)

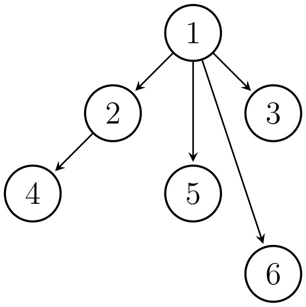
The following questions concern the following control-flow graph, where the nodes numbered 1–6 represent *basic blocks* and the arrows denote control-flow edges. The entry point is node 1.



**a.** (6 points) "Draw" the dominator tree by putting an "X" in each box where there is an immediate dominator edge from a source node to a target node. (There are no "self edges" so we leave those blanks empty.)

|  | tgt: | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| src: |  |  |  |  |  |  |  |
| 1 |  |  | ☒ | ☒ | □ | ☒ | ☒ |
| 2 |  | □ |  | □ | ☒ | □ | □ |
| 3 |  | □ | □ |  | □ | □ | □ |
| 4 |  | □ | □ | □ |  | □ | □ |
| 5 |  | □ | □ | □ | □ |  | □ |
| 6 |  | □ | □ | □ | □ | □ |  |

**Solution Picture:**



**b.** (3 points) Which of following nodes are in the *dominance frontier* for node 2?

□ 1 ☒ 2 □ 3 □ 4 ☒ 5 □ 6

**c.** (3 points) Which of following nodes are in the *dominance frontier* for node 3?

□ 1 □ 2 □ 3 □ 4 ☒ 5 ☒ 6

**d.** (4 points) Could this control flow graph be generated by compiling a well-formed Oat program using the HW06 frontend? Briefly explain.

☒ No □ Yes

This control flow graph cannot be generated by compiling a well-formed Oat program using the HW06 frontend because the only looping constructed in Oat is "while" but it compiles to a set of blocks independently of the "if" statement, so you can't have either conditional branch targeting a non-entry node of a while loop.

## 5. LLVM IR and Optimization Miscellany (18 points total)

**a.** (6 points) Consider the LLVM IR's `phi` instruction, shown below, and assume that it appears as part of some block of code in a well-formed control-flow graph.

```
%ans = phi i64 [%y, %block1], [%ans, %block2]
```

Which of the following statements must be true of this LLVM IR program?

☒ The block containing the instruction above has exactly two *predecessor* blocks named `%block1` and `%block2` in the control-flow graph.

□ The block containing the instruction above has exactly two *successor* blocks named `%block1` and `%block2` in the control-flow graph.

□ The block containing the instruction above must itself be named `%block1`

□ The block containing the instruction above must itself be named `%block2`.

□ The control-flow graph contains a back edge whose source is the block labeled by `%block1`

☒ The control-flow graph contains a back edge whose source is the block labeled by `%block2`

**b.** (6 points) Briefly explain one reason why optimizing a program by *unrolling loops* is not always guaranteed to improve performance, even though doing so cuts down the total number of conditional branches executed by the code.

(1) The resulting loop body code might exceed the processor's instruction cache, which means that it might need to do many more memory accesses to run the instructions. (2) The processor's branch prediction might effectively achieve the same behavior as loop unrolling anyway, so the performance may not improve.

**c.** (6 points) Briefly explain why, when using Kempe's graph-coloring algorithm for register allocation, it is useful to mark a node a "possibly spilled" when simplifying the graph, but defer the decision to actually spill it until the coloring phase.

Even though the node is marked as spilled, there might be a color available for it during the coloring phase. If there is one available, the coloring is still guaranteed to succeed, but there will be fewer stack accesses, so the performance will almost certainly improve.

# CIS341 Final Exam 2020 Appendices

# APPENDIX A: Polymorphism Inference Rules

Consider a variant of the typed lambda calculus whose types $\tau$ and terms $e$ are generated by the following grammars (we also use parentheses when writing terms to disambiguate their parses):

$$\tau \ ::= \ \texttt{bool} \mid \tau_1 \to \tau_2 \mid \texttt{<}\alpha\texttt{>}\ \tau \mid \alpha$$

$$e \ ::= \ x \mid \texttt{true} \mid \texttt{false} \mid \lambda(x{:}\tau).\ e \mid e_1\ e_2 \mid \texttt{<}\alpha\texttt{>}\ e \mid e\ \texttt{<}\tau\texttt{>}$$

The term type checking relation is given by the following collection of inference rules, where $\Delta$ is the set of type variables (written $\alpha$, $\beta$, etc.) that are in scope, and $\Gamma$ is the typing context that associates variables with their types. Recall that we write $x{:}\tau \in \Gamma$ to mean that $\Gamma$ maps $x$ to type $\tau$ and we write $x \notin \Gamma$ to mean that $x$ is not associated with any type in $\Gamma$. An empty context is written as "$\cdot$".

$$\boxed{\Delta \vdash \tau\ \texttt{ok}}$$

$$\frac{}{\Delta \vdash \texttt{bool ok}}\ [\mathit{TBool}] \qquad \frac{\Delta \vdash \tau_1\ \texttt{ok} \qquad \Delta \vdash \tau_2\ \texttt{ok}}{\Delta \vdash \tau_1 \to \tau_2\ \texttt{ok}}\ [\mathit{TArr}]$$

$$\frac{\alpha \notin \Delta \qquad \Delta, \alpha \vdash \tau\ \texttt{ok}}{\Delta \vdash \texttt{<}\alpha\texttt{>}\ \tau\ \texttt{ok}}\ [\mathit{TGeneric}] \qquad \frac{\alpha \in \Delta}{\Delta \vdash \alpha\ \texttt{ok}}\ [\mathit{TVar}]$$

