CS421 Sample Final
NO SOLUTION GIVEN

Someday, December ?? 20??, sometime

Name:	
NetID:	

- You have **180 minutes** to complete this exam.
- This is a **closed-book** exam. You are allowed only this exam and writing materials. **All other materials (e.g., notes, note cards, calculators and cell phones), are prohibited.**
- Do not share anything with other students. Do not talk to other students. Do not look at another students exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, you may seek clarification from myself or one of the TAs. You must use a whisper, or write your question out. Speaking out aloud is not allowed.
- Including this cover sheet and rules at the end, there are 27 pages to the exam, each on a separate sheet. Please verify that you have all 27 pages.
- There are 160 points on the base part of this exam, so that one point on this exam is worth the same as one point on either of the midterms. There are 20 points of extra credit.
- Please write your name and NetID in the spaces above, and also in the provided space at the top of every sheet.

Question	Points	Score
1	7	
2	16	
3	18	
4	17	
5	12	
6	10	
7	18	
8	12	
9	14	
10	12	
11	12	
12	12	
Total:	160	

Question	Extra Credit	Score
13	0	
14	10	
Total:	10	

Problem 1. (7 points)

Write a function `is_sorted_ascend : 'a list -> bool` that checks if the input list is sorted in ascending order. Note, [1;2;2;3;5] is sorted in ascending order. A list is sorted in ascending order if each element in the list is less than or equal to every element that comes after it in the list.

```
# let rec is_sorted_ascend l = ...;;
val is_sorted_ascend : 'a list -> bool = <fun>
# is_sorted_ascend [1;2;3];;
- : bool = true
```

Problem 2. (16 points)

Consider the following code:

```
let rec exists p list =
    match list with [] -> false
                | (x :: xs) -> if p x then true else exists p xs
```

(a) (2 points) Is the definition of `exists` forward recursive, tail recursive, both or neither?

(b) (6 points) Write `exists : ('a -> bool) -> 'a list -> bool` using `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`, but with no other source of recursion.

(c) (8 points) Write the function
`existsk : ('a -> (bool -> 'b) -> 'b) -> 'a list -> (bool -> 'b) -> 'b`
that is the complete Continuation Passing Style transformation **of the original code given for** `exists`.

Problem 3. (18 points)

(a) (8 points) Write an `bool_exp` data type in Ocaml that represents boolean expressions. Specifically, the `bool_exp` data type must be able to represent the following:

- a variable with its name given by a string
- the negation of a boolean expression
- the conjunction of two boolean expressions
- the disjunction of two boolean expressions

(b) (10 points) Write a function `bool_eval:(string -> bool) -> bool_exp -> bool` that takes a function giving boolean values for each variable and returns the boolean value of the boolean expression given using your datatype above.

Problem 4. (17 points)

You are asked to **implement the case for conditionals (`if_then_else`)** for the function:

```
gather_exp_ty_substitution
  : type_env -> exp -> monoTy -> (proof * substitution) option
```

from MP6. We will call this function `gather` to reduce the amount of writing needed.

You must add comments indicating how the code you give relates to the specification given. The comments may be short, but they should be clear.

Background and utility procedures:
Details of the specification of the problem and types and utility functions you may use follow. You are allowed to use `List` library functions. Alternately, you may wish to write auxiliary functions. If you do so, it is acceptable to put their code after the code for `gather`.

The code given for MP6 in the `Mp6common` module includes the following elided data types to represent the terms and types of PicoML:

```
type exp =
   | VarExp of string                    (* variables *)
   | ConstExp of const                   (* constants *)
   .  .  .
   | IfExp of exp * exp * exp            (* if exp1 then exp2 else exp3 *)
   .  .  .
```

and

```
type typeVar = int
type monoTy = TyVar of typeVar | TyConst of (string * monoTy list)
type polyTy = typeVar list * monoTy  (* the list is for quantified variables *)
```

When inferring types, you will need to generate fresh type-variable names. For this, you may use the side-effecting function `fresh: unit -> monoTy` that returns a fresh type variable. You may also find it useful to use the term

```
let bool_ty = TyConst("bool",[])
```

We associate polymorphic types with term variables via a type environment. We represent type environments with the type:

```
type 'a env = (string * 'a) list
type type_env = polyTy env
```

