Midterm Exam

CSE 130: Programming Languages

July 16, 2019

This is a **80 minute** exam. This exam is **closed book**, but you may use a US letter double sided cheat sheet. The maximum possible score is **100 points**. There are four questions and one extra credit problem. We recommend reading through all the problems first! **Avoid getting stuck on any one problem!**

Make sure you print your name legibly and sign the honor code below. All of the intended answers may be written within the space provided. You may use the back of the preceding page for scratch work. **You may not use the back side of a page to write your answer.** If you need extra scratch paper, ask the proctors.

The answers provided in this exam are my own. By signing below I agree to UC San Diego's policy on integrity of scholarship and acknowledge that I have not received nor provided any help during the exam.

_____

Signature

_____ _____

Print your name *ID*

1 [20pts] True/False

For the true false questions below, circle or box one of the two options. If you would like to provide a defense of your argument, feel free to write an explanation next to each question (this is optional). (2 pts. each)

1. **TRUE** False Closures store environment information.
2. True **FALSE** Dynamically typed languages in general have better performance than a statically typed ones.
3. **TRUE** False Haskell treats functions as first class.
4. True **FALSE** The order of evaluation makes no difference to a Lambda Calculus expression.
5. True **FALSE** Haskell uses a strict type checking algorithm.
6. True **FALSE** Haskell evaluates expressions using Call-by-Value semantics.
7. **TRUE** False Javascript chases the access link to find free variables.
8. **TRUE** False $(\lambda x.x) + 3$ will cause a runtime type error.
9. **TRUE** False Javascript is type safe.
10. **TRUE** False Lambda Calculus is really cool. [1pt.]
11. True **FALSE** Javascript makes sense. [1pt.]

2 [20pts] Short Answers

For the short answers below, keep you explanations short and concise and to the limits of the bounded boxes.

1. [2pts] What is $FV(\lambda x.\lambda y.x\ y\ \lambda y.y\ z)$?

   Answer:

   $\{z\}$

2. [4pts] What is $(\lambda x.\lambda y.x\ y\ \lambda y.y\ z)[y := z]$? (Capture-avoiding substitution.)

   Answer:

   $(\lambda x.\lambda y.x\ y\ \lambda y.y\ z)$

3. [4pts] What is an advantage of call-by-name over call-by-value? What is an advantage of call-by-value over call-by-name?

   Answer:

   CBN adv. over CBV: Faster if you don't use an argument. Example: $(\lambda y.x)(3+2)$
   CBV adv. over CBN: Faster if you use an argument more than once because you only evaluate it a single time. Example: $(\lambda y.y+y+y)(3+2)$

4. [4pts] Why can't we use our λ-calculus equations (β, α, and η) to implement a simple interpreter for JavaScript? Give an example of a problematic JavaScript program.

   Answer:

   JavaScript functions can mutate variables, hence we cannot do substitution. TODO:give example.

5. [3pts] Explain how Haskell interpreter would interpret the following Haskell code:

   ```
   let f x = f x in length [1,2,f 3]
   ```

   differently from how the JavaScript interpreter would interpret the following JavaScript code:

   ```
   function f(x) { return f(x); }; [1,2,f(3)].length
   ```

   Answer:

   Haskell uses lazy evaluation, so the expression returns 3 since `f` is never evaluated. The JavaScript code does not terminate since JavaScript eagerly evaluates all elements.

6. [3pts] What's the difference between the access link and the control link?

Answer:

Control link is used to keep track of control flow. Access link is used to keep track of our variable-lookup environment.

3 [20pts] Lambda calculus

1. [4pts] Reduce the following lambda term to normal form (i.e. no further reductions possible) using **two** distinct reduction strategies.

$$(\lambda x.(\lambda y.(\lambda x.x\, y))\, x)\, z$$

Answer:

$$\begin{array}{ll} \rightarrow (\lambda y.(\lambda x.x\, y))\, z & (\beta \text{ reduction}) \\ \rightarrow (\lambda x.x\, z) & (\beta \text{ reduction}) \end{array}$$

or

$$\begin{array}{ll} \rightarrow (\lambda x.(\lambda y.(\lambda a.a\, y))\, x)\, z & (\alpha \text{ renaming}) \\ \rightarrow (\lambda x.(\lambda a.a\, x))\, z & (\beta \text{ reduction}) \\ \rightarrow (\lambda a.a\, z) & (\beta \text{ reduction}) \end{array}$$

Rubric: 2 points for each

2. [4pts] A *reduction path* from a λ-term M is a finite or infinite sequence $M \rightarrow_\beta M_1 \rightarrow_\beta M_2 \rightarrow_\beta \ldots$. There may be many reduction paths for a term.

