HW 7

50pts. You can work on your own or in teams of two.
Posted Tuesday, November 29, 2022
Due Friday, December 9, 2022

Problem 1 (7pts). Consider the *twice* combinator:

$$twice = \lambda f.\lambda x.f(f\ x)$$

Reduce expression *twice twice f x* into *normal form* using *normal order reduction*. For full credit, show each step on a separate line.

Problem 2 (8pts). Now consider the Haskell implementation of *twice*:

```
twice f x = f (f x)
```

(a) What is the type of `twice`?
(b) What is the type of expression `twice twice`?
(c) If the type of `fun` is `Int->Int`, what is the type of expression `twice twice fun`?
(d) If the type of `fun` is `Int->Int` and expression `twice twice fun v` is well-typed, what is the type of `twice twice fun v`?

Note: You do not need to justify your answer, just state the corresponding type expression.

Problem 3 (10pts). This is a skeleton of the quicksort algorithm in Haskell:

```
quicksort [] = []
quicksort (a:b) = quicksort ...  ++ [a] ++ quicksort ...
```

(a) Fill in the two elided expressions (shown as ...) with appropriate list comprehensions.
(b) Now fill in the two elided expressions with the corresponding monadic-bind expressions.

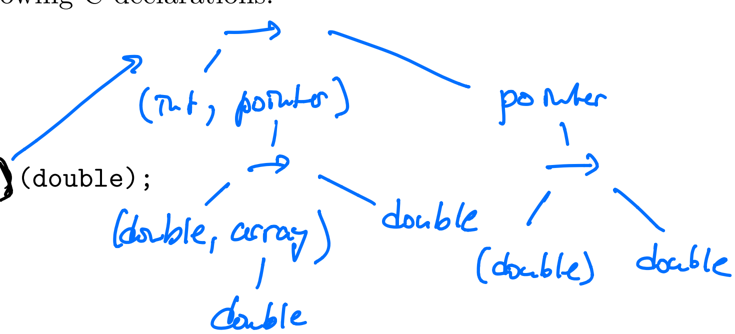
Problem 4 (5pts). In the following code, which of the variables will a compiler consider to have compatible types under structural equivalence? Under strict name equivalence? Under loose name equivalence?

```
type A = array [1..10] of integer
type B = A
a : A
b : A
c : B
d : array [1..10] of integer
```

Structural: a, b, c, d
Loose name: a, b, c d
Strict name: a, b c d

Problem 5 (10pts). Show the type trees for the following C declarations:

```
double *a[n];
double (*a)[n];
double (*a[n])();
double (*a())[n];
double (*a(int, double(*)(double, double[])))(double);
```



Problem 6 (10pts). Consider the Pascal-like code for function `compute`. Assume that the programming language allows a mixture of parameter passing mechanisms as shown in the definition.

```
double compute(first : integer /*by value*/, last : integer /*by value*/,
   incr : integer /*by value*/, i : integer /*by name*/, term : double /*by name*/)

   result : double := 0.0
   i := first
   while i <= last do
       result := result + term
       i := i + incr
   endwhile
   return result
```

(a) (2pts) What is returned by call `compute(1, 10, 1, i, A[i])`?
(b) (2pts) What is returned by call `compute(1, 5, 2, j, 1/A[j])`?
(c) (2pts) `compute` is a classic example of *Jensen's device*, a technique that exploits call by name and side effects. In one sentence, explain what is the benefit of Jensen's device.
(d) (4pts) Write `max`, which uses Jensen's device to compute the maximum value in a set of values based off of an array `A`.

(a) A[1] + A[2] + ... + A[10]

(b) 1/A[1] + 1/A[3] + 1/A[5]

(c) reuse / polymorphism

(d)

```
result : double := term // initialize to first term.
i := first
while i ≤ last do
   if result < term then
      result := term
   i := i + incr
endwhile
return result
```

Q1. $\text{twice twice } f\, x =$

$$\underline{(\lambda f. \lambda x. f\,(f\,x))\ \text{twice}}\ f\, x \longrightarrow$$

$$\underline{(\lambda x.\ \text{twice}\ (\text{twice}\ x))\ f}\ x \longrightarrow$$

$$(\text{twice}\ (\text{twice}\ f))\ x =$$

$$\underline{(\lambda f. \lambda x. f\,(f\,x))\ (\text{twice}\ f)}\ x \longrightarrow$$

$$\underline{(\lambda x.\ (\text{twice}\ f)\ ((\text{twice}\ f)\, x))\ x} \to$$

$$(\text{twice}\ f)\ (\text{twice}\ f\ x) =$$

$$(\underline{(\lambda f. \lambda x. f\,(f\,x))\ f})\ (\text{twice}\ f\ x) \to_\beta$$

$$\underline{(\lambda x. f\,(f\,x))\ (\text{twice}\ f\ x)} \to_\beta$$

$$f\ (f\ (\text{twice}\ f\ x)) =$$

$$f\ (f\ (\underline{(\lambda f. \lambda x. f\,(f\,x))\ f}\ x)) \to_\beta$$

$$f\ (f\ ((\lambda x. f\,(f\,x))\ x) \to_\beta$$

$$f\ (f\ (f\ (f\ x))) \ \blacksquare\ \boxed{NF}$$

Q2: a) $(t \to t) \to t \to t$

b) $(t \to t) \to t \to t$

c) $Int \to Int$ (or $(Num\ t) \Rightarrow t \to t$)

d) Int (or $(Num\ t)$)

Q3:

a)
```
qs [] = []
qs (a:b) = qs [x | x <- b, x ≤ a] ++ [a] ++
           qs [x | x <- b, x > a]
```

b) (one example, just turn guard predicate into bind function)

```
qs [] = []
qs (a:b) = qs (b >>= (\x -> if x ≤ a then [x] else []))
           ++ [a] ++
           qs (b >>= (\x -> if x > a then [x] else []))
```

Q4: above

Q5:

(a) `double *a[n];`

array
pointer
double

(b) `double (*a)[n];`

pointer
array
double

(c) `double (*a[n])();`

array
pointer
→
() double

(d) double (*a())[n]

( ) $\rightarrow$ pointer

array

double

(e) see above

Q6. See above.