One interacts with environments using the following functions:

```
val make_env : string -> 'a -> 'a env = <fun>           (*create env with single pair
val lookup_env : 'a env -> string -> 'a option = <fun>  (*look up x in env*)
val sum_env : 'a env -> 'a env -> 'a env = <fun>        (*update snd env with first*)
val ins_env : 'a env -> string -> 'a -> 'a env = <fun>  (*insert x|->y into env*)
```

The result returned by the function `gather : type_env -> exp -> monoTy -> (proof * substitution) option` contains two components, a `proof` and a `substitution`. The type of `proof` is given by the following:

```
type proof = Proof of proof list * judgment
```

where

```
type judgment =
   ExpJudgment of type_env * exp * monoTy
 | DecJudgment of type_env * dec * type_env
```

Substitutions are represented as lists of pairs in much the same way as environments:

```
type substitution = (typeVar * monoTy) list
```

To apply a substitution to each of monomoprhic types, polymporphic types and environments, you are given the following functions:

```
val monoTy_lift_subst : substitution -> monoTy -> monoTy
val polyTy_lift_subst : substitution -> polyTy -> polyTy
val env_lift_subst : substitution -> env -> env
```

You also need to be able to create the substitution that represents the composition of two substitutions. If `s1` and `s2` are two substitutions, then

```
val subst_compose : substitution -> substitution -> substitution
```

can be used to generate their composition.

$$\texttt{subst\_compose s1 s2} = \texttt{s1} \circ \texttt{s2}$$

Main problem statement:
Implement the rule for `if_then_else`:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \mid \sigma_1 \quad \sigma_1(\Gamma) \vdash e_2 : \sigma_1(\tau) \mid \sigma_2 \quad \sigma_2 \circ \sigma_1(\Gamma) \vdash e_3 : \sigma_2 \circ \sigma_1(\tau) \mid \sigma_3}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau \mid \sigma_3 \circ \sigma_2 \circ \sigma_1}$$

To start the function, we have given the code for the constant case. Extend the following code with the case for conditionals. **You must add comments indicating how the code you give relates to the specification given. The comments may be short, but they should be clear.**

```
open Mp6common

let rec gather gamma exp tau =
    let judgment = ExpJudgment(gamma, exp, tau) in
    match exp
    with ConstExp c ->
         let tau' = const_signature c in
         (match unify [(tau, freshInstance tau')]
          with None       -> None
             | Some sigma -> Some(Proof([],judgment), sigma))
    | (* your code here ... *)
```

Problem 5. (12 points)

In MP7, you were asked to implement a unification algorithm with the function

```
unify : (monoTy * monoTy) list -> substitution option
```

In this problem you are asked to **implement the Decompose rule** for this algorithm.

You must add comments indicating how the code you give relates to the specification given. The comments may be short, but they should be clear.

Background and utility procedures:
Details of the specification of the problem and types and utility functions you may use follow. You are allowed to use `List` library functions. Alternately, you may wish to write auxiliary functions. If you do so, it is acceptable to put their code after the code for `unify`.

The code given for MP7 in the `Mp7common` module includes the following data types to represent the types of PicoML:

```
type typeVar = int
type monoTy = TyVar of typeVar | TyConst of (string * monoTy list)
```

The unification algorithm takes a set of pairs of types that need to be made equal. A system of constraints looks like the following set

$$\{(s_1, t_1), (s_2, t_2), ..., (s_n, t_n)\}$$

Each pair is called an *equation*. A (lifted) substitution ϕ *solves* an equation (s, t) if $\phi(s) = \phi(t)$. It solves a constraint set if $\phi(s_i) = \phi(t_i)$ for every (s_i, t_i) in the constraint set. The unification algorithm will return a substitution that solves the given constraint set (if a solution exists).

You will remember from lecture that the unification algorithm consists of four transformations. These transformations can be expressed in terms of how an action on an element of the unification problem affects the remaining elements.

