CS242 Final
Sample Solution
Fall 2021

- Please read all instructions (including these) carefully.
- There are 6 questions on the exam, some with multiple parts. You have 180 minutes to work on the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: ____

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

SIGNATURE: ____

Problem	Max points	Points
1	20	
2	20	
3	20	
4	20	
5	20	
6	20	
TOTAL	120	

1. **More Pi** (20 points)

   Recall the syntax of the π-calculus from lecture 9:

$$\begin{array}{llll} \text{Prefixes} & p & ::= & a(x) \\ & & \mid & \bar{a}x \\ \text{Agents} & P & ::= & 0 \\ & & \mid & p.\,P \\ & & \mid & P + P \\ & & \mid & P \mid P \\ & & \mid & x = y \Rightarrow P \\ & & \mid & x \neq y \Rightarrow P \\ & & \mid & \nu x\, P \\ & & \mid & !P \end{array}$$

   Assume that there is a process that writes a non-negative integer i, encoded as a sequence of i `succ` messages followed by one `zero` message, on the channel c. Write a process that reads from c and writes the number **floor**$(i/2)$ (encoded as a series of `succ` messages followed by a `zero` message) on channel d. You may assume there is only one process that reads from d. Do not use integer division, or any other operations not in the grammar given above, in your solution.

   Solution:

$$\begin{array}{l} \nu\texttt{lock}\ \nu s \\ \quad (!\texttt{lock}(s).\, c(v_1). \\ \qquad (v_1 = \texttt{zero} \Rightarrow \bar{d}\,\texttt{zero}.\, 0 + \\ \qquad\ v_1 = \texttt{succ} \Rightarrow \\ \qquad\qquad c(v_2). \\ \qquad\qquad (v_2 = \texttt{zero} \Rightarrow \bar{d}\,\texttt{zero}. + \\ \qquad\qquad\ v_2 = \texttt{succ} \Rightarrow \bar{d}\,\texttt{succ}.\, \overline{\texttt{lock}}\, s.\, 0)) \mid \\ \quad \overline{\texttt{lock}}\, s.\, 0) \end{array}$$

2. **Loop Invariants** (20 points)

Consider the following program with the usual semantics:

```
// assume $n \geq 0$
$j := 0$
$i := 1$
// (1)
while $i \leq n$ do
// (2)
  $j := j + i$
  $i := i + 1$
end
// (3)
assert($j = 1 + \cdots + n$)
```

Write a loop invariant for this loop and show it satisfies the following three properties: the invariant holds initially at line (1), the invariant holds on each execution of the loop body at line (2), and when the loop terminates, the invariant at line (3) together with the negation of the `while` predicate implies the assertion.

Solution: Invariant:

$$j = \sum_{1 \leq i' < i} i' \quad \wedge \quad i \leq n + 1.$$

(a) Initially,

$$0 = \sum_{1 \leq i' < 1} i' \quad \wedge \quad 1 \leq n + 1.$$

(b) Assume $j = \sum_{1 \leq i' < i} i'$ at (2). We also know that $i \leq n$ because the loop predicate must be true to reach (2). After executing $j_0 := j + i$ and $i_0 := i + 1$ we have

$$j_0 = \sum_{1 \leq i' < i} i' + i = \sum_{1 \leq i' < i_0} i' \quad \text{and} \quad i_0 \leq n + 1.$$

(c) Assume $j = \sum_{1 \leq i' < i} i'$, $i \leq n + 1$, and $i > n$. Therefore $i = n + 1$ and it follows that

$$j = \sum_{1 \leq i' < n+1} i'.$$

3. **References** (20 points)

As discussed in lecture, the Simply Typed Lambda Calculus (STLC) is not Turing-complete, as simple types do not by themselves allow for the definition of recursive functions. In this problem, we'll show that adding state to the STLC increases the computational power of the STLC by enabling recursion.

We consider the STLC extended with references and integers:

$$e ::= x \mid \lambda x\colon \tau.\, e \mid e\ e \mid i \mid \&e \mid !e \mid e := e$$

and the types:

$$\tau ::= \mathsf{int} \mid \alpha \mid \tau \to \tau \mid \mathsf{ref}\ \tau$$

where the type $\mathsf{ref}\ \tau$ is the type of references (pointers) to τ.

Note that this language is slightly different from the untyped language we considered in lecture 11. In particular, the new expression has been replaced by the $\&e$ expression, which evaluates to the address of the value of e (assume that every value has such an address if needed). The value of an assignment $e_1 ::= e_2$ is the value of e_2.

(a) **Types.**

Give type rules (not evaluation rules!) for the $\&e$ and $!e$ expressions.

