

Functional Programming

with and without Haskell

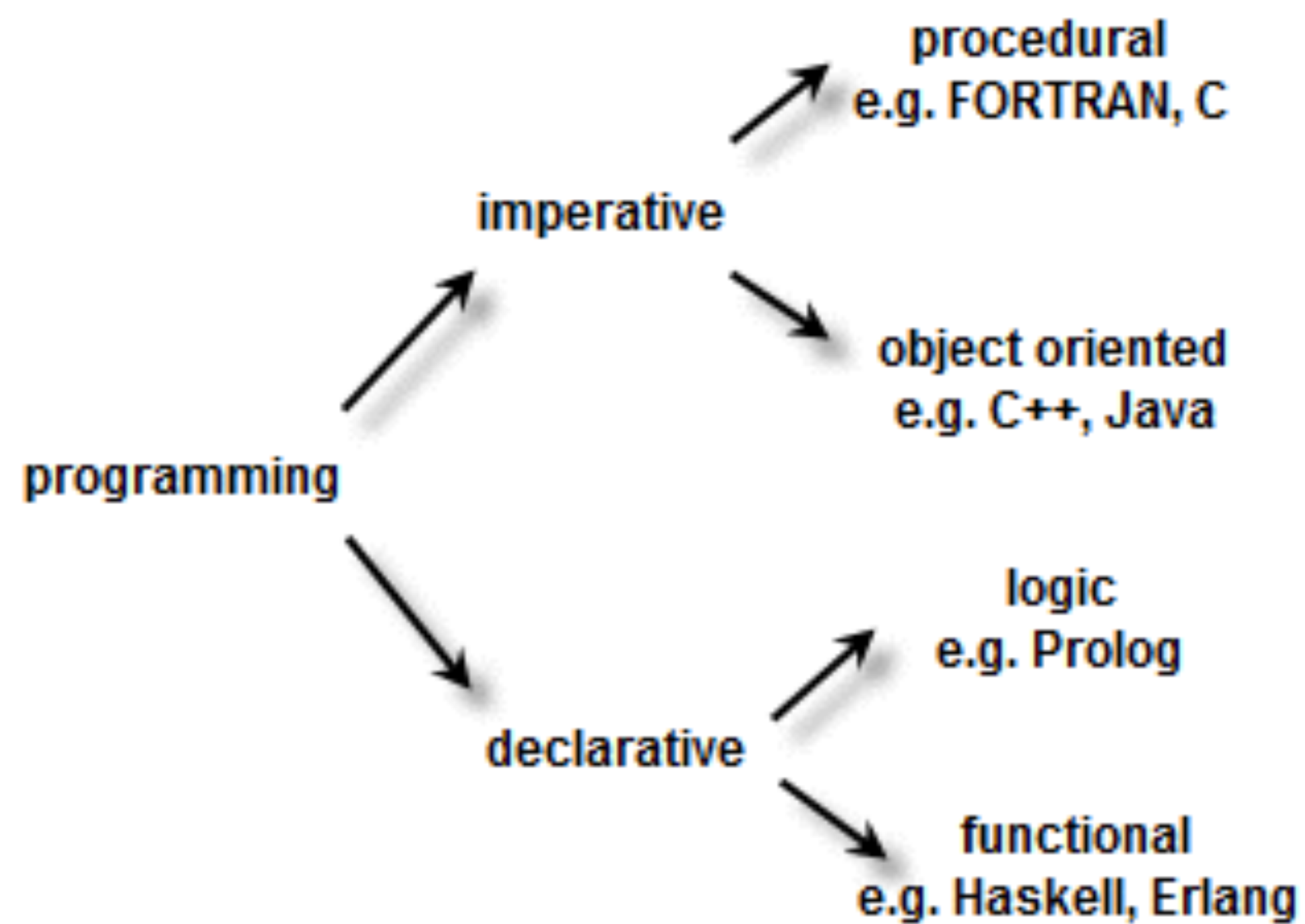
— Alcides Fonseca

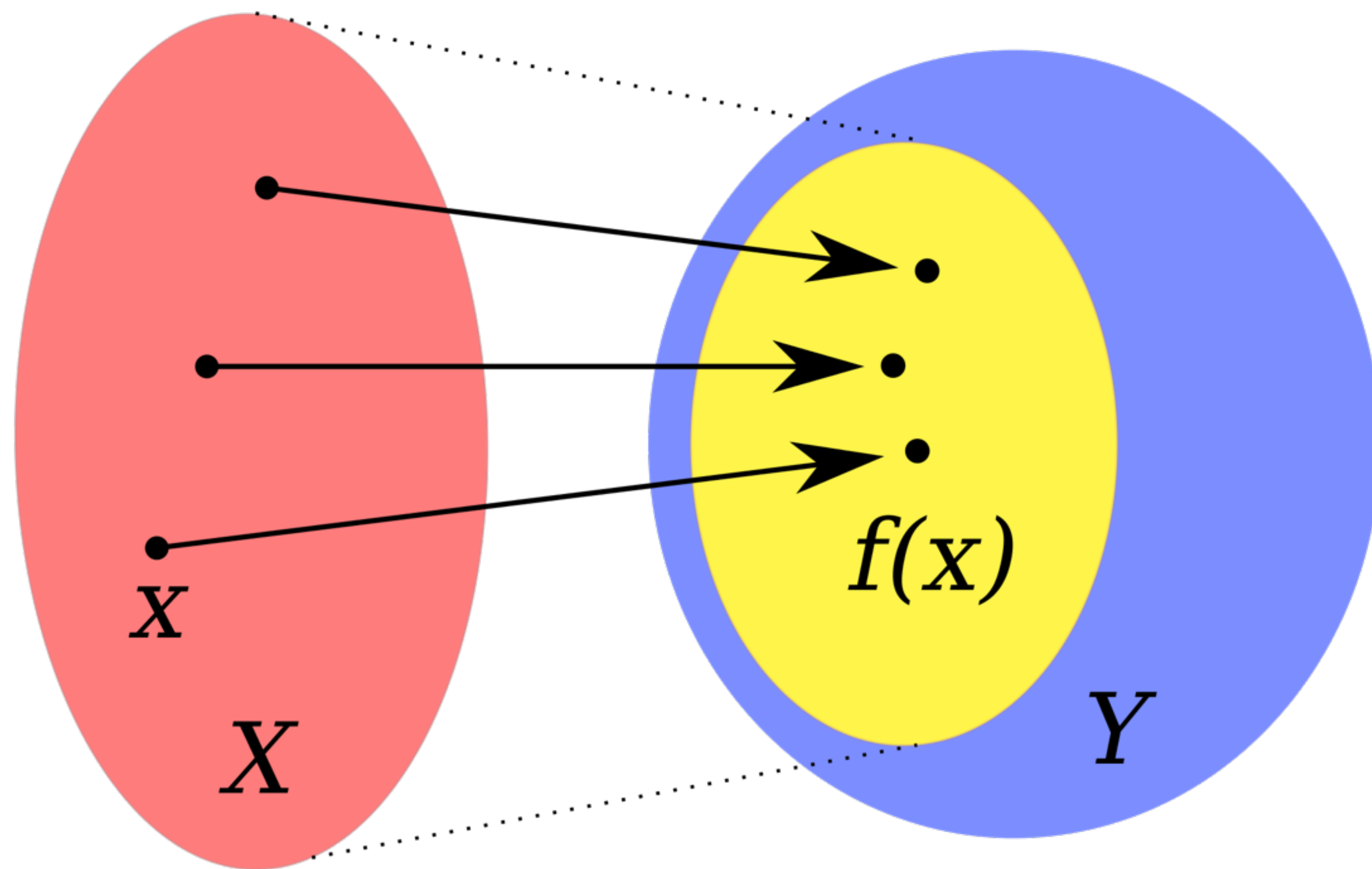
Functional Programming

\ˈfʊŋ(k)-ʃnəl prō-,gramɪŋ\

noun.

The art of writing programs that function as expected.