Given a constraint set C

1. If C is empty, return the identity substitution.
2. If C is not empty, pick an equation $(s, t) \in C$. Let C' be $C \setminus \{(s, t)\}$.
   (a) **Delete rule:** If s and t are are equal, discard the pair, and unify C'.
   (b) **Orient rule:** If t is a variable, and s is not, then discard (s, t), and unify $\{(t, s)\} \cup C'$.
   (c) **Decompose rule:** If $s =$ `TyConst(`$name, [s_1; \ldots; s_n]$`)` and $t =$ `TyConst(`$name, [t_1; \ldots; t_n]$`)`, then discard (s, t), and unify $C' \cup \bigcup_{i=1}^{n} \{(s_i, t_i)\}$.
   (d) **Eliminate rule:** If s is a variable, and s does not occur in t, substitute s with t in C' to get C''. Let ϕ be the substitution resulting from unifying C''. Return ϕ updated with $s \mapsto \phi(t)$.

(e) If none of the above cases apply, it is a unification error (your `unify` function should return the `None` option in this case).

In our system, function, integer, list, etc. types are the terms; `TyVar`s are the variables.

Main problem statement: Give the case in the `unify: (monoTy * monoTy) list -> (typeVar * monoTy) list option` function that implements the **Decompose rule**. You may use elisions (. . .) for the omitted cases. You are allowed to use `List` library functions. Alternately, you may wish to write auxiliary functions. If you do so, it is acceptable to put their code after the code for `unify`.

```
let rec unify eqlst =
  match eqlst with [] -> Some([])
  |
```

Problem 6. (10 points)

In MP10, you were asked to write a function `eval_exp : (exp * value env) -> value` for evaluating expressions in `PicoML` based on a family of Natural Semantics rules. **Extend `eval_exp (exp, m)` to handle `if` constructs.**

You must add comments indicating how the code you give relates to the specification given. The comments may be short, but they should be clear.

Background and utility procedures:

You were given the following (here abbreviated) types and functions:

```
type exp =
   | VarExp of string                    (* variables *)
   | ConstExp of const
   | IfExp of exp * exp * exp            (* if exp1 then exp2 else exp3 *)
   | . . .
type value =
    UnitVal
  | BoolVal of bool
  | IntVal of int
  | ...
type 'a env = (string * 'a) list

val empty_env : 'a env = []
val make_env : string -> 'a -> 'a env = <fun>
val lookup_env : 'a env -> string -> 'a option = <fun>
val sum_env : 'a env -> 'a env -> 'a env = <fun>
val ins_env : 'a env -> string -> 'a -> 'a env = <fun>
```

You must add comments indicating how the code you give relates to the specification given. The comments may be short, but they should be clear.

Main problem statement:
Extend `eval_exp (exp, m)` to handle `if_then_else_` constructs.

$$\frac{(e_1, m) \Downarrow \texttt{true} \qquad (e_2, m) \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v} \qquad \frac{(e_1, m) \Downarrow \texttt{false} \qquad (e_3, m) \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}$$

A sample test case.

```
# eval_exp (IfExp(ConstExp(BoolConst true),
                  ConstExp(IntConst 1),
                  ConstExp(IntConst 0)), []);;
- : Mp10common.value = IntVal 1
```

Extend `eval_exp (exp, m)` to handle `if` constructs. You may assume all other clauses for texttteval_exp have been given already.

```
let rec eval_exp (exp, m) =
(* Your code for the if_then_else_ case here.
   Use ... for the other cases *)
```

Problem 7. (18 points)

Give a polymorphic type derivation for the following type judgment:

```
{ } |- let f = fun x -> (x,x) in f(f 5) : (int * int) * (int * int)
```

Problem 8. (12 points)

Use the unification algorithm described in class and in MP7 to answer which of these given equations holds for the stated reason, and when it doesn't, to indicate why not. Capital letters (A, B, C) denote variables of unification. The lower-case letters (f, n, p) are constants or term constructors. (f and p have arity 2 - i.e., take 2 arguments, and n has arity 0 - i.e. it is a constant.) In this problem, we use $=$ as the separator for constraints. For each problem you are asked to check if the equation is correct for the stated reason(s), and if not, briefly why not. For a reason why not, you may underline a portion and write in the given space what it should have been if this will yield an approrpiate reason.

$$\{f(A, C) = f(A, A);\ p(B, n) = p(A, B)\}$$

(a) (2 points)

$$\begin{array}{rl} & \mathsf{Unify}\{\quad f(A, C) = f(A, A);\quad p(B, n) = p(A, B)\quad\} \\ = & \mathsf{Unify}\{\quad (A, C) = (A, A);\quad p(B, n) = p(A, B)\quad\} \\ & \text{by Decompose with } f(A, C) = f(A, A) \end{array}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(a) _____

