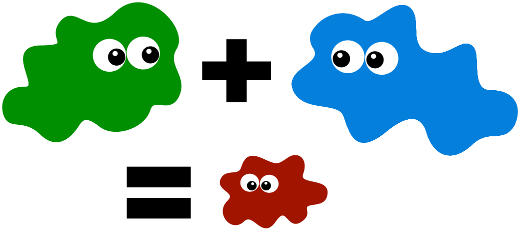
The Evolution of Curry

From Protozoon to Mammal

From Prolog to Haskell

The Best Genes of Both Parents



- functional (algebraic data types, higher order, laziness)
- logic (non-determinism, narrowing)

Why do You need Non-Determinism / Narrowing?

Why do You need Non-Determinism / Narrowing?

- logic puzzles (wolf, sheep and cabbage)

Why do You need Non-Determinism / Narrowing?

- ~~logic puzzles (wolf, sheep and cabbage)~~

Why do You need Non-Determinism / Narrowing?

- ~~logic puzzles (wolf, sheep and cabbage)~~
- enumeration of test cases (SmallCheck)

Why do You need Non-Determinism / Narrowing?

- ~~logic puzzles (wolf, sheep and cabbage)~~
- enumeration of test cases (SmallCheck)
- efficient enumeration of test cases (Lazy SmallCheck)

Why do You need Non-Determinism / Narrowing?

- ~~logic puzzles (wolf, sheep and cabbage)~~
- enumeration of test cases (SmallCheck)
- efficient enumeration of test cases (Lazy SmallCheck)
- circuit analysis (Wired)

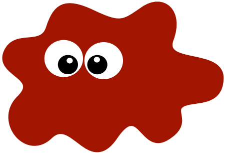
Why do You need Non-Determinism / Narrowing?

- ~~logic puzzles (wolf, sheep and cabbage)~~
- enumeration of test cases (SmallCheck)
- efficient enumeration of test cases (Lazy SmallCheck)
- circuit analysis (Wired)
- <insert your idea here>

The first of its Kind

characteristics PAKCS

- Portland Aachen Kiel Curry System
- Michael Hanus
- non-determinism, narrowing
- target language: Prolog
- search strategy: depth first



Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1
```

Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
```

Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
0
More?
```

Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
0
More?
1
More?
```

Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
0
More?
1
More?
No more Solutions
```

Non-Determinism

- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
0
More?
1
More?
No more Solutions
```

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

Non-Determinism

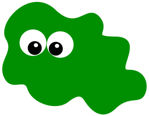
- overlapping rules induce non-determinism

```
coin :: Int
coin = 0
coin = 1

Main> coin
0
More?
1
More?
No more Solutions
```

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y

coin’ :: Int
coin’ = 0 ? 1
```

Functional Species

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

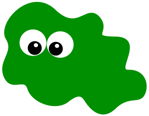


call-by-value

call-by-name

call-by-need

Functional Species



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

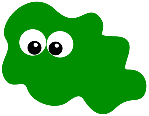
call-by-value

```
double (0+1) = double 1
```

call-by-name

call-by-need

Functional Species



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

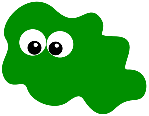
call-by-value

```
double (0+1) = double 1
             = 1+1
```

call-by-name

call-by-need

Functional Species



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

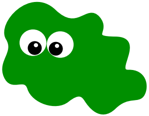
call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

call-by-name

call-by-need

Functional Species



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

call-by-value

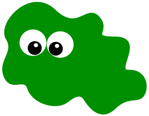
```
double (0+1) = double 1
             = 1+1
             = 2
```

call-by-name

```
double (0+1) = (0+1) + (0+1)
```

call-by-need

Functional Species



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

call-by-value

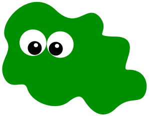
```
double (0+1) = double 1
             = 1+1
             = 2
```

call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
```

call-by-need

Functional Species



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

call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

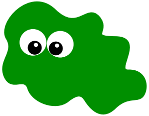
call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
```

call-by-need

Functional Species

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



call-by-value

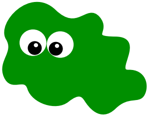
```
double (0+1) = double 1
             = 1+1
             = 2
```

call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
             = 2
```

call-by-need

Functional Species



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

call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

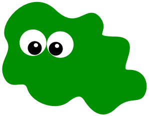
call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
             = 2
```

call-by-need

