CS242 Final
Fall 2024

- Please read all instructions (including these) carefully.
- There are 4 questions on the exam, all with multiple parts. You have 3 hours 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.
- Several questions ask you to write programs in one of the languages covered in the course (e.g., Rust, Haskell, and Agda). For these problems, grading will not be strict about the syntax—if we can understand your solution, we will ignore syntax errors. However, if your answer is ambiguous or we cannot decipher it because of confusion about syntax, we will grade it accordingly, so striving to write clear solutions as much within the language syntax as you can will be beneficial.
- 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: ______________________________

STUDENT ID: ______________________________

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	30	
2	26	
3	30	
4	24	
TOTAL	110	

1. **Types and Continuations** (30 points)

For each of the following problems, write a lambda term for which the given type is the most general type possible (i.e., the type that would be assigned by type inference). You may use integer constants (of type `Int`) and addition (of type `Int` $\rightarrow$ `Int` $\rightarrow$ `Int`) in your answers. Do not use types in your lambda terms — lambda abstractions should have the form $\lambda x.e$, not $\lambda x : t.e$.

a. (1 point) $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$

$\lambda x.\lambda f.f\ x$

b. (1 point) $\alpha \rightarrow \beta \rightarrow$ `Int`

$\lambda x.\lambda y.0$

c. (2 points) $\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow$ `Int`

$\lambda x.\lambda f.(\lambda z.0)\ (f\ x)$

For parts d and e, recall the definition of the CPS Monad from lecture 9, slide 27:

$$\texttt{Cont r a} = (\texttt{a} \rightarrow \texttt{r}) \rightarrow \texttt{r}$$

d. (2 points) `Cont r Int`

$\lambda k.k\ 0$

e. (5 points) `Int` $\rightarrow$ `Cont Int a`

$\lambda i.\lambda k.(\lambda z.0)\ (\lambda f.\lambda g.f\ (g\ (\lambda a.1 + i))\ (g\ k))$

f. (6 points) For remaining subproblems we will use Haskell syntax. You may use the following Haskell definitions:

```
newtype Cont r a = Cont ((a -> r) -> r)

extract :: Cont r a -> ((a -> r) -> r)
extract (Cont f) = f
```

It turns out that for every Applicative instance (recall lecture 13), there is an inverted Applicative instance whose `(<*>)` operator performs the side effects in the opposite order (i.e., right-to-left). Write Haskell code that implements an inverted Applicative instance for `Cont` below:

```
instance Applicative (Cont r) where
    pure :: a -> Cont r a
    pure x =
```

```
Cont (\k -> k x)
```

```
    (<*>) :: Cont r (a -> b) -> Cont r a -> Cont r b
    f <*> x =
```

```
let f' = extract f
x' = extract x
in Cont $ \k -> x' (\a -> f' (\atob -> k (atob a)))
```

Here are some additional Haskell definitions for subproblems g, h, and i:

```
double :: Int -> Int
double x = 2 * x

inc :: Int -> Int
inc x = x + 1

g :: Cont Int (Int -> Int)
g = Cont (\k -> inc (k inc))

a :: Cont Int Int
a = Cont $ \k -> k (double (k 3))

b :: Cont Int Int
b = g <*> a

runCont :: Cont r r -> r
runCont c = (extract c) (\x -> x)

f5 :: Int
f5 = runCont b
```

g. (3 points) Evaluate the above Haskell program using the standard Applicative instance for `Cont`:

`f5 =` $\boxed{10}$

h. (3 points) Evaluate the above Haskell program using the inverted Applicative instance for `Cont`:

`f5 =` $\boxed{12}$

i. (7 points) Consider the following code:

```
f4' [] ys k = return ys
f4' xs [] k = return xs
f4' (x:xr) (y:yr) k
  = if (x < y)
      then do rest <- f4' xr (y:yr) k
              return (x:rest)
      else if (y < x)
        then do rest <- f4' (x:xr) yr k
                return (y:rest)
        else (k [])

f4 :: [Int]
   -> ([Int] -> Cont r [Int])
   -> Cont r [Int]
f4 [] _ = return []
f4 (h:[]) k = if (h < 0)
                then (k [])
                else return [h]
f4 l k =
  let n = length l
      n' = n `div` 2
      (a, b) = splitAt n' l
   in do a' <- f4 a k
         b' <- f4 b k
         f4' a' b' k

f44 :: [Int] -> [Int]
f44 xs = runCont (callCC (f4 xs))
```

Give a concise English description of the input-output functionality of this program.