(b) (2 points)

$$\begin{array}{rl} & \mathsf{Unify}\{\quad A = A;\quad C = A;\quad p(B, n) = p(A, B)\quad\} \\ = & \mathsf{Unify}\{\quad\quad\quad\quad C = A;\quad p(B, n) = p(A, B)\quad\} \\ & \text{by Delete with } A = A \end{array}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(b) _____

(c) (2 points)

$$\begin{array}{rl} & \mathsf{Unify}\{\ \ C = A;\ \ p(B,\, n) = p(A,\, B)\ \ \} \\ = & \mathsf{Unify}\{\ \ C = A;\ \ \ \ B = A;\ \ \mathrm{B} = \mathrm{n}\ \ \ \} \\ & \text{by Decompose with } p(B,\, n) = p(A,\, B) \end{array}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(c) _____

(d) (2 points)

$$\begin{array}{rl} & \mathsf{Unify}\{\ \ C = A;\ \ B = A;\ \ \mathrm{B} = \mathrm{n}\ \ \} \\ = & \mathsf{Unify}\{\ \ C = A;\ \ \ \ \ \ \ \ \mathrm{A} = \mathrm{n}\ \ \} \circ \{B \mapsto A\} \\ & \text{by Eliminate with } B = A \end{array}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(d) _____

(e) (2 points)

$$\begin{array}{rl} & \mathsf{Unify}\{\ \ C = A;\ \ \mathrm{A} = \mathrm{n}\ \ \} \circ \{B \mapsto A\} \\ = & \mathsf{Unify}\{\ \ C = n;\ \ \ \ \ \ \ \ \} \circ \{A \mapsto n; B \mapsto A\} \\ & \text{by Eliminate with } A = n \end{array}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(e) _____

(f) (2 points)

$$\begin{aligned} & \mathsf{Unify}\{ \quad C = n; \quad \} \circ \{A \mapsto n; B \mapsto A\} \\ = \; & \mathsf{Unify}\{ \qquad\qquad \} \circ \{A \mapsto n; B \mapsto A; C \mapsto n\} \\ & \text{by Eliminate with } C = n \end{aligned}$$

Select one:

- ◯ The step is correct for the stated reason
- ◯ The right-hand side does not equal the left-hand side for the stated reason because

(f) _____

Problem 9. (14 points)

Consider the following grammar over the alphabet $\{\&,\ \texttt{++},\ x,\ y,\ z,\ (,\ )\}$:

$$\begin{array}{lll} < \texttt{exp} > & ::= & < \texttt{var} > \ | \ \& < \texttt{exp} > | \ < \texttt{exp} > \texttt{++} \ | \ (< \texttt{exp} >) \\ < \texttt{var} > & ::= & \texttt{x} \ | \ \texttt{y} \ | \ \texttt{z} \end{array}$$

(a) (2 points) Show that the above grammar is ambiguous by the definition of ambiguous.

(b) (9 points) Write a new grammar accepting the same language that is unambiguous, and such that the postfix < `exp` > `++` has higher precedence than the prefix `&` < `exp` >.

(c) (3 points) Give the parse tree for `&x++` using the grammar you gave in the previous part of this problem.

Problem 10. (12 points)

Put a Y in each box where the relation labeling the column holds between the lambda term on the left and the lambda term on the right, and an N if it fails to hold

Lambda Term 1	α-conv	α-equiv	$\alpha\beta$-equiv	Lambda Term 2
$\lambda x.\ \lambda y.\ x\, y\, x$				$\lambda s.\ (\lambda t.\ s\, t)\, s$
$\lambda u.\ (\lambda v.\ v\, (u\, v))$				$\lambda g.\ (\lambda v.\ v\, (g\, v))$
$\lambda x.\ \lambda y.\ x\, y\, x$				$\lambda y.\ \lambda x.\ y\, x\, y$
$\lambda y.\ (\lambda z.\ (\lambda x.\ x\, x)(\lambda u.\ u)\, z)\, y$				$\lambda x.\ x$

Problem 11. (12 points)

Reduce the following term as much as possible using eager evaluation. Label each step of reduction with the rule justifying it. You do not need to label uses of congruence, or break them out as separate steps, in this problem.

$$(\lambda x.x(\lambda y.xy))((\lambda u.u)(\lambda w.w))$$

Workspace

Problem 12. (12 points)

Below are a series of partial Floyd-Hoare Logic proofs of program partial correctness assertions. For each one select whether all steps of the formal proof present are correct, and if not, circle at least one error with the derivation. There may be more than one error, but you only need to circle one. Program assertions $\{P\}C\{Q\}$ will sometimes be typeset as $\begin{array}{c}\{P\}\\ C\\ \{Q\}\end{array}$ to facilitate fitting the derivations on the page.