```
double (0+1) = let x=0+1 in x+x
```

Functional Species



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

call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

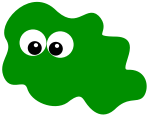
call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
             = 2
```

call-by-need

```
double (0+1) = let x=0+1 in x+x
             = let x=1 in x+x
```

Functional Species



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

call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

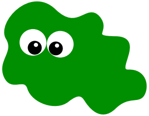
call-by-name

```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
             = 2
```

call-by-need

```
double (0+1) = let x=0+1 in x+x
             = let x=1 in x+x
             = 1+1
```

Functional Species



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

call-by-value

```
double (0+1) = double 1
             = 1+1
             = 2
```

call-by-name

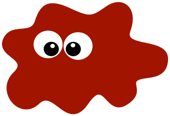
```
double (0+1) = (0+1) + (0+1)
             = 1 + (0+1)
             = 1 + 1
             = 2
```

call-by-need

```
double (0+1) = let x=0+1 in x+x
             = let x=1 in x+x
             = 1+1
             = 2
```

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```

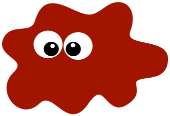


call-by-value

call-by-name

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```

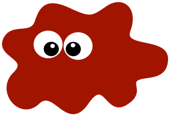


call-by-value

```
double coin = double (0|1)
```

call-by-name

Functional Logic Species



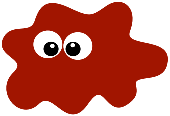
```
coin :: Int
coin = 0
coin = 1
```

call-by-value

```
double coin = double (0|1)
            = double 0 | double 1
```

call-by-name

Functional Logic Species



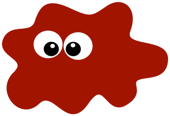
```
coin :: Int
coin = 0
coin = 1
```

call-by-value

```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
```

call-by-name

Functional Logic Species



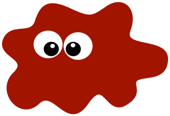
```
coin :: Int
coin = 0
coin = 1
```

call-by-value

```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

call-by-name

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

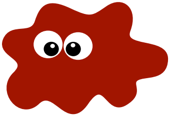
call-by-value

```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

$\Longrightarrow$ call-time choice

call-by-name

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-value

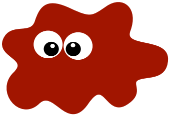
```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

⟹ call-time choice

call-by-name

```
double coin = coin+coin
```

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-value

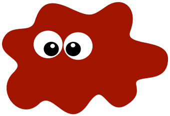
```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

$\Longrightarrow$ call-time choice

call-by-name

```
double coin = coin+coin
            = (0|1)+(0|1)
```

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-value

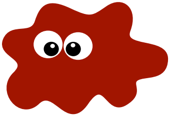
```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

$\Longrightarrow$ call-time choice

call-by-name

```
double coin = coin+coin
            = (0|1)+(0|1)
            = ... = 0 | 1 | 1 | 2
```

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-value

```
double coin = double (0|1)
            = double 0 | double 1
            = 0+0 | 1+1
            = 0 | 2
```

$\Longrightarrow$ call-time choice

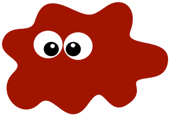
call-by-name

```
double coin = coin+coin
            = (0|1)+(0|1)
            = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