`f44` performs a mergesort, with the exception that the empty list is returned if any of the elements are repeated or less than 0.

2. **Evaluation Order, Termination, and Confluence** (26 points)

You're the on-call engineer at the hot new startup ski.ai™, whose website allows anyone in the world to execute any SKI expression they choose. At 3am you're awakened by an emergency alert: requests are failing, the site is down, and AWS expenditures are through the roof! You rush to the logs and find the culprit: someone has entered the expression $(S\ I\ I)\ (S\ I\ I)$. Immediately you realize what's gone wrong: someone has entered an SKI expression that doesn't terminate, and now all of your servers are spending all of their cycles in an infinite loop. You call a meeting of your top engineers, but unfortunately they are asleep, and only Alex, Ben, and Colin are available. You explain the problem to them.

a. (4 points) Alex answers first: "I've got the fix!" he shouts, and begins drawing on the whiteboard. "Whenever our evaluator applies a reduction, if it yields the same AST we know we've hit an infinite loop!". On the whiteboard you see written:

```
def eval(ast):
  while not is_fully_evaluated(ski_expr):
    new_ski_expr = apply_single_reduction(ski_expr)
    if new_ski_expr == ski_expr:
      return ski_expr
    else:
      ski_expr = new_ski_expr
```

"With this deployed, we can be sure that any SKI expression will eventually terminate," Alex says. Thanks to that fantastic CS242 class you took back at Stanford, you're able to immediately **write an SKI expression that doesn't terminate under normal order evaluation despite Alex's fix**. You may include a brief justification for your answer for possible partial credit.

For all parts of this problem you may find it useful to refer to Lecture 4, slides 11-19. You can also use any combinator defined in the lecture notes in your answers (e.g., true, pair, c1, fac, Y, . . .).

(S I I) (S I I)
This non-terminating expression does not rewrite to itself in a single step, and so will not terminate with this definition of eval.

b. (6 points) “Exactly what I was thinking!” says Ben. “Here’s an actual fix:”

```
def eval(ski_expr):
   seen_before = set()
   while ski_expr not in seen_before:
     seen_before.add(ski_expr)
     ski_expr = apply_single_reduction(ski_expr)

     if is_fully_evaluated(ski_expr):
       return ski_expr
```

CS242 saves you again! You quickly spot the flaw and **write an SKI expression in the space below that doesn’t terminate under normal order evaluation despite Ben’s fix**. You may include a brief justification for your answer for possible partial credit.

(S I I) (S I I) also works as an answer to this problem: under normal order, the reduction sequence is
(S I I) (S I I) ->
I (S I I) (I (S I I)) ->
(S I I) (I (S I I)) ->
I (I (S I I)) (I (I (S I I))) ->
I (S I I) (I (I (S I I))) ->
(S I I) (I (I (S I I))) ->
...
Every time (S I I) comes into function position, it’s argument has grown in size by one additional application of I, so terms never repeat.

c. (8 points) Finally, Colin stands up and walks to the whiteboard. “I think I’ve got the solution,” he says, and writes the following on the board:

```
def eval(ski_expr, fuel: int):
  while not is_fully_evaluated(ski_expr) and fuel > 0:
    ski_expr = apply_single_reduction(ski_expr)
    fuel = fuel - 1

  return ski_expr
```

“This way we can only perform at most a finite number of reductions, so execution should always terminate,” he says. “You’re right about that,” you say, “but now the language is no longer confluent.” **Write an SKI expression, an initial fuel value, and two valid sequences of reductions given that fuel value that produce different results**. For example, if your SKI expression is `(I I) I` and your chosen fuel value is 2, then a valid reduction sequence would be `(I I) I` → `I I` → `I` because this sequence shows two steps of reduction of `(I I) I`.

i. SKI expression:

(K I S) (K I S)

ii. Fuel value (e.g., 2): 1

iii. Reduction sequence 1:

(K I S) (K I S) → I (K I S)

iv. Reduction sequence 2:

(K I S) (K I S) → (K I S) I

d. (8 points) “Not only that”, Alex says, “but now we can’t use normal order anymore, as there are programs for which normal order doesn’t finish evaluating but another evaluation order would.” He looks to you for help, and smiling, you **write an SKI expression, an initial `fuel` value, the reduction sequence for normal order, and the alternative reduction sequence that returns a fully evaluated result**. Formally, if your SKI expression is e and $e \to^{\texttt{fuel}} e'$ under normal order evaluation and $e \to^{\texttt{fuel}} e''$ via the alternative reduction sequence, e' should not be fully evaluated under normal order, while e'' should be the term that would result from full normal order evaluation.

i. SKI expression:

S I I (I I)

ii. Fuel value (e.g., 2):

5

iii. Normal order reduction sequence:

S I I (I I) $\to$ (I (I I)) (I (I I)) $\to$ (I I) (I (I I)) $\to$ I (I (I I)) $\to$ I (I I) $\to$ I I

iv. Alternative reduction sequence:

S I I (I I) $\to$ S I I I $\to$ (I I) (I I) $\to$ I (I I) $\to$ I I $\to$ I

3. **Rust** (30 points)

Consider the following Rust function (where "$\cdots$" is an unspecified, valid boolean expression):

```
fn swizzle<'a, 'b, 'c, 'd, 'e, 'f>(x: &'a str, y: &'b str, z: &'c str)
    -> (&'d str, &'e str, &'f str)
where ???
{
    (if ··· { x } else { y },
     if ··· { x } else { z },
     if ··· { y } else { z })
}
```

a. (6 points) In the box below, write the lifetime constraints that should be placed in the `where` clause to yield the most general signature for `swizzle` that passes the borrow checker. Recall that Rust considers lifetime `'a` to be a subtype of lifetime `'b` if lifetime `'a` is *longer* than lifetime `'b`—that is, the lifetime `'a` contains the lifetime `'b`—and that a lifetime constraint has the syntax `'a:'b`, where `'a` is a subtype of `'b`. You may also find it helpful to review the discussion of lifetimes in lecture 11, slides 30-34.

```
'a: 'd, 'b: 'd, 'a: 'e, 'c: 'e, 'b: 'f, 'c: 'f
```

Let $\mathtt{T}_1 <: \mathtt{T}_2$ denote that $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$. Recall Rust's subtyping rules for borrows and functions, as induced by the subtyping relation on lifetimes:

$$\begin{aligned} \mathtt{L} &\in \text{Lifetime} \\ \mathtt{T}, \mathtt{S} \in \text{Type} &::= \&\mathtt{'L\ T} \mid \&\mathbf{mut}\,\mathtt{'L\ T} \mid \mathtt{T}_1 \to \mathtt{T}_2 \mid \cdots \end{aligned}$$

$$\frac{}{\mathtt{T} <: \mathtt{T}} \qquad \frac{\mathtt{L}_1 <: \mathtt{L}_2 \quad \mathtt{T}_1 <: \mathtt{T}_2}{\&\mathtt{'L}_1\ \mathtt{T}_1 <: \&\mathtt{'L}_2\ \mathtt{T}_2} \qquad \frac{\mathtt{L}_1 <: \mathtt{L}_2}{\&\mathbf{mut}\,\mathtt{'L}_1\ \mathtt{T} <: \&\mathbf{mut}\,\mathtt{'L}_2\ \mathtt{T}} \qquad \frac{\mathtt{S}_1 <: \mathtt{T}_1 \quad \mathtt{T}_2 <: \mathtt{S}_2}{\mathtt{T}_1 \to \mathtt{T}_2 <: \mathtt{S}_1 \to \mathtt{S}_2}$$

Subtyping is discussed in lecture 18, slides 37-50. Here is a brief review of some Rust features used in this problem:

- String literal expressions, e.g. `"hello"`, have type `&'static str`, where `'static` is the lifetime of the program (the longest lifetime).
- `String::from` creates a new, owned `String` object from a string literal, e.g. `String::from("hello")`. For all `'a`, `String::as_str` takes a `&'a String` and returns a `&'a str`, e.g. `s.as_str()`.
- A scope can be created anywhere, e.g. `{ e }`, and all values declared in that scope are dropped where it ends. A scope is an expression and it evaluates to the value of the final expression in the scope, as for function bodies.
- The syntax for a closure is `|args| body`. The trait that is satisfied by closures/functions mapping `Args` to `Ret` is `Fn(Args) -> Ret`.

