HW 7

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

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.

$$\begin{aligned}
twice\ twice\ f\ x &= \lambda f.\lambda x.f(f\ x)twice\ f\ x \\
&\rightarrow_{\beta}\ \lambda x.twice\ (twice\ x))f\ x \\
&\rightarrow_{\beta\&\alpha}\ \lambda x.\lambda f.\lambda x_{1}.f\ (f\ x_{1})(\lambda f.\lambda x_{1}.f\ (f\ x_{1})\ x))\ f\ x \\
&\rightarrow_{\beta}\ \lambda f.\lambda x_{1}.f\ (f\ x_{1})(\lambda f.\lambda x_{1}.f\ (f\ x_{1})\ f))\ x \\
&\rightarrow_{\beta\&\alpha}\ \lambda x_{1}.\lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f(\lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f\ x_{1}))\ x \\
&\rightarrow_{\beta}\ \lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f(\lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f\ x)) \\
&\rightarrow_{\beta}\ \lambda x_{2}.f\ (f\ x_{2})(\lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f\ x)) \\
&\rightarrow_{\beta}\ f\ (f\ (\lambda f.\lambda x_{2}.f\ (f\ x_{2})\ f\ x))) \\
&\rightarrow_{\beta}\ f\ (f\ (\lambda x_{2}.f\ (f\ x_{2})\ x))) \\
&\rightarrow_{\beta}\ f\ (f\ (f(f\ x)))
\end{aligned}$$

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

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

(a) What is the type of `twice`?

The type of `twice` is `(t→ t) → t → t` where `t` is a type variable of any type

(b) What is the type of expression `twice twice`?

A canonical use case would be twice twice f x, which takes a f of type (t→t) to twice to produce a closure of type (t→t). The type of `twice twice` is `(t→ t) → t →t`.

(c) If the type of `fun` is `Int->Int`, what is the type of expression `twice twice fun`?

The type of `twice twice fun` is `Int→Int`

(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`?

The type is `Int`

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.

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

(b) Now fill in the two elided expressions with the corresponding monadic-bind expressions.

```
quicksort (a:b) = (quicksort [x <- b, x < a]) >>= (\x -> [x] ++ [a] ++ (quicksort
[x <- b, x >= a]))
```