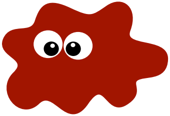
Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```



call-by-need

Functional Logic Species

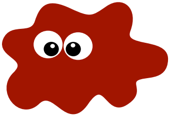


```
coin :: Int
coin = 0
coin = 1
```

call-by-need

```
double coin = let x=coin in x+x
```

Functional Logic Species

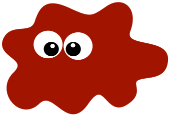


```
coin :: Int
coin = 0
coin = 1
```

call-by-need

```
double coin = let x=coin in x+x
            = let x=0|1 in x+x
```

Functional Logic Species



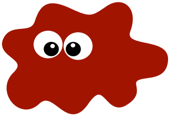
```
coin :: Int
coin = 0
coin = 1
```

call-by-need

```
double coin = let x=coin in x+x
             = let x=0|1 in x+x
             = (0|1) + (0|1)
```

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```

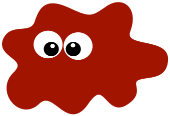


call-by-need

```
double coin = let x=coin in x+x
            = let x=0|1 in x+x
            = (0|1) + (0|1)
            = ... = 0 | 1 | 1 | 2
```

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```

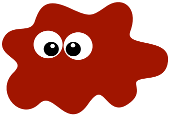


call-by-need

```
double coin = let x=coin in x+x
             = let x=0|1 in x+x
             = (0|1) + (0|1)
             = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

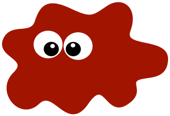
call-by-need

```
double coin = let x=coin in x+x
            = let x=0|1 in x+x
            = (0|1) + (0|1)
            = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

```
double coin = let x=0|1 in x+x
```

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-need

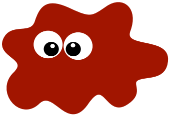
```
double coin = let x=coin in x+x
             = let x=0|1 in x+x
             = (0|1) + (0|1)
             = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

```
double coin = let x=0|1 in x+x
             = let x=0 in x+x | let x=1 in x+x
```

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```



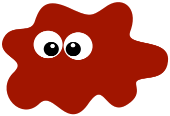
call-by-need

```
double coin = let x=coin in x+x
             = let x=0|1 in x+x
             = (0|1) + (0|1)
             = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

```
double coin = let x=0|1 in x+x
             = let x=0 in x+x | let x=1 in x+x
             = 0+0 | 1+1
```

Functional Logic Species



```
coin :: Int
coin = 0
coin = 1
```

call-by-need

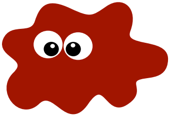
```
double coin = let x=coin in x+x
            = let x=0|1 in x+x
            = (0|1) + (0|1)
            = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

```
double coin = let x=0|1 in x+x
            = let x=0 in x+x | let x=1 in x+x
            = 0+0 | 1+1
            = 0 | 2
```

Functional Logic Species

```
coin :: Int
coin = 0
coin = 1
```



call-by-need

```
double coin = let x=coin in x+x
             = let x=0|1 in x+x
             = (0|1) + (0|1)
             = ... = 0 | 1 | 1 | 2
```

$\Longrightarrow$ run-time choice

```
double coin = let x=0|1 in x+x
             = let x=0 in x+x | let x=1 in x+x
             = 0+0 | 1+1
             = 0 | 2
```

$\Longrightarrow$ call-time choice

Non-Determinism

```
insertnd :: a -> [a] -> [a]
insertnd x ys = x:ys
insertnd x (y:ys) = y:insertnd x ys
```

Non-Determinism

```
insertnd :: a -> [a] -> [a]
insertnd x ys = x:ys
insertnd x (y:ys) = y:insertnd x ys

Main> insertnd 3 [1,2]
```

Non-Determinism

```
insertnd :: a -> [a] -> [a]
insertnd x ys = x:ys
insertnd x (y:ys) = y:insertnd x ys

Main> insertnd 3 [1,2]
[1,2,3]
```

Non-Determinism

```
insertnd :: a -> [a] -> [a]
insertnd x ys = x:ys
insertnd x (y:ys) = y:insertnd x ys

Main> insertnd 3 [1,2]
[1,2,3]
[1,3,2]
```

Non-Determinism

```
insertnd :: a -> [a] -> [a]
insertnd x ys = x:ys
insertnd x (y:ys) = y:insertnd x ys