b. (8 points) Suppose, counterfactually, that function types were contravariant in their range (the type of the return value). Write a `launder` function such that the Rust code below exhibits a use-after-free error. Both the type signature and the code for `launder` are important in your answer.

```
let hello = {
    let hello = String::from("hello");
    launder(|| hello.as_str())
};
println!("{hello}");
```

```
fn launder<F: Fn() -> &'static str>(f: F) -> &'static str {
    f()
}
```

c. (8 points) Suppose, counterfactually, that function types were covariant in their argument type. Write a `launder` function such that the Rust code below exhibits a use-after-free error. Both the type signature and the code for `launder` are important in your answer.

```
let hello = &mut "hello";
{
    let world = String::from("world");
    launder(|s| { *hello = s; }, world.as_str())
}
println!("{hello}");
```

```
fn launder<'a, F: Fn(&'a str)>(f: F, s: &'a str) {
    f(s)
}
```

d. (8 points) Suppose, counterfactually, that mutable borrows were covariant in type component `T` as well as the lifetimes:

$$\frac{L_1 <: L_2 \quad T_1 <: T_2}{\&\texttt{mut}\,{}'L_1\; T_1 <: \&\texttt{mut}\,{}'L_2\; T_2}$$

Write a `launder` function such that the Rust code below exhibits a use-after-free error. Both the type signature and the code for `launder` are important in your answer.

```
let mut hello: &'static str = "hello";
{
    let world = String::from("world");
    launder(&mut hello, world.as_str());
}
println!("{hello}");
```

```
fn launder<'a>(r: &mut &'a str, s: &'a str) {
    *r = s;
}
```

4. **Dependent Types** (24 points)

Assume the existence of the type `Nat` of natural numbers with constructors `zero` and `suc` as usual and (infix) addition `_+_ : Nat -> Nat -> Nat` and (infix) multiplication `_*_ : Nat -> Nat -> Nat`. In Agda, we can define `Vec`, the type of arrays indexed by their length, as follows:

```
data Vec (A : Set) : Nat -> Set where
    nil  : Vec A zero
    cons : (n : Nat) -> A -> Vec A n -> Vec A (suc n)
```

Fill in the Agda functions below according to their specifications using **only** `Nat`, `zero`, `suc`, `_+_`, `_*_`, `Vec`, `nil`, `cons`, and built-in Agda constructs. **Give a type signature and a definition for each function**.

a. (8 points) Define `append`. Given a `Vec` of values of type `A`, $[x_1, \ldots, x_n]$, and another `Vec` of values of type `A`, $[y_1, \ldots, y_m]$, `append` should return $[x_1, \ldots, x_n, y_1, \ldots, y_m]$.

```
append : (A : Set)
    -> (m : Nat) -> Vec A m
    -> (n : Nat) -> Vec A n -> Vec A (m + n)
append A zero nil n ys = ys
append A (suc m) (cons m x xs) n ys =
    cons (m + n) x (append A m xs n ys)
```

b. (8 points) Define `dot`, the pointwise product of two vectors. Given a `Vec` of natural numbers, $[x_1, \ldots, x_n]$, and another `Vec` of natural numbers of the same length, $[y_1, \ldots, y_n]$, `dot` should return $[x_1 * y_1, \ldots, x_n * y_n]$. An application of `dot` should not type-check if its inputs are vectors of unequal length.

```
dot : (n : Nat) -> Vec Nat n -> Vec Nat n -> Vec Nat n
dot zero nil nil = nil
dot (suc n) (cons n x xs) (cons n y ys) =
    cons n (x * y) (dot n xs ys)
```

c. (8 points) Define **first**. Given a non-empty `Vec` of values of type `A`, $[x_1, \ldots, x_n]$, `first` should return x_1. An application of `first` should not type-check if its input is the empty vector.

```
first : (A : Set) -> (n : Nat) -> Vec A (suc n) -> A
first A n (cons n x xs) = x
```