$$f : X \rightarrow Y$$

```
int ageOf(Person p);
```

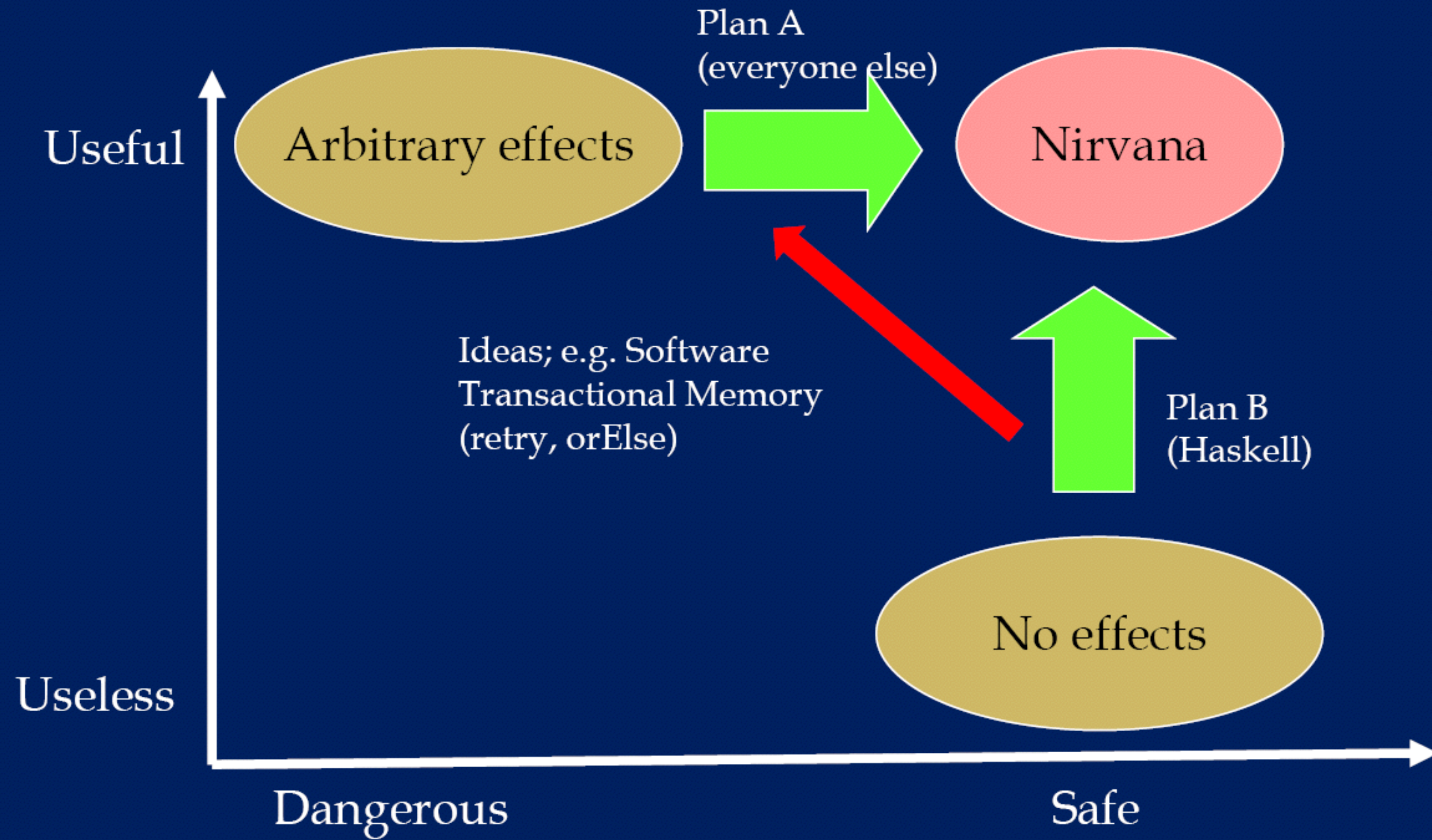
```
int ageOf(Person p) {  
    return today() - birthdate[d];  
}
```

CODE WRITTEN IN HASKELL
IS GUARANTEED TO HAVE
NO SIDE EFFECTS.

...BECAUSE NO ONE
WILL EVER RUN IT?



Lots of cross-over



```
double :: Int -> Int  
double x = x + x
```

```
times :: Int -> Int -> Int
```

```
times 0 _ = 0
```

```
times 1 x = x
```

```
times n x = x + (times (n-1) x)
```

```
double :: Int -> Int
```

```
double = times 2
```

WHY DO YOU LIKE FUNCTIONAL
PROGRAMMING SO MUCH? WHAT
DOES IT ACTUALLY *GET* YOU?

TAIL RECURSION IS
ITS OWN REWARD.



```
times 1 n = n
times a b = a + (times (b-1) a)
```