Solution:

$$\frac{A \vdash e : t}{A \vdash \&e : \mathsf{ref}\ t} \qquad \frac{A \vdash e : \mathsf{ref}\ t}{A \vdash\ !e : t}$$

(b) **Backpatching** is a programming technique where a forward reference is created and then filled in with its value later. Backpatching can be used to create cyclic data structures and recursive functions. The following example creates a non-terminating function in the simply typed lambda calculus with references:

// The initial value of r doesn't matter,
// except that it must have the correct type
let $r = \&\lambda x\colon \mathsf{int}.\, x$ in
let $f = \lambda x.\, (!r)\ x$ in
let $z = r := \&f$ in
$f\ 0$

This program first defines a function f that calls the function pointed to by r (the line "let $f = \ldots$"). Then the reference r is updated so that the function it points to is f (the line "let $z = \ldots$"). Finally, f is called, but all it does is call $!r$, which is f, and so it goes into an infinite loop.

Use backpatching to write a recursive function that computes $n!$ (i.e., the factorial of a non-negative integer n). You may assume that let $a = e$ in e, integer constants, $e + e$, $e - e$, $e * e$, $e == e$, and if $e\ e\ e$ are all available as primitives.

Your program should type check under the rules of the simply typed lambda calculus. Recall that let $x = e_1$ in e_2 is syntactic sugar for $(\lambda x.\, e_2)\ e_1$. Assume the following types for the other primitives:

$$
\begin{aligned}
+, -, * &: \mathsf{int} \to \mathsf{int} \to \mathsf{int} \\
== &: \mathsf{int} \to \mathsf{int} \to \mathsf{bool} \\
\text{if} &: \mathsf{bool} \to \tau \to \tau \to \tau
\end{aligned}
$$

Solution:

let $r = \&\lambda n\colon \mathsf{int}.\, n$ in
let $f = \lambda n\colon \mathsf{int}.\, \text{if } (x == 0)\ 1\ ((!r\ (n-1)) * n)$ in
let $z = r := \&f$ in
f

4. **Set Constraints** (20 points)

In lecture 15 we developed a set constraint-based analysis for the lambda calculus that computed which syntactic lambdas an expression could evaluate to during program execution. The lambda calculus we considered had the following constructs:

$$e ::= x \mid \lambda x.\, e \mid e\ e \mid i$$

The analysis $\mathcal{L}(e)$ on these syntactic constructs was defined recursively as follows:

$$\begin{aligned}
\lambda x.\, e &\quad\Rightarrow\quad \lambda x.\, e \in \mathcal{L}(\lambda x.\, e)\\
i &\quad\Rightarrow\quad i \in \mathcal{L}(i)\\
e_1\ e_2 &\quad\Rightarrow\quad \forall \lambda x.\, e \in \mathcal{L}(e_1)\colon (\mathcal{L}(e_2) \subseteq \mathcal{L}(x) \wedge \mathcal{L}(e) \subseteq \mathcal{L}(e_1\ e_2))
\end{aligned}$$

(a) **Extending the analysis.**

Suppose we extend the language definition to contain several new features, including let expressions, if expressions, booleans, and sequencing. Concretely, consider the lambda calculus's syntax extended with the following constructs:

$$e ::= \ \ldots \mid b \mid \mathsf{if}\ e\ e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid e; e$$

Extend the analysis to support these features on an expression.

$b \quad \Rightarrow$ ________________________

$\mathsf{if}\ e_1\ e_2\ e_3 \quad \Rightarrow$ ________________________

$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \quad \Rightarrow$ ________________________

$e_1; e_2 \quad \Rightarrow$ ________________________

Solution: One correct solution is:

$$\begin{aligned}
b &\quad\Rightarrow\quad b \in \mathcal{L}(b)\\
\mathsf{if}\ e_1\ e_2\ e_3 &\quad\Rightarrow\quad \mathcal{L}(e_2) \subseteq \mathcal{L}(\mathsf{if}\ e_1\ e_2\ e_3) \wedge \mathcal{L}(e_3) \subseteq \mathcal{L}(\mathsf{if}\ e_1\ e_2\ e_3)\\
\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 &\quad\Rightarrow\quad \mathcal{L}(e_1) \subseteq \mathcal{L}(x) \wedge \mathcal{L}(e_2) \subseteq \mathcal{L}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)\\
e_1; e_2 &\quad\Rightarrow\quad \mathcal{L}(e_2) \subseteq \mathcal{L}(e_1; e_2)
\end{aligned}$$

(b) **Applications.**

Formulating analyses as set constraints allows for the implementation of many different (and useful) analyses within the same logical framework. One such analysis is *constant propagation,* which discovers and eliminates computations that have arguments known to be constants at compile-time. For example, a constant propagation pass could rewrite the expression $\mathsf{let}\ x = 5\ \mathsf{in}\ x + 1$ to just 6 by realizing that x can only take the value 5. In 1–3 sentences describe how the analysis pass in part (a) can be used to write a constant propagation pass.

Solution: For any program variable x, if $\mathcal{L}(x)$ only contains one value, a constant c, the uses of x can be replaced with c.