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

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

$$\frac{x \notin \Gamma \quad \Delta \vdash \tau_1\ \texttt{ok} \quad x{:}\tau_1, \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \lambda(x{:}\tau_1).\ e_2 : \tau_1 \to \tau_2}\ [\mathit{lam}]$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \to \tau \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau}\ [\mathit{app}]$$

$$\frac{\alpha \notin \Delta \quad \Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \texttt{<}\alpha\texttt{>}\ e : \texttt{<}\alpha\texttt{>}\ \tau}\ [\mathit{generalize}] \qquad \frac{\Delta; \Gamma \vdash e : \texttt{<}\alpha\texttt{>}\ \tau_1 \quad \Delta \vdash \tau\ \texttt{ok}}{\Delta; \Gamma \vdash e\ \texttt{<}\tau\texttt{>} : \tau_1\{\tau/\alpha\}}\ [\mathit{instantiate}]$$

The *type substitution* operation $\tau_1\{\tau/\alpha\}$ replaces free occurrences of $\alpha$ by $\tau$ in the type $\tau_1$ and it is defined recursively on the structure of $\tau_1$:

$$\texttt{bool}\{\tau/\alpha\} = \texttt{bool} \qquad (\tau_1 \to \tau_2)\{\tau/\alpha\} = \tau_1\{\tau/\alpha\} \to \tau_2\{\tau/\alpha\}$$

$$(\texttt{<}\beta\texttt{>}\ \tau_1)\{\tau/\alpha\} = \begin{cases} \texttt{<}\beta\texttt{>}\ \tau_1 & \text{if } \alpha = \beta \\ \texttt{<}\beta\texttt{>}\ (\tau_1\{\tau/\alpha\}) & \text{if } \alpha \neq \beta \end{cases} \qquad \beta\{\tau/\alpha\} = \begin{cases} \tau & \text{if } \alpha = \beta \\ \beta & \text{if } \alpha \neq \beta \end{cases}$$

# APPENDIX B: Typechecking Code

Note: For your reference the code in this appendix is also available for download as the supplementary file `tcpoly.ml` available on the course web site and linked to on Piazza.

```
module Fun = struct
  type tvar = string   (* type variables, "alpha" "beta" etc. *)
  type var = string    (* term variables, "x", "y", "f", etc. *)

  (* types *)
  type tp =
    | BoolT                 (* the type of booleans *)
    | FunT of tp * tp       (* ty1 -> ty2 the type of functions *)
    | VarT of tvar          (* type variables "alpha", "beta", "gamma" *)
    | GenT of tvar * tp     (* generic type: <a>t *)

  (* Abstract syntax of "polymorphic lambda terms" *)
  type exp =
    | Lit of bool
    | Var of var                (* local variables *)
    | Fun of var * tp * exp     (* functions:  fun (x:t) -> e *)
    | App of exp * exp          (* function application *)
    | Gen of tvar * exp         (* generic type parameter *)
    | Ins of exp * tp           (* type instantiation *)

end

(* This module implements a simple type checker for the polymorphic language *)
module TC = struct
  open Fun

  type typ_environment = tvar list
  type environment = (var * tp) list

  let contains (alpha:tvar) (delta:typ_environment) = List.mem alpha delta
  let lookup x (gamma:environment) = List.assoc x gamma

  (* Well-formed types:  returns unit if delta |- t ok  and fails otherwise *)
  let rec wft (delta:typ_environment) (t:tp) : unit =
    begin match t with
      | BoolT -> ()
      | FunT(t1, t2)-> wft delta t1 ; wft delta t2
      | VarT alpha ->
        if contains alpha delta then ()
        else failwith "type variable not in scope"
      | GenT(alpha, t1) ->
        if contains alpha delta
        then failwith "rebound type variable"
        else wft (alpha::delta) t1
    end

  (* ... continued ... *)
  (* Type substitution: implements t1{t/alpha} *)
  let rec tsubst (t1:tp) (t:tp) (alpha:tvar) : tp =
    begin match t1 with
      | BoolT -> BoolT
      | FunT(t11, t12) -> FunT(tsubst t11 t alpha, tsubst t12 t alpha)
      | VarT (beta) ->
        if alpha = beta then t else VarT(beta)
      | GenT(beta, t11) ->
        if alpha = beta then t1 else GenT(beta, tsubst t11 t alpha)
    end

  (* Compute the type of an exp, raising an error if there is no such type
     Returns the type t such that  delta ; gamma |- e : t
  *)
  let rec typecheck (delta:typ_environment) (gamma:environment) (e:exp) : tp =
    begin match e with
      | Lit i -> BoolT

      | Var x -> lookup x gamma

      | Fun (arg, t, body) ->
        let _ = wft delta t in
        FunT(t, typecheck delta ((arg,t)::gamma) body)

      | App (e1, e2) ->
        begin match (typecheck delta gamma e1, typecheck delta gamma e2) with
          | (FunT(t1, t2), t3) ->
            if t1 = t3 then t2
            else failwith "function argument mismatch"
          | _ -> failwith "tried to apply non-function"
        end

      | Gen(alpha, e1) ->  (* TODO *)

      | Ins(e1, t) ->   (* TODO *)

    end
end (* TC Module *)
```

**KUDOS-ONLY BONUS QUESTION:**
There is a subtle bug in the code for the App case of typecheck. This bug *does not* affect any of the answers to questions in this exam, and isn't relevant to the code you are asked to write. What is the bug? When would it be triggered? How would you fix it? (Hint: the code is *correct* for the simply-typed lambda calculus; adding polymorphic types breaks it.) Mail your answer to cis341@seas.upenn.edu.

*DO NOT WORRY ABOUT OR ATTEMPT THIS BONUS PROBLEM UNLESS YOU FINISH THE REST OF THE EXAM AND ARE BORED*