Main> insertnd 3 [1,2]
[1,2,3]
[1,3,2]
[3,1,2]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
[3,1,2]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
[3,1,2]
[2,3,1]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
[3,1,2]
[2,3,1]
[2,1,3]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
[3,1,2]
[2,3,1]
[2,1,3]
[1,3,2]
```

Non-Determinism

```
permute :: [a] -> [a]
permute [] = []
permute (x:xs) = insertnd x (permute xs)

Main> permute [1..3]
[3,2,1]
[3,1,2]
[2,3,1]
[2,1,3]
[1,3,2]
[1,2,3]
```

Non-Determinism

```
sort :: [a] -> [a]
sort xs | ordered ys = ys
 where
  ys = permute xs
```

Non-Determinism

```
sort :: [a] -> [a]
sort xs | ordered ys = ys
 where
  ys = permute xs

Main> sort [3,2,1]
```

Non-Determinism

```
sort :: [a] -> [a]
sort xs | ordered ys = ys
 where
  ys = permute xs

Main> sort [3,2,1]
[1,2,3]
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free

Main> list
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free

Main> list
[1]
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free

Main> list
[1]
[_a,1]
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free

Main> list
[1]
[_a,1]
[_a,_b,1]
```

Narrowing

```
list :: [Int]
list = ys ++ [1]
 where
  ys free

Main> list
[1]
[_a,1]
[_a,_b,1]
[_a,_b,_c,1]
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] =:= xs = y
 where
  ys, y free
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
 where
  ys, y free
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
 where
  ys, y free

Main> last [1..4]
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
 where
  ys, y free

Main> last [1..4]
4
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
 where
  ys, y free

Main> last [1..4]
4

Main> last [error "_|_",2]
```

Narrowing

```
last :: [a] -> a
last xs | ys ++ [y] == xs = y
 where
  ys, y free

Main> last [1..4]
4

Main> last [error "_|_",2]
ERROR: '_|_'
```

~~Narrowing~~ Non-Determinism

```
data Peano = Zero | Succ Peano
```

~~Narrowing~~ Non-Determinism

```
data Peano = Zero | Succ Peano

peano :: Peano
peano = Zero
peano = Succ peano
```

~~Narrowing~~ Non-Determinism

```
data Peano = Zero | Succ Peano

peano :: Peano
peano = Zero
peano = Succ peano

pList :: [Peano]
pList = []
pList = peano : pList
```

~~Narrowing~~ Non-Determinism

```
data Peano = Zero | Succ Peano

peano :: Peano
peano = Zero
peano = Succ peano

pList :: [Peano]
pList = []
pList = peano : pList

last :: [Peano] -> Peano
last xs | ys ++ [y] == xs = y
 where
  ys = pList
  y = peano
```

An Evolution Step

characteristics KiCS

- Kiel Curry System
- Bernd Braßel, Frank Huch
- non-determinism
- target language: Haskell
- search strategy: arbitrary, explicit searchtree
- On a Tighter Integration of Functional and Logic Programming, APLAS 2007

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)
```

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)

check :: (a -> b -> Bool) -> (a,b)
check prop | not (prop x y) = (x,y)
  where
   x, y free
```

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)

check :: (a -> b -> Bool) -> (a,b)
check prop | not (prop x y) = (x,y)
  where
   x, y free

Main> check prop_Insert
```

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)

check :: (a -> b -> Bool) -> (a,b)
check prop | not (prop x y) = (x,y)
  where
   x, y free

Main> check prop_Insert
(Zero,(Succ _a:Zero:_b))
```

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)

check :: (a -> b -> Bool) -> (a,b)
check prop | not (prop x y) = (x,y)
  where
   x, y free

Main> check prop_Insert
(Zero,(Succ _a:Zero:_b))
(Succ Zero,(Succ _a:Zero:_b))
```

Killer Application

```
prop_Insert :: Peano -> [Peano] