Homework 2: Functions, scope, and storage

CSE 130: Programming Languages

Early deadline: July 11 23:59, Hard deadline: July 14 23:59

Names & IDs:

Solutions

1 Variable Bindings and Closures [9pts]

In this problem we will explore how variable capture differs in JavaScript and Haskell due to mutable/immutable bindings.

1. [3pts] Consider the following JavaScript code snippet:

```
let x = y => y + 1;
x = y => y + x(2);
x(2);
```

What happens when the function call `x(2)` is made? (Run the code.) Why does this happen instead of returning 2 + 2 + 1 (or 5)? (Hint: think about which function the binding `x` refers to – line 1 or 2 – in evaluating the function body on line 2.)

Answer:

Calling `x(2)` results in a stack overflow. (`RangeError: Maximum call stack size exceeded`)

This is because, in JavaScript, `x` on line 2 does not capture `x`'s old value; rather, it becomes a (recursive) reference to the `x` currently being defined. Thus, calling this version of `x` results in endless recursion, which causes the JavaScript engine to throw a stack overflow exception.

Rubric: 1 point for correct behavior; 2 points for correct reason

2. [3pts] Now consider the following code snippet:

```
let x = y => y + 1;
x = (z => y => y + z(2))(x);
x(2);
```

What happens when this code is executed? In contrast to the previous code snippet, why does this code return the correct sum? Explain briefly.

Answer:

Calling `x(2)` results in `5` as an answer.

Because `x` is explicitly passed as a parameter into a function that creates a new function given a value for parameter `z`, the old value of `x` (from line 1) is captured (as `z`) in the closure created at line 2. Thus, inside the function defined on line 2, `x` refers to the `x` defined on line 1 and explicitly captured: that is, it is not a recursive occurrence, so does not lead to diverging computation.

Rubric: 1 point for correct behavior; 2 points for correct reason

3. [3pts] Now consider the same code snippet but, written in Haskell:

```
let x = \y -> y + 1 in let x = (\z -> \y -> y + (z 2)) x in x 2
```

What happens when this expression is evaluated? (Run the code with GHCi.) Briefly explain why this behavior is different from its JavaScript counterpart.

Answer:

Calling `x 2` results in a Stack Overflow (`** Exception:  stack overflow`)

Haskell treats each let binding as a redefinition. Thus, rather than referring to the `x` defined on line 1, the occurrence of `x` does not refer to line 1; instead, Haskell treats this x as a recursive occurrence. (Put differently, there is no way to redefine a variable while capturing its previous value).

Thus, the semantics of this program are very much like the first example: we treat `x` on line 2 as a recursive call rather than as a reference to the previous definition (line 1); thus, we get unbounded recursion, and eventually a stack overflow.

Rubric: The original HW question was incorrect. It didn't use nested `let` expressions so failed at compile time. Everybody gets full credit for this question.

2 Closures and access links [14pts]

Consider the following JavaScript code.

```
1: let z = 2;
2: let f = function(x) {return f(x+1);}
3: let h = f;
4: f = function(x) {z++; return x+1;}
5: let y = h(4)*z;
```

1. [9pts] Fill in the missing parts in the following diagram of the run-time structures for execution of this code up to the point where the call inside `f(5)` is about to return. Note that `y` is still unassigned at the time.

   In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code (▷). Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link". The first one is done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

		Activation Records		*Closures*	*Compiled Code*
(1)		access link	(0)		
		`z`	3		
		`f`	●	▷⟨ (1) , ● ⟩	▷| code on line 2 |
		`h`	●	▷⟨ (1) , ● ⟩	▷| code on line 4 |
		`y`	— —		
(2)	`h(4)`	access link	1		
		`x`	4		
(3)	`f(5)`	access link	1		
		`x`	5		

Rubric: 1 point for each other correct fill-in/line.

2. [1pts] If we run the code, what is the final value that is assigned to `y`?

Answer:

18

Rubric: 1 point for correct answer

3. [4pts] Suppose we change the definition on line 2 to a named function as follows:

```
let f = function f(x) {return f(x + 1);}
```

If we run the code again, the evaluation of `h(4)` will not terminate with a return value. What do you suspect is the reason for this change in behavior? (Please keep your answer short, but specific.)