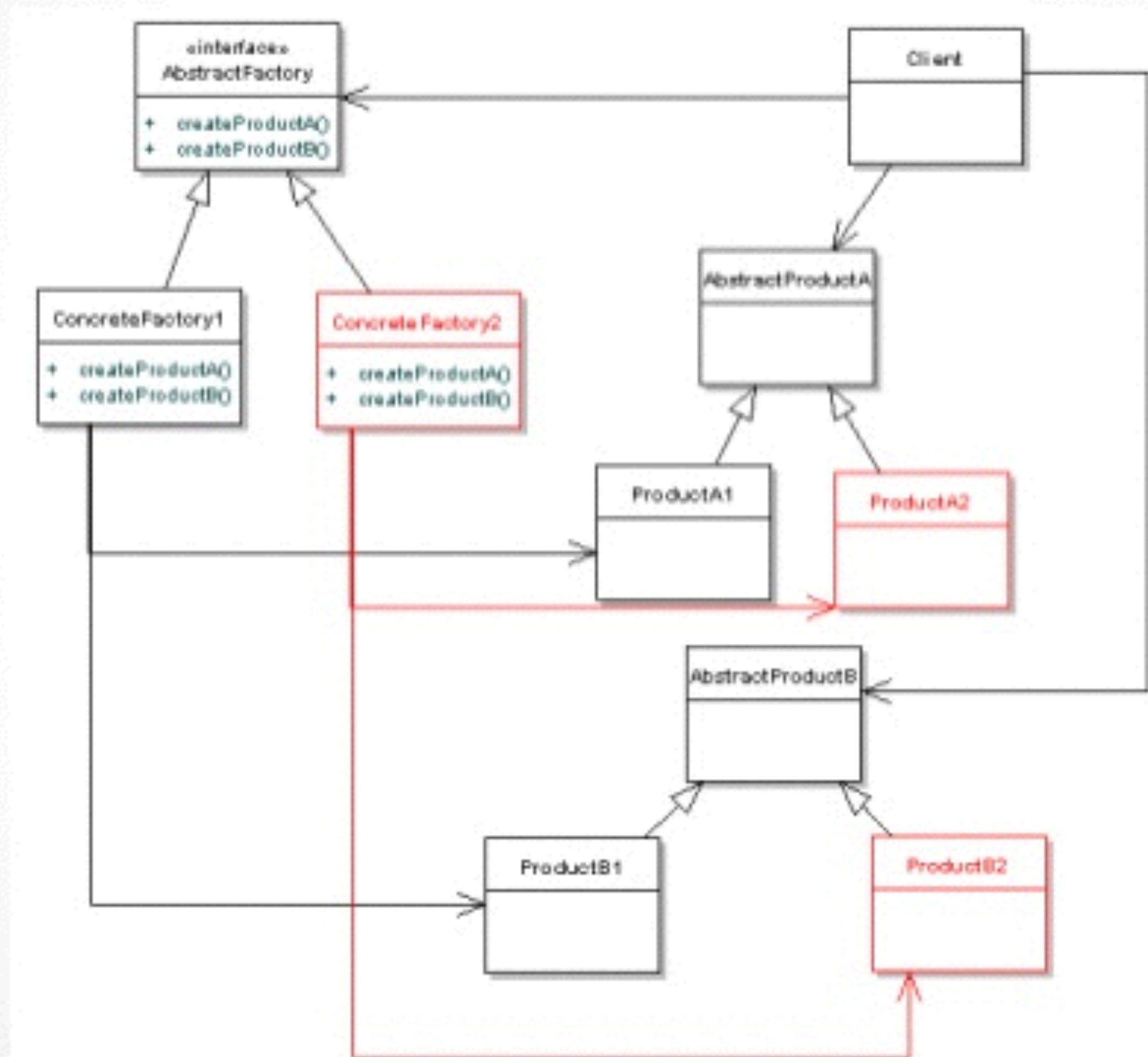
```
power 1 n = n
power a b = a * (power (b-1) a)
```

```
recursiveOp :: (Int -> Int -> Int) -> Int -> Int -> Int
recursiveOp op a 1 = a
recursiveOp op a b = op a (self a (b-1))
                    where self = recursiveOp op
```

```
times = recursiveOp (+)
double = times 2
```

```
recursiveOp :: Num t => (t -> t -> t) -> t -> Int -> t
recursiveOp op a 1 = a
recursiveOp op a b = op a (self a (b-1))
    where self = recursiveOp op
```

Abstract Factory Pattern in Java



```
ones :: [Int]  
ones = 1:ones
```

`doubleList :: [Int] -> [Int]`

`doubleList [] = []`

`doubleList (d:ds) = (double d):(doubleList ds)`

1 :
2 :
3 :
4 :
[]

`applyOverList :: (Int -> Int) -> [Int] -> [Int]`

`applyOverList f xs = [ f x | x <- xs ]`

`map :: (a -> b) -> [a] -> [b]`

`foldr :: (a -> b -> b) -> b -> [a] -> b`

Hoogλe

Data.Map.insert

Search

Packages

containers

insert :: Ord k => k -> a -> Map k a -> Map k a

containers Data.Map.Lazy, containers Data.Map.Strict

⊕

O(log n)

Insert a new key and value in the map. If the key is already present in the map, the associated value is replaced with the supplied value. insert is equivalent to insertWith const. > insert 5 'x' (fromList [(5,'a'), (3,'b')]) == fromList [(3, 'b'), (5, 'x')] >

insertLookupWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> (Maybe a, Map k a)

containers Data.Map.Lazy, containers Data.Map.Strict

⊕

O(log n)

Combines insert operation with old value retrieval. The expression (insertLookupWithKey f k x map) is a pair map</t> and the second element equal to (insertWithKey f k x map). > let f key new_value old_value = (show key) ++ "." ++ new_value

insertLookupWithKey' :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> (Maybe a, Map k a)

containers Data.Map

⊕

Deprecated. As of version 0.5, replaced by insertLookupWithKey. *O(log n)*. Same as insertLookupWithKey, but the value being inserted to the map is evaluated to WHNF beforehand.

insertWith :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a

containers Data.Map.Lazy, containers Data.Map.Strict

⊕

O(log n)

Insert with a function, combining new value and old value. insertWith f key value mp will insert the pair (key, value) into mp if key does not exist in the map. If the key does exist, the function will insert the pair (key, f new_value old_value). >

insertWith' :: Ord k => (a -> a -> a) -> k -> a -> Map k a -> Map k a

containers Data.Map

⊕

Deprecated. As of version 0.5, replaced by insertWith. *O(log n)*. Same as insertWith, but the value being inserted to the map is evaluated to WHNF beforehand. For example, to update a counter: > insertWith' (+) k 1 m

insertWithKey :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a

containers Data.Map.Lazy, containers Data.Map.Strict

⊕

O(log n)

Insert with a function, combining key, new value and old value. insertWithKey f key value mp will insert the pair (key, value) into mp if key does not exist in the map. If the key does exist, the function will insert the pair (key,f key new_value

insertWithKey' :: Ord k => (k -> a -> a -> a) -> k -> a -> Map k a -> Map k a

containers Data.Map

⊕

Deprecated. As of version 0.5, replaced by insertWithKey. *O(log n)*. Same as insertWithKey, but the value being inserted to the map is evaluated to WHNF beforehand.

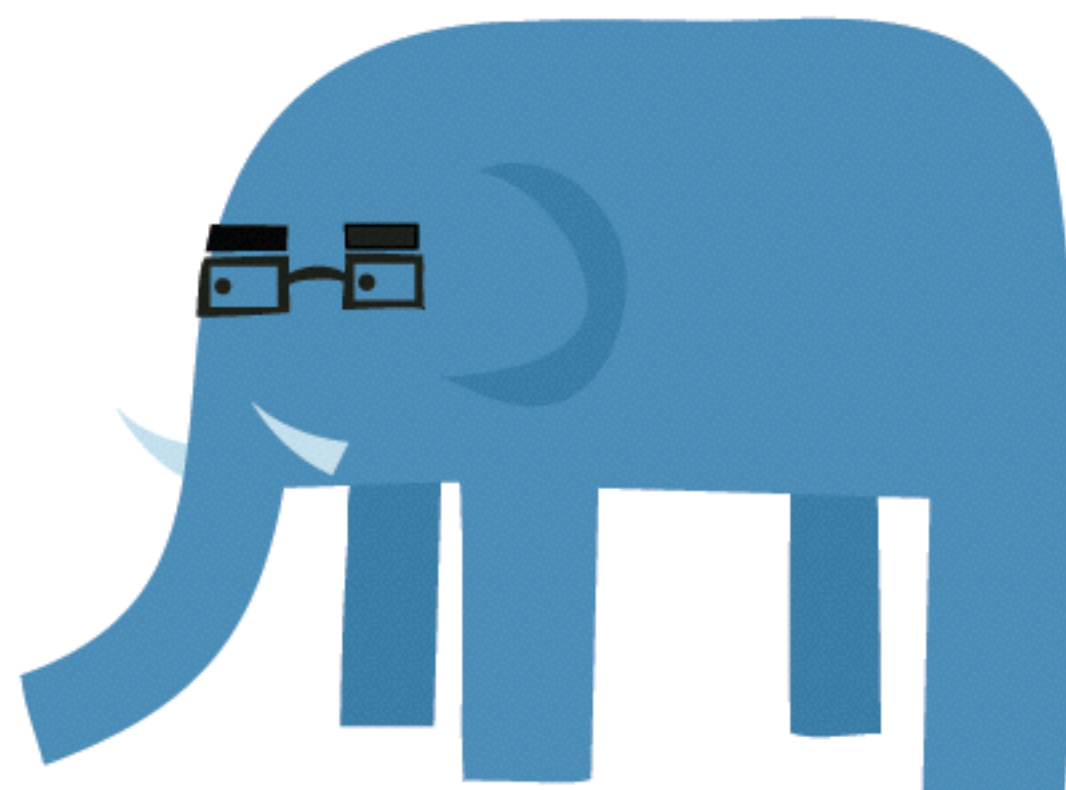
© Neil Mitchell 2004-2013, version 4.2.26