(a) (4 points)

$$\dfrac{\begin{array}{cc}\{\texttt{x + y > 0 and x < 5}\} & \{\texttt{x + y > 0 and (not x < 5)}\}\\ \texttt{x := (x * x) + 5} & \texttt{y := x + y}\\ \{\texttt{x + y > 0 and x < 5}\} & \{\texttt{x + y > 0 and (not x < 5)}\}\end{array}}{\begin{array}{c}\{\texttt{x + y > 0}\}\\ \texttt{if x < 5 then x := (x * x) + 5 else y := x + y}\\ \{\texttt{x + y > 0}\}\end{array}}\ \text{If\_then\_else Rule}$$

Select one:

- ◯ The proof is correct, as much as is shown
- ◯ The proof is incorrect and I have circled an error

(b) (4 points)

$$\dfrac{\begin{array}{c}\{\texttt{x + y > 0 and x < 5}\}\\ \texttt{x := (x * x) + 5}\\ \{\texttt{x + y > 0}\}\end{array}\qquad (\texttt{x + y > 0 and x < 5}) \Longrightarrow (\texttt{x + y > 0})}{\begin{array}{c}\{\texttt{x + y > 0 and x < 5}\}\\ \texttt{x := (x * x) + 5}\\ \{\texttt{x + y > 0 and x < 5}\}\end{array}}\ \text{Postcondition Weakening}$$

Select one:

- ◯ The proof is correct, as much as is shown
- ◯ The proof is incorrect and I have circled an error

(c) (4 points)

$$\dfrac{}{\{\texttt{((x * x) + 5) + y > 0}\}\texttt{x := (x * x) + 5}\{\texttt{x + y > 0}\}}\ \text{Assignment Axiom}$$

Select one:

- ◯ The proof is correct, as much as is shown
- ◯ The proof is incorrect and I have circled an error

Workspace

Extra Credit

13. Consider the following basic imperative programming language, where the commands are given by:

$$C ::= I\texttt{:=}E \mid \texttt{skip} \mid C\ \texttt{;}\ C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi} \mid \texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}\ C$$

where I ranges over program identifiers, E ranges over program arithmetic expressions, and B ranges over boolean-valued expressions. This language is a subset of the one we have used as an example for semantics in class and should be assumed to have the same rules. You may assume that the evaluation of boolean and arithmetic expressions does not alter the program memory.

Starting from this as our base, we wish to expand our language with a new construct in the syntactic category C of commands:

$$\texttt{for(}I\texttt{,}\ N_1\texttt{,}\ N_2\texttt{)}\ \texttt{do}\ C\ \texttt{od}$$

Informally, the semantics of this construct checks that N_2 (where N_1 and N_2 are integer constants) is positive. If N_2 is not positive, then the evaluation of this construct is complete with no alteration of the program memory. If N_2 is positive, then we assign I the value N_1, evaluate C and then repeat with $\texttt{for(}I\texttt{,}\ M_1\texttt{,}\ M_2\texttt{)}\ \texttt{do}\ C\ \texttt{od}$ where $M_1 = N_1 + 1$ and $M_2 = N_2 - 1$. I is a variable that is in scope both during and after the execution of $\texttt{for(}I\texttt{,}\ N_1\texttt{,}\ N_2\texttt{)}\ \texttt{do}\ C\ \texttt{od}$.

(a) (5 points) Give a complete set of Natural (big step, Structural Operational) Semantics rule(s) for
`for(`I`,` N_1`,` N_2`) do` C `od`.