Answer:

The program will loop forever because JavaScript's hoisting will move the function declaration to the top of the program, so the name `f` within the function body will refer to this function, on line 2 instead of the one on line 4.

Rubric: 2 points for explaining the program will loop forever; 2 points for why.

3 Memory management and high-order functions [15pts]

This question asks about memory management in the evaluation of the following code that contains high-order functions.

```
let x = 5;
{
  function f(y) {
    return (x+y)-2;
  }
  {
    function g(h) {
      let x = 7;
      return h(x);
    }
    {
      let x = 10;
      g(f);
    }
  }
}
```

1. [13pts] Fill in the missing information in the following depiction of the run-time stack after the call to `h` inside the body of `g`. Remember that function values are represented by closures, and that a closure is a pair consisting of an environment (pointer to an activation record) and compiled code.

   In this drawing, a bullet (•) indicates that a pointer should be drawn from this slot to the appropriate closure or compiled code (▷). Since the pointers to activation records cross and could become difficult to read, each activation record is numbered at the far left. In each activation record, place the number of the activation record of the statically enclosing scope in the slot labeled "access link". The first two are done for you. Also use activation record numbers for the environment pointer part of each closure pair. Write the values of local variables and function parameters directly in the activation records.

Activation Records

(1)		access link	(0)
		`x`	5
(2)		access link	(1)
		`f`	•
(3)		access link	(2)
		`g`	•
(4)		access link	(3)
		`x`	10
(5)	`g(f)`	access link	(3)
		`h`	•
		`x`	7
(6)	`h(x)`	access link	(2)
		`y`	7

Closures

▷⟨ (2) , • ⟩

▷⟨ (3) , • ⟩

Compiled Code

▷| code for `f` |

▷| code for `g` |

Rubric: 1 point for each other correct fill-in/line.

2. [2pts] What is the value of the call `g(f)`? Why?

Answer:

10. Well the `h(x)` call returns (5+7)-2, which is itself returned by `g(f)`; note that the `x` is the one from activation record 1 (chased from $6 \to 2 \to 1$).

Rubric: 2 points for correct answer and explanation; 0 points for only answer.

4 More substitution and variable capture [12pts]

In this problem we're going to look at capture-avoiding substition again. Once you've mastered capture-avoiding substitution and β-reduction, you will be able to do more advance things (like encode numbers and booleans).

For each of the terms below, perform the capture-avoiding substitution showing intermediate steps.

1. [4pts] Perform this substitution $((\lambda x.((\lambda x.x)\ 2) + x)\ x)[x := 3]$

Answer:

$(\lambda x.((\lambda x.x)\ 2) + x)[x := 3]\ x[x := 3] = (\lambda x.((\lambda x.x)\ 2) + x)\ 3$

Rubric: 2 points for correct distribution of $[x := 3]$; 1 point for correct final answer

Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

Answer:

No.

Rubric: 1 point for correct answer

2. [4pts] Perform this substitution $(\lambda y.(\lambda xyz.z)\ y\ z\ y)[z := w]$

Answer:

$(\lambda y.(\lambda xyz.z)\ y\ z\ y)[z := w] =$
$(\lambda y.(\lambda xyz.z)[z := w]\ y[z := w]\ y[z := w]\ y[z := w]) =$
$(\lambda y.(\lambda xyz.z)\ y\ w\ y)$

Rubric: 1 point for each correct step

Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

Answer:

Yes, it would've produced $(\lambda y.(\lambda xyz.w)\ y\ w\ y)$.

Rubric: 1 point for correct answer

3. [4pts] Perform this substitution $(\lambda p.(\lambda x.p\ (x\ x))\ (\lambda x.p))[x := p]$:

Answer:

$(\lambda p.(\lambda x.p\ (x\ x))\ (\lambda x.p))[x := p] =$
$(\lambda p.(\lambda x.p\ (x\ x))[x := p]\ (\lambda x.p)[x := p]) =$
$(\lambda p.(\lambda x.p\ (x\ x))\ (\lambda x.p))$

Rubric: 1 point for each correct step

Suppose you used non-capture-avoiding substitution instead. Would the result be incorrect in this case? Explain.