   Write all possible reduction sequences for $(\lambda x.(\lambda y.x)\ z)\ y$, α-rename as you see fit.

Answer:

$(\lambda x.(\lambda y.x)\ z)\ y =_\alpha (\lambda x.(\lambda y_0.x)\ z)\ y \rightarrow_\beta (\lambda y_0.y)\ z \rightarrow_\beta y$
$(\lambda x.(\lambda y.x)\ z)\ y \rightarrow_\beta (\lambda x.x)\ y \rightarrow_\beta y$

Rubric: 2 points for each

3. [12pts] We call a term *weakly normalizing* if one of its paths terminates in a normal form and call a term *strongly normalizing* if all paths terminate in normal forms (i.e., there are no infinite reduction paths.) Terms that are neither weakly normalizing nor strongly normalizing are said to be *non-normalizing*. Are the following terms non-normalizing, weakly normalizing, or strongly normalizing? If the term is weakly-normalizing, show a finite reduction sequence. Please only perform *one* β-reduction per step.

   (a) $\mathbf{I}$, where $\mathbf{I} = \lambda x.x$

   Answer:

   Strongly normalizing

   (b) Ω or $\omega\ \omega$, where $\omega = \lambda x.x\ x$

   Answer:

   Non-normalizing

(c) $\mathbf{K}\ \mathbf{I}\ \Omega$, where $\mathbf{K} = \lambda x.\lambda y.x$

Answer:

Weakly-normalizing, $\mathbf{K}\ \mathbf{I}\ \Omega \rightarrow_\beta \mathbf{I}$

(d) $\mathbf{K}\ \mathbf{I}\ (\lambda x.\Omega)$

Answer:

Weakly-normalizing, $\mathbf{K}\ \mathbf{I}\ (\lambda x.\Omega) \rightarrow_\beta \mathbf{I}$ (*not* strongly-normalizing!)

(e) $(\lambda x.\mathbf{K}\ \mathbf{I}\ (x\ x))(\lambda y.\mathbf{K}\ \mathbf{I}\ (y\ y))$

Answer:

Weakly-normalizing, $(\lambda x.\mathbf{K}\ \mathbf{I}\ (x\ x))(\lambda y.\mathbf{K}\ \mathbf{I}\ (y\ y)) \rightarrow_\beta (\lambda x.\mathbf{I})(\lambda y.\mathbf{K}\ \mathbf{I}\ (y\ y)) \rightarrow_\beta \mathbf{I}$

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

Answer:

Non-normalizing (note, however, that this term is in *head-normal form)*

4 [20pts] Activation Records

Consider the following Javascript program.

```
 1: let f = () => {
 2:   let a = 5;
 3:   let g = (y) => {
 4:     if (y > 10) {
 5:         a = a-1;
 6:         return g(y/a);
 7:     } else {
 8:         return (y+a);
 9:     }
10:   };
11:   return g;
12: };
13: let h = f();
14: console.log(h(20));
```

1. [18pts] Fill in the missing parts in the following diagram of the run-time structures for the execution of this code up to the point immediately before `g` returns from the `if`-statement on line `6`. For each of the twelve empty blanks, write in either the value, or the number/letter of the appropriate activation record, closure or code snippet.

Activation Records | *Closures* | *Compiled Code*

(1)	top	access link	(0)
		f	F
		h	G
(2)	f()	access link	(1)
		a	4
		g	G
(3)	h(20)	access link	(2)
		y	20
(4)	g(5)	access link	(2)
		y	5

F: ⟨ (1) , code for f ⟩ | code for f |

G: ⟨ (2) , code for g ⟩ | code for g |

Rubric: 1.5 points for each fill-in

2. [2pts] What will be the result of executing the above program?

Answer:

It prints out 9

5 [20pts] Type Inference

In this problem you will apply the Hindley-Milner type inference algorithm to figure out the type for the `filter` function as declared in μHaskell. You must go through the five steps: creating the parse tree(s), assigning type variables, generating constraints, solving the constraints, and finally circling your final type answer (if no type error was encountered).

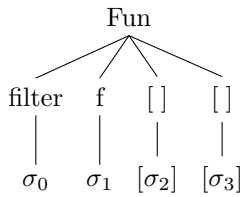
```
filter f []     = []
filter f (x:xs) = if f x
                    then x : (filter f xs)
                    else xs
```

[3pts] Let's first start with the first definition:

```
filter f [] = []
```

The parse tree is partially filled in for you. Complete the parse tree by assigning types to nodes, generating the constraints, and solve them.