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort (filter (<= x) xs)) ++ [x]
                  ++ (quicksort (filter (> x) xs))
```



Learn You a Haskell for Great Good!

A Beginner's Guide



Miran Lipovača





```
main :: IO ()
```

```
main = do
```

```
    putStr "double: "
```

```
    putStrLn (show (double 3))
```

```
main :: (RealWorld ->) ((), RealWorld)
```

```
main = return a
```

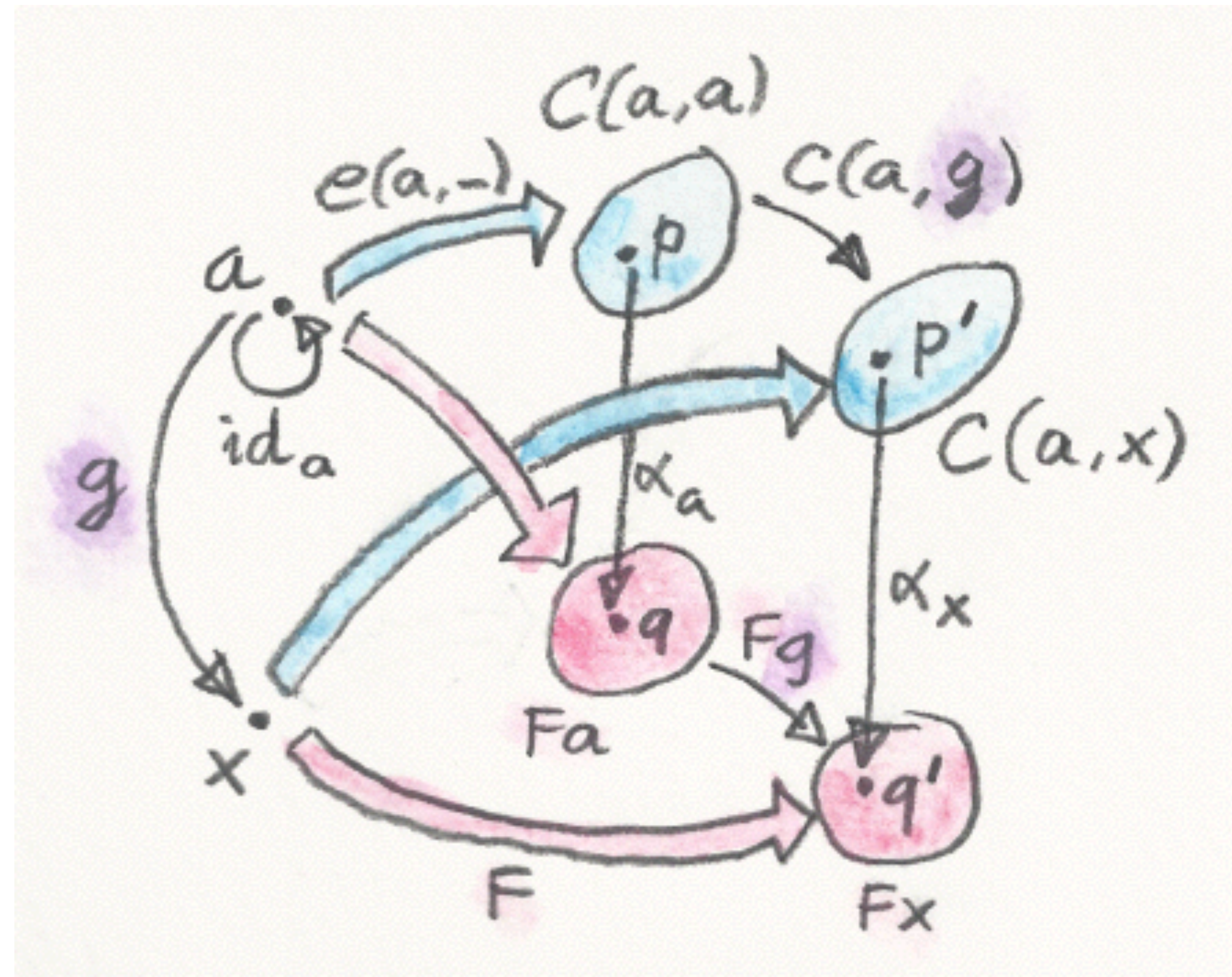
```
    where a = putStr "double: "
```

```
           >>= putStrLn (show (double 3))
```

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