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
```

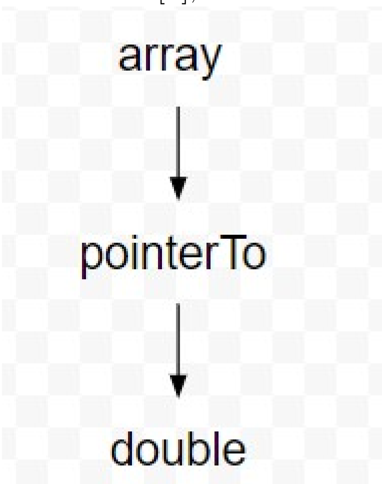
(1) Structural equivalence:
(a,b,c,d)

(2) Strict name equivalence:
(a,b)
(c)
(d)

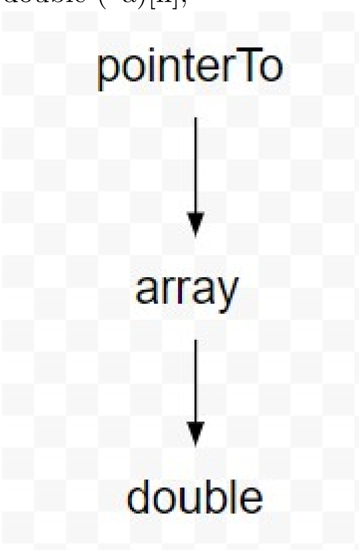
(3) Loose name equivalence:
(a,b,c)
(d)

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

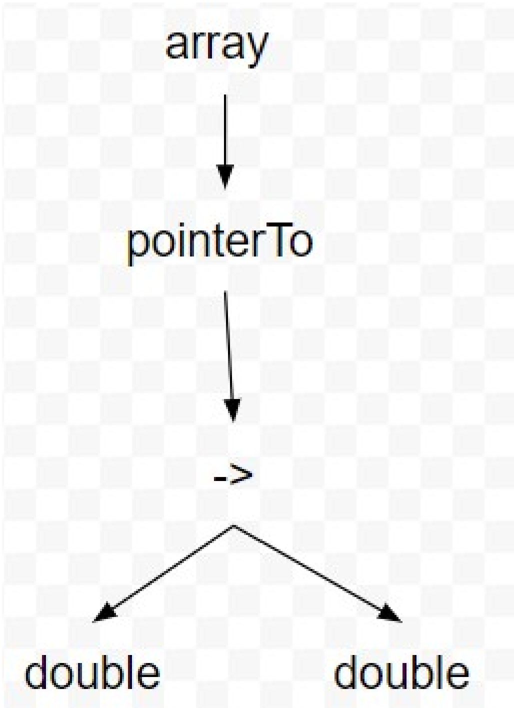
double *a[n];



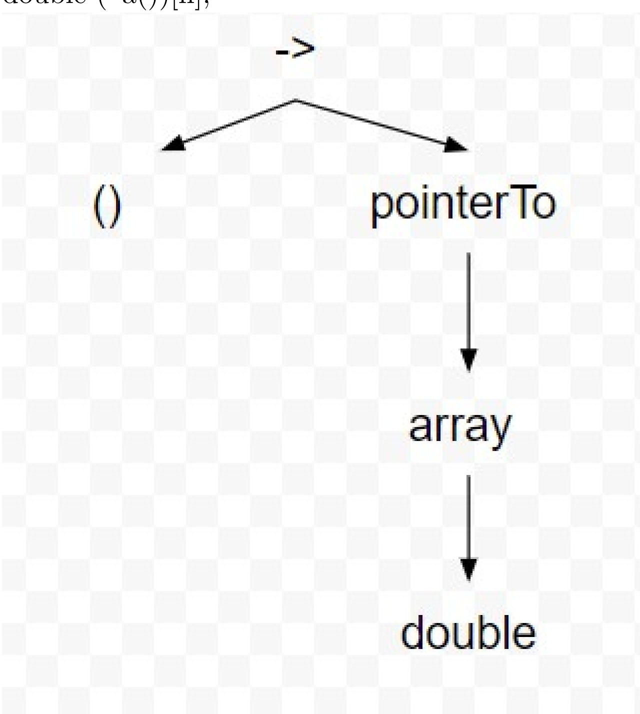
double (*a)[n];



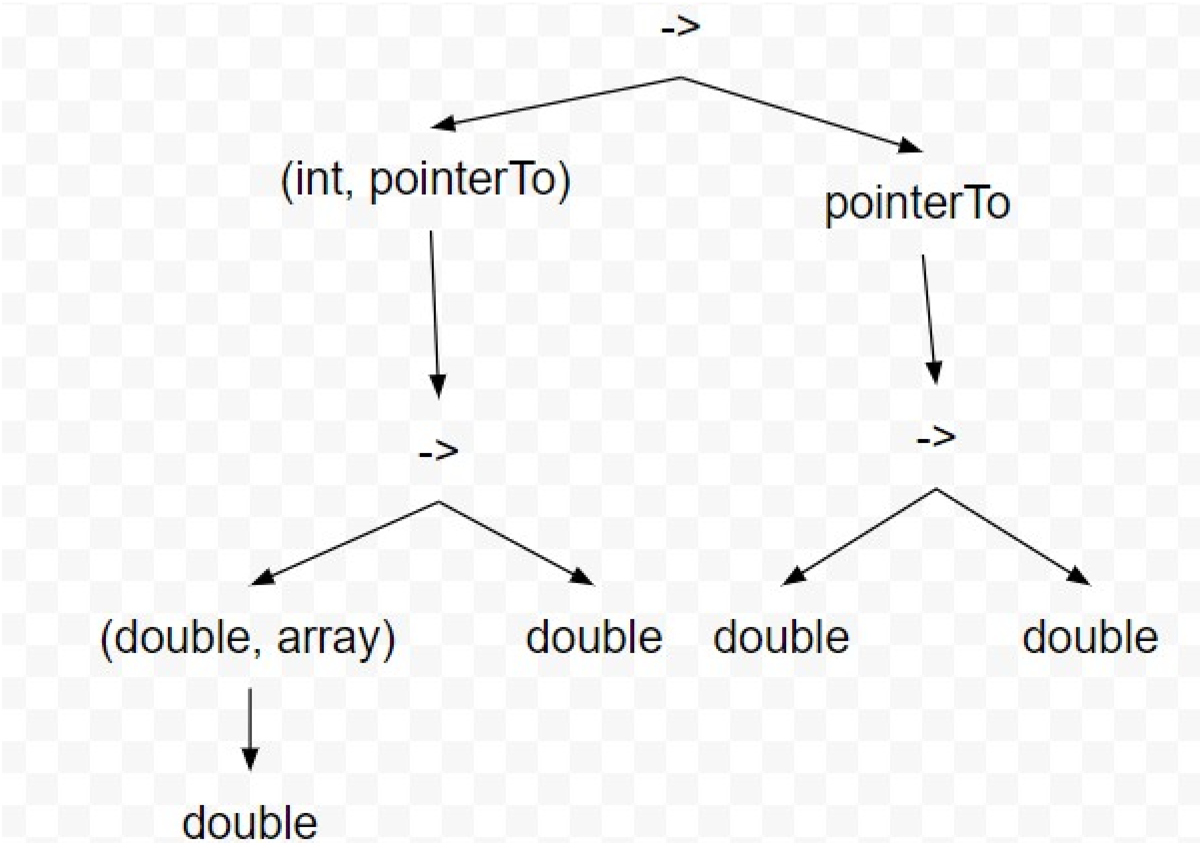
double (*a[n])();



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])`?

$$\sum_{i=1}^{10} A[i]$$

(b) (2pts) What is returned by call `compute(1, 5, 2, j, 1/A[j])`?

$$\frac{1}{A[1]} + \frac{1}{A[3]} + \frac{1}{A[5]}$$

(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.

It provides a easy and flexible way for coding so that we can pass expressions in the function call which would be re-evaluated for multiple times during executions.

(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`.

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

   result : double := term
   i := first
   while i <= last do
       if term > result then
           result := term
       endif
       i := i + incr
   endwhile
   return result
```