Answer:



From this parse tree we have the type for `filter`:

(0) $\sigma_0 = \sigma_1 \to [\sigma_2] \to [\sigma_3]$

Rubric: -1 points for setting $\sigma_2 = \sigma_3$; -1 points for specializing σ_1 to a function type; -1 point for not giving list type for $[\sigma_2]$ and $[\sigma_3]$

[2pts] The second definition is more complicated. As a starting point, rewrite the definition to make applications explicit, i.e., in prefix form, so we can create the parse tree more easily. Please add all necessary parentheses to reflect the fact that functions (and thus function application) are single-argument. (*Hint:* For example, `g x y` is rewritten as `(g x) y`.).

Answer:

```
filter f []     = []
filter f (x:xs) = if f x
                    then ((:) x) ((filter f) xs)
                    else xs
```

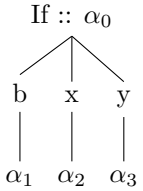
Rubric: -0.5 point for each set of missing parantheses.

[1pts] As a second step towards tackling the second definition, give the general type constraints for `if-then-else` expressions. Specifically, infer the type for:

```
if b then x else y
```

for some `b`, `x`, `y`. The parse tree has already been drawn for you.

Answer:

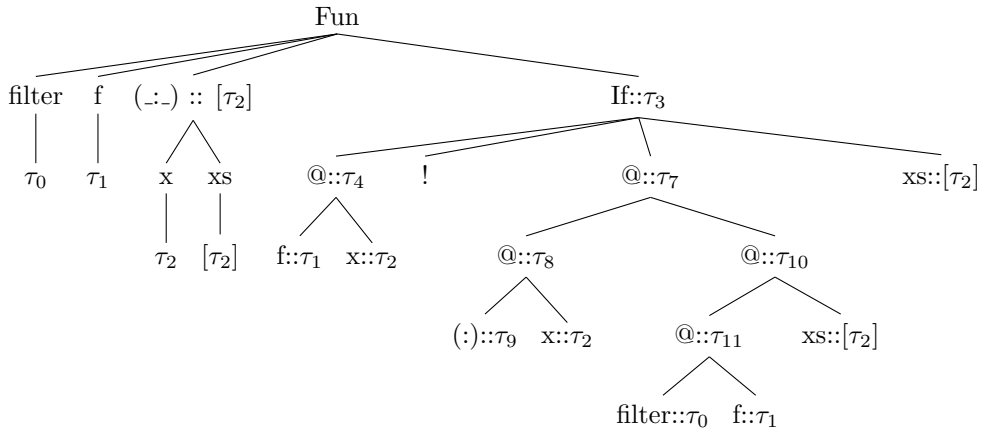


From this parse tree we have the following type constraints:

- $\alpha_0 = \alpha_2$
- $\alpha_1 = \texttt{Bool}$
- $\alpha_2 = \alpha_3$

[11pts] Armed with the above we can now infer the type of the second definition of `filter`. We partially filled in the tree on your behalf. When solving the constraints you may find the type of cons useful: $(:) :: \alpha \to ([\alpha] \to [\alpha])$ for any α. In the interst of time, you don't need to solve the equations iteratively as we did in class. You *may* skip steps, but you must show which equations you are unifying. (*Hint:* you may find it useful to guess what the final type of `filter` is before you get started.)

Answer:



Type constraint equations:

Fun node:

(1) $\tau_0 = \tau_1 \to [\tau_2] \to \tau_3$

If node:

(2) $\tau_3 = \tau_7$

(3) $\tau_7 = [\tau_2]$

(4) $\tau_4 = \texttt{Bool}$

@ node:

(5) $\tau_1 = \tau_2 \to \tau_4$

@ node:

(6) $\tau_8 = \tau_{10} \to \tau_7$

@ node:

(7) $\tau_9 = \tau_2 \to \tau_8$

From given type of cons:

(8) $\tau_9 = \tau_2 \to ([\tau_2] \to [\tau_2])$

@ node:

(9) $\tau_{11} = [\tau_2] \to \tau_{10}$

@ node:

(10) $\tau_0 = \tau_1 \to \tau_{11}$

Solving the constraints:

(1) $\tau_0 = \tau_1 \to ([\tau_2] \to \tau_3)$

(2) $\tau_3 = \tau_7$

(3) $\tau_7 = [\tau_2]$

(4) $\tau_4 = \texttt{Bool}$

(5) $\tau_1 = \tau_2 \to \tau_4$

(6) $\tau_8 = \tau_{10} \to \tau_7$

(7) $\tau_9 = \tau_2 \to \tau_8$

(8) $\tau_9 = \tau_2 \to ([\tau_2] \to [\tau_2])$

(9) $\tau_{11} = [\tau_2] \to \tau_{10}$

(10) $\tau_0 = \tau_1 \to \tau_{11}$

Unifying (1) and (10):