Answer:

Yes it would've produced $(\lambda p.(\lambda x.p\ (p\ p))\ (\lambda x.p))$.

Rubric: 1 point for correct answer

5 [32pts] Type Inference

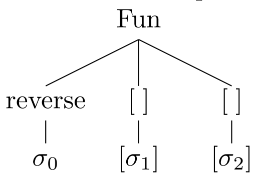
In this problem you will apply the Hindley-Milner type inference algorithm we discussed in class to figure out the type of two μHaskell declarations. For both, you must go through the five steps: creating the parse trees, assigning type variables, generating constraints, solving the constraints, and finally circling your final type answer (if no type error was encountered).

5.1 [14pts] Infer the type of `reverse`:

```
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
               = reverse xs ++ (x:[])
               = (++) (reverse xs) (x:[])
               = ((++) (reverse xs)) (x:[])
               = ((++) (reverse xs)) (((:) x) [])
```

Answer:

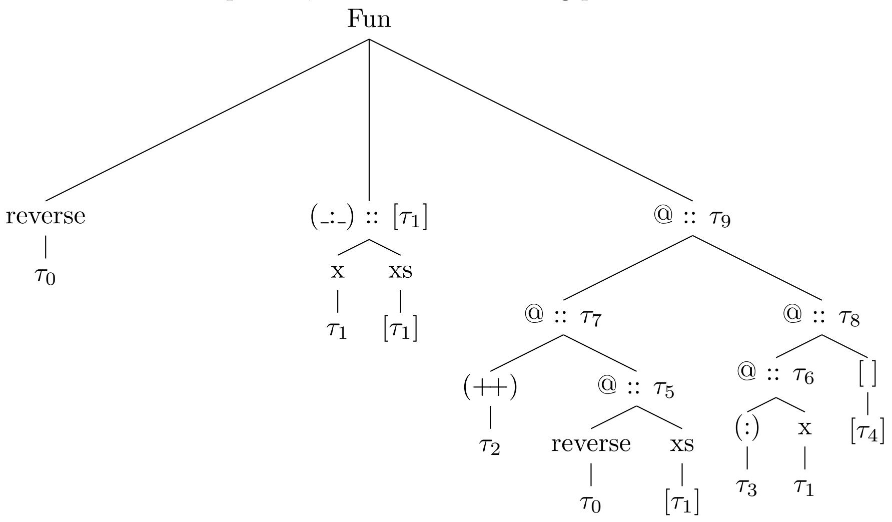
From the first equation, we have the following parse tree:



From this parse tree we then have this single constraint:

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

From the second equation, we have the following parse tree:



From this parse tree we then have these constraints:

(1) $\tau_0 = [\tau_1] \to \tau_9$

(2) $\tau_2 = \tau_5 \to \tau_7$

(3) $\tau_0 = [\tau_1] \to \tau_5$

(4) $\tau_3 = \tau_1 \to \tau_6$

(5) $\tau_3 = \tau_1 \rightarrow [\tau_1] \rightarrow [\tau_1]$

(6) $\tau_6 = [\tau_4] \rightarrow \tau_8$

(7) $\tau_7 = \tau_8 \rightarrow \tau_9$

Unifying (1) and (3):

(8) $\tau_9 = \tau_5$

Unifying (4) and (5):

(9) $\tau_6 = [\tau_1] \rightarrow [\tau_1]$

Unifying (9) and (6):

(10) $\tau_4 = \tau_1$

(11) $\tau_8 = [\tau_1]$

Unifying (11) and (7):

(12) $\tau_7 = [\tau_1] \rightarrow \tau_9$

Unifying (2) and (12) and (8):

(13) $\tau_2 = \tau_9 \rightarrow [\tau_1] \rightarrow \tau_9$

Also we know the type of node $\tau_2 =$ `(++)`

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

Unifying (13) with (14)

(15) $\tau_{10} = \tau_1$

(16) $\tau_9 = [\tau_{10}]$

(17) $\tau_9 = [\tau_1]$

Substituting (17) into (1)

(18) $\tau_0 = [\tau_1] \rightarrow [\tau_1]$

Since $\sigma_0 = \tau_0$, unifying (0) and (18):