(b) (5 points) Give a complete set of Transition (little step) Semantics rule(s) for
`for(`I`,` N_1`,` N_2`) do` C `od`

14. (10 points (bonus)) Give a proof in Floyd-Hoare Logic of the following Hoare triple:

$\{\texttt{k} \geq 0\}$
`p:= 0; s:= 0; t:= 1; while p < k do s:= s + t; t:= 2 * t; p:= p + 1 od`
$\{\texttt{s} = 2^{\texttt{k}} - 1\}$

Workspace

A Polymoprhic Typing Rules

Polymorphic constant signatures:

$\mathsf{sig}(n) = \texttt{int}$ n an integer constant $\quad \mathsf{sig}(\oplus) = \texttt{int} \to \texttt{int} \to \texttt{int}$ for $\oplus \in \{+, -, *, \% \ldots\}$
$\mathsf{sig}(\texttt{true}) = \texttt{bool} \quad \mathsf{sig}(\texttt{false}) = \texttt{bool}$
$\mathsf{sig}(\sim) = \forall\alpha.\, \alpha \to \alpha \to \texttt{bool}$ for $\sim \in \{<, >, =, \leq, \geq\}$
$\mathsf{sig}(\texttt{[ ]}) = \forall\alpha.\, \alpha\texttt{list} \quad \mathsf{sig}((::)) = \forall\alpha.\, \alpha \to \alpha\ \texttt{list} \to \alpha\ \texttt{list}$
$\mathsf{sig}((\ ,\ )) = \forall\alpha\beta.\, \alpha \to \beta \to \alpha * \beta$

Constants:

$$\frac{}{\Gamma \vdash\ c\ :\ \tau'}\ \textsc{Const}$$

where c is a constant listed above, $\mathsf{sig}(c) = \forall\alpha_1 \ldots \alpha_n.\, \tau$ and there exist $\sigma_1, \ldots, \sigma_n$ such that $\tau' = \tau[\sigma_1/\alpha_1; \ldots; \sigma_n/\alpha_n]$

Variables:

$$\frac{}{\Gamma \vdash x : \tau'}\ \textsc{Var}$$

where $\forall\alpha_1 \ldots \alpha_n.\, \tau = \Gamma(x)$ and there exist $\sigma_1, \ldots, \sigma_n$ such that $\tau' = \tau[\sigma_1/\alpha_1; \ldots; \sigma_n/\alpha_n]$

Connectives:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ \&\&\ e_2 : \texttt{bool}}\ \textsc{Conn} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ ||\ e_2 : \texttt{bool}}\ \textsc{Conn}$$

If_then_else rule:

$$\frac{\Gamma \vdash e_c : \texttt{bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_e : \tau}{\Gamma \vdash \texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_e : \tau}\ \textsc{If}$$

Application rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{App}$$

Function rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}\ \textsc{Fun}$$

Let rule:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad [x : \mathsf{Gen}(\tau_1, \Gamma)] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{Let}$$

Let Rec rule:

$$\frac{[x : \tau_1] + \Gamma \vdash e_1 : \tau_1 \qquad [x : \mathsf{Gen}(\tau_1, \Gamma)] + \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau_2}\ \textsc{Rec}$$

B Floyd-Hoare Logic

Simple Imperative Programming Language:

$$C ::= I\texttt{:=}E \mid C\ \texttt{;}\ C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C\ \texttt{fi} \mid \texttt{while}\ B\ \texttt{od}\ C\ \texttt{od}\ C$$

where I ranges over program identifiers, E ranges over program arithmetic expressions, and B ranges boolean-valued expressions.

Rules:

Assignment Axiom:

$$\frac{}{\{P[e/x]\}; x\texttt{:=}e\{P\}}$$

Sequencing Rule:

$$\frac{\{P\}C_1\{Q\} \quad \{Q\}C_2\{R\}}{\{P\}C_1\ \texttt{;}\ C_2\{R\}}$$

If_then_else Rule:

$$\frac{\{P \text{ and } B\}C_1\{Q\} \quad \{P \text{ and (not } B)\}C_2\{Q\}}{\{P\}\texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}\{Q\}}$$

While Rule:

$$\frac{\{P \text{ and } B\}C\{Q\}}{\{P\}\texttt{while}\ B\ \texttt{do}\ C\ \texttt{od}\{P \text{ and (not } B)\}}$$

Precondition Strengthening:

$$\frac{P \Longrightarrow P' \quad \{P'\}C\{Q\}}{\{P\}C\{Q\}}$$

Postcondition Weakening

$$\frac{\{P\}C\{Q'\} \quad Q' \Longrightarrow Q}{\{P\}C\{Q\}}$$