(11) $[\tau_2] \to \tau_3 = \tau_{11}$

Unifying (2) and (11):

(12) $[\tau_2] \to \tau_7 = \tau_{11}$

Unifying (3) and (6):

(13) $\tau_8 = \tau_{10} \to [\tau_2]$

Unifying (3) and (12):

(14) $[\tau_2] \to [\tau_2] = \tau_{11}$

Unifying (4) and ():

(15) $\tau_1 = \tau_2 \to \texttt{Bool}$

Unifying (5) and (10):

(16) $\tau_0 = (\tau_2 \to \tau_4) \to \tau_{11}$

Unifying (5) and (15):

(17) $\tau_4 = \texttt{Bool}$

Unifying (6) and (7):

(18) $\tau_9 = \tau_2 \to (\tau_{10} \to \tau_7)$

Unifying (6) and (13): produces (3)

Unifying (7) and (8):

(19) $\tau_8 = [\tau_2] \to [\tau_2]$

Unifying (7) and (18):

(20) $\tau_8 = \tau_{10} \to \tau_7$

Unifying (8) and (18):

(21) $[\tau_2] = \tau_{10}$

Unifying (9) and (10):

(22) $\tau_0 = \tau_1 \to ([\tau_2] \to \tau_{10})$

Unifying (9) and (11):

(23) $\tau_3 = \tau_{10}$

Unifying (9) and (12):

(24) $\tau_7 = \tau_{10}$

Unifying (9) and (14): produces (21)

Unifying (9) and (16):

(25) $\tau_0 = (\tau_2 \to \tau_4) \to ([\tau_2] \to \tau_{10})$

Unifying (10) and (16):

(26) $\tau_1 = \tau_2 \to \tau_4$

Unifying (10) and (22):

(27) $\tau_{11} = [\tau_2] \to \tau_{10}$

...

Unifying (17) and (25):

(28) $\tau_0 = (\tau_2 \to \texttt{Bool}) \to ([\tau_2] \to \tau_{10})$

Unifying (21) and (28):

(28) $\tau_0 = (\tau_2 \to \texttt{Bool}) \to ([\tau_2] \to [\tau_2])$

[3pts] Now that we inferred the type of `filter` for both definitions, derive the final type of `filter` by unifying the two. Show all the equations produced by unification. If you did not complete the type inference above, you may guess what the type inference algorithm would have produced and use the final inferred types for this part of the question.

Answer:

From above, we have:

$\sigma_0 = \sigma_1 \to ([\sigma_2] \to [\sigma_3])$

$\tau_0 = (\tau_2 \to \texttt{Bool}) \to ([\tau_2] \to [\tau_2])$

Unifying these, we have:

$\tau_0 = \sigma_0$

$\sigma_1 = \tau_2 \to \texttt{Bool}$

$\sigma_2 = \tau_2$

$\sigma_3 = \tau_2$

Hence, the type of `filter` is: $(\alpha \to \texttt{Bool}) \to [\alpha] \to [\alpha]$

Rubric: 1 point for each equation, including final

6 [10pts] Extra Credit. Y not.

Let $\mathbf{Y} = \lambda f.(\lambda x.f\ (x\ x))\ (\lambda y.f\ (y\ y))$ and P be any expression.

1. [7pts] Evaluate $\mathbf{Y}\ P$ and write the result only in terms of $\mathbf{Y}$ and P.
   (*Hint:* You will not reach a normal form. You will need to use η-reduction.)

$$
\begin{aligned}
& & \mathbf{Y}\ P \\
&= & (\lambda f.(\lambda x.f\ (x\ x))\ (\lambda y.f\ (y\ y)))\ P \\
&=_\beta & (\lambda x.P\ (x\ x))\ (\lambda y.P\ (y\ y)) \\
&=_\beta & P\ ((\lambda y.P\ (y\ y))\ (\lambda y.P\ (y\ y))) \\
&=_\eta & P\ ((\lambda f.(\lambda y.f\ (y\ y))\ (\lambda y.f\ (y\ y)))\ P) \\
&= & P\ (\mathbf{Y}\ P)
\end{aligned}
$$

   Rubric: 1 point for 1st line; 2 points for each other line

2. [3pts] Give a type for $\mathbf{Y}$ or explain why it does not have a type.

   Answer:

   There is no type—it fails the occurs check! The expression recurses infinitely, as per part a, and therefore cannot be represented with a type.