(19) $\tau_1 = \sigma_1$

(20) $\tau_1 = \sigma_2$

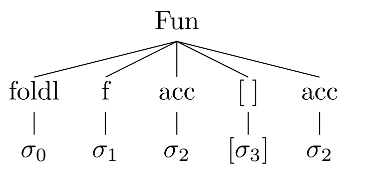
Thus, reverse :: $\tau_0 = [\tau_1] \rightarrow [\tau_1]$

5.2 [18pts] Infer the type of `foldl`:

```
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
                   = ((foldl f) ((f acc) x)) xs
```

Answer:

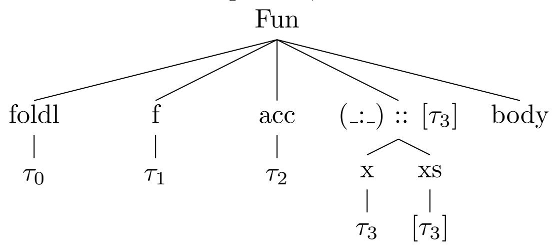
From the first equation, we have the following parse tree:

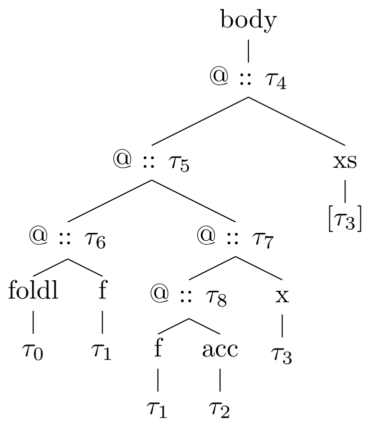


From this parse tree we then have this single constraint:

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

From the second equation, we have the following parse tree:





From this parse tree we then have these constraints:

(1) $\tau_0 = \tau_1 \to \tau_2 \to [\tau_3] \to \tau_4$

(2) $\tau_0 = \tau_1 \to \tau_6$

(3) $\tau_1 = \tau_2 \to \tau_8$

(4) $\tau_8 = \tau_3 \to \tau_7$

(5) $\tau_6 = \tau_7 \to \tau_5$

(6) $\tau_5 = [\tau_3] \to \tau_4$

Unifying (1) and (2):

(7) $\tau_6 = \tau_2 \to [\tau_3] \to \tau_4$

Substituting (4) into (3):

(8) $\tau_1 = \tau_2 \to \tau_3 \to \tau_7$

Substituting (5) into (2):

(9) $\tau_0 = \tau_1 \to \tau_7 \to \tau_5$

Unifying (5) and (7):

(10) $\tau_5 = [\tau_3] \to \tau_4$

(11) $\tau_7 = \tau_2$

Substituting (6) into (5):

(12) $\tau_6 = \tau_7 \to [\tau_3] \to \tau_4$

Unifying (8) and (3):

(13) $\tau_8 = \tau_3 \to \tau_7$

Substituting (8) into (9):

(14) $\tau_0 = (\tau_2 \to \tau_3 \to \tau_7) \to \tau_7 \to \tau_5$

Unifying (9) and (2):

(15) $\tau_6 = \tau_7 \to \tau_5$

Substituting (10) into (14):

(16) $\tau_0 = (\tau_2 \to \tau_3 \to \tau_7) \to \tau_7 \to [\tau_3] \to \tau_4$

Substituting (11) into (16):

(17) $\tau_0 = (\tau_2 \to \tau_3 \to \tau_2) \to \tau_2 \to [\tau_3] \to \tau_4$

Since $\sigma_0 = \tau_0$, unifying (0) and (17):

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

(18) $\sigma_1 = (\tau_2 \to \tau_3 \to \tau_2)$

(19) $\sigma_2 = \tau_2$

(20) $\sigma_3 = \tau_3$

(21) $\sigma_2 = \tau_4$

Unifyng (19) and (21):

(22) $\tau_2 = \tau_4$

Substituting (22) into (17):

foldl :: $\tau_0 = (\tau_2 \to \tau_3 \to \tau_2) \to \tau_2 \to [\tau_3] \to \tau_2$

Acknowledgements

Any acknowledgements, crediting external resources or people should be listed below.