5. **Gradual Types** (20 points)

The following is the syntax for a version of lambda calculus with gradual types, as in lecture 13, augmented with integers:

$$e ::= x \mid \lambda x{:}\,\tau.\, e \mid e\ e \mid i$$
$$\tau ::= ? \mid \tau \rightarrow \tau \mid \textsf{int}$$

The type rules are given in Lecture 13, slide 27. To these rules we add integer constants, which have type int.

(a) Write an expression that typechecks correctly, but fails with a type error at runtime. Explain in 1–3 sentences why your example is type-correct and why it causes a runtime error.

Solution: One such expression is: $(\lambda x{:}\,?.\, x\ 2)\ 1$.

Inside the function definition, x has a dynamic type, so it is legal to treat is as anything, such as a function. Furthermore, again since x has a dynamic type, we can pass an argument of any type (such as integer) to it. As a result, we can treat an integer (1) as a function and call it, resulting in a runtime type error.

(b) In the typechecking rules for gradual typing, recall that we defined a consistency relation (written as $t_1 \sim t_2$). This consistency relation is used in the following typing rule (from lecture 13):

$$\frac{A \vdash e_1 : t_1 \to t_3 \quad A \vdash e_2 : t_2 \quad t_1 \sim t_2}{A \vdash e_1\ e_2 : t_3}$$

The consistency relation is defined with the following rules (from lecture 13):

i. $t \sim t$

ii. $? \sim t$

iii. $t \sim ?$

iv. $(t_1 \to t_2) \sim (t_3 \to t_4)$ if and only if $t_1 \sim t_3$ and $t_2 \sim t_4$

Write an expression that type checks correctly, but would not type check without rule iv. Explain your solution in 1–3 sentences.

Solution: One such expression is: $(\lambda y\colon (? \to ?).\, y\ 2)\ (\lambda x\colon \mathsf{int}.\, x + 1)$

This expression is of the form $e_1\ e_2$ where e_1 has type $(? \to ?) \to ?$ and e_2 has type $\mathsf{int} \to \mathsf{int}$. Typechecking such an expression requires checking that $(? \to ?) \sim (\mathsf{int} \to \mathsf{int})$, which requires rule iv.

6. **Monads** (20 points)

In this problem, we use monads from lecture 16 to keep track of the balance stored in a bank account. This balance is represented by the following data type:

$$\begin{array}{l} \text{data } \textbf{Balance } a = \\ \quad \text{Dollars } a \\ \mid \text{Bankrupt} \end{array}$$

For example, "Dollars 12" represents a bank balance of \$12. If a client's balance dips below \$0, they are considered "Bankrupt" and no further actions can be performed on the bank account.

When implementing the following functions, you can make use of the lambda calculus, arithmetic operators, **if** $e\ e\ e$ and a case expression that pattern matches on the constructor of its argument:

```
case $x$ of
    Dollars($y$) $\rightarrow e$  // evaluated if $x$ = Dollars $y$, variable $y$ is bound in $e$
    Bankrupt  $\rightarrow e'$ // evaluated if $x$ = Bankrupt
```

For example, the following use of case returns the balance as an integer or 0 if x is "Bankrupt":

```
case $x$ of
    Dollars($z$) $\rightarrow z$
    Bankrupt  $\rightarrow 0$
```

(a) Show that **Balance** can be implemented using a monad. That is, implement two functions: `return` and $\gg=$ (pronounced bind). Recall from lecture the types of these functions for a monad M: `return`: $a \rightarrow M\ a$ and `bind`: $M\ a \rightarrow (a \rightarrow M\ b) \rightarrow M\ b$.

Solutions:

```
monad M = Balance
    return  = Dollars
    $b \gg= f$ = case $b$ of
                   Dollars $d$  $\rightarrow f\ d$
                   Bankrupt $\rightarrow$ Bankrupt
```

(b) Next, implement a function `deposit` : `int` $\to$ `int` $\to$ Balance `int`, which takes two arguments representing the number of dollars to deposit and the existing dollar value of a bank account, in that order. The function returns a **Balance** value that represents the state of the account after making the deposit. *Note that deposits can be negative.* A negative deposit signifies a withdrawal. If a (negative) deposit would cause the account value to dip below 0, return Bankrupt.

Solution:

```
deposit = $\lambda x.\, \lambda y.$
    if $x + y < 0$
    then Bankrupt
    else Dollars $(x + y)$
```

(c) Use `return`, $\gg\!=$, and `deposit` to write a function that takes an initial account value n and three deposit values x, y, and z and returns the **Balance** of the account after making the three deposits. Note that if, at any point, the balance dips below 0, the function should return Bankrupt and any remaining operations have no effect on the account.

Solution:

```
$\lambda n.\, \lambda x.\, \lambda y.\, \lambda z.$
    return $n \gg\!=$
    deposit $x \gg\!=$
    deposit $y \gg\!=$
    deposit $z$
```