```
return :: a -> IO a
```

Resumo



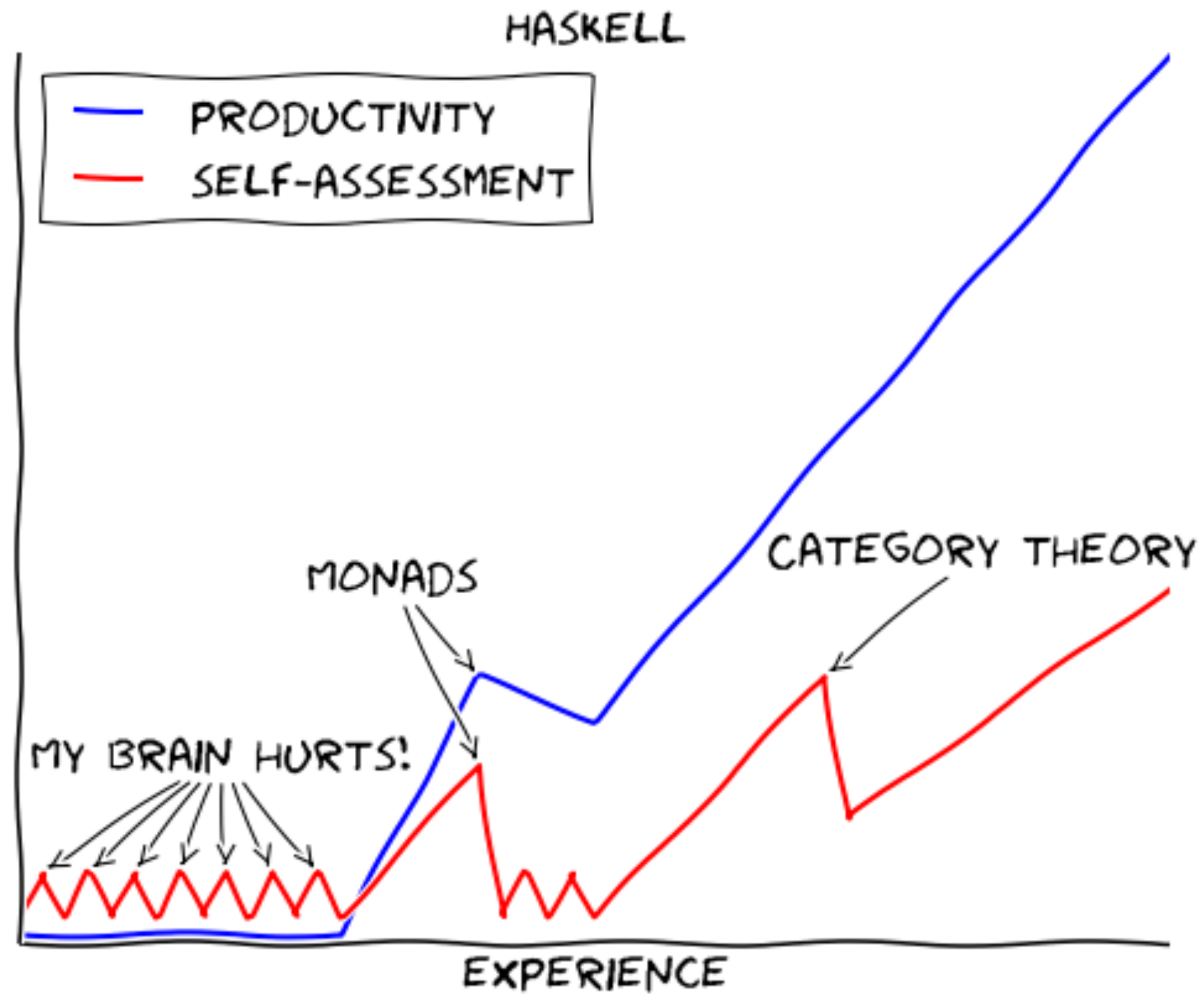
The Good

- Modularity of Patterns
(1001 classes = 3 lines of code)
- Signatures express all effects
(longer, but makes dependencies and bugs easier to find)
- Speeds in the order of magnitude of C
- Hackage, Cabal & Hoogle

The Great

- $\text{loc}(f) \leq 5$
- If it compiles, it will most surely work
- Free parallelization
- Can be proved correct

The Bad



“Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.”

— Eric Raymond

“Ditto for Haskell.”

— Alcides Fonseca

Learn Haskell, so you can improve at X

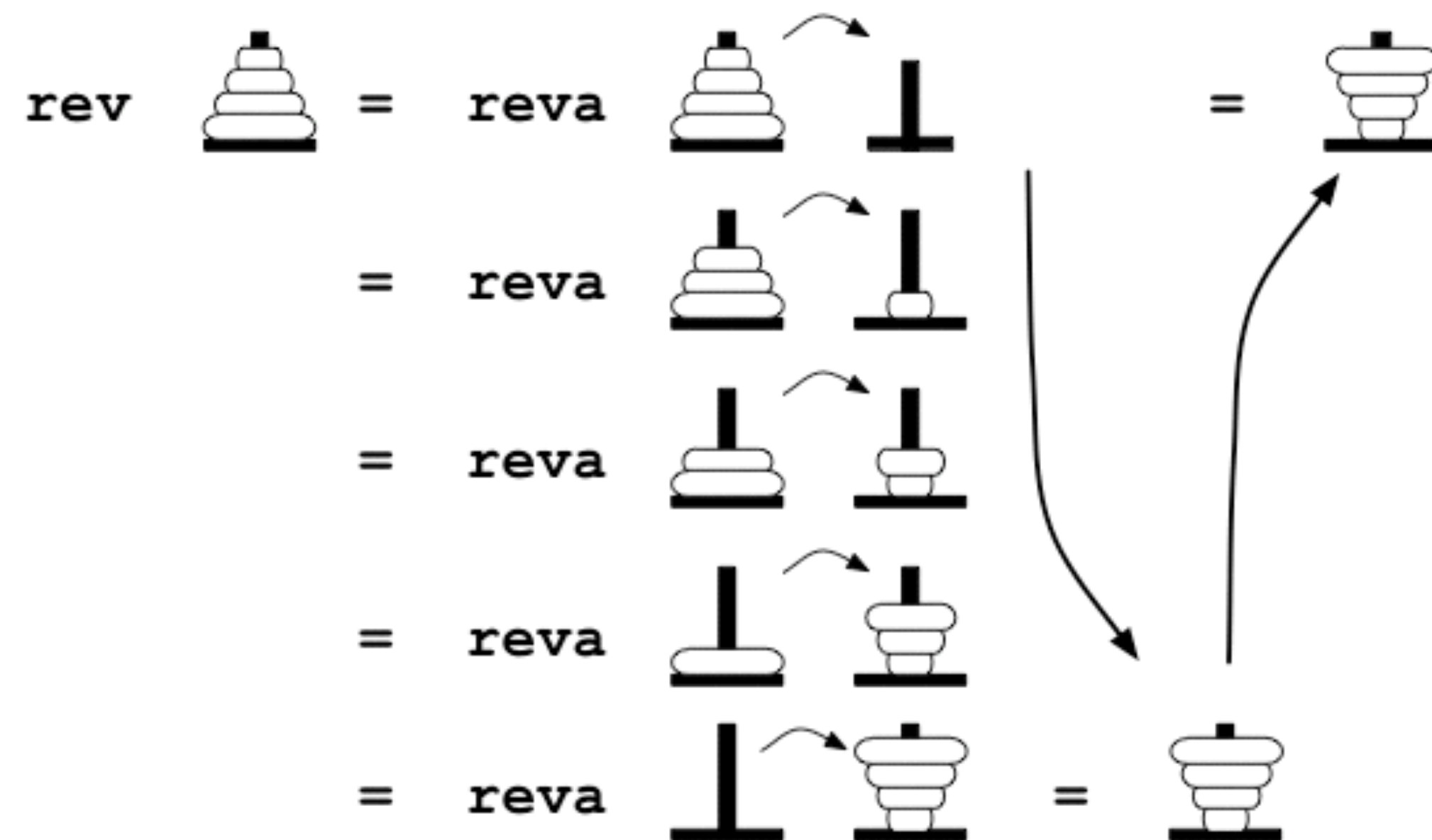
```
trips = [ (from_, to) for from_ in cities for to in cities if not to == from_ ]
```

```
double integral(double (*f)(double x), double a, double b);
```

```
def parseArgument(arg: String):String = arg match {  
  case "-h" | "--help" => displayHelp  
  case "-v" | "--version" => displayVersion  
  case _ => "RTFM"  
}
```

```
var files = from file in DirectoryInfo.EnumerateFiles()  
where file.CreationTimeUtc > dt1 &  
file.CreationTimeUtc < dt2  
select file;
```

$$\begin{aligned}
 \text{rev } \boxed{\text{xs}} &= \text{reva } \boxed{\text{xs}} \perp \\
 \text{where } \text{reva } \perp \boxed{\text{ys}} &= \boxed{\text{ys}} \\
 \text{reva } \boxed{\text{xs}} \boxed{\text{ys}} &= \text{reva } \boxed{\text{xs}} \boxed{\text{ys}}
 \end{aligned}$$



Thank you

— Alcides

me@alcidesfonseca.com

https://github.com/alcides/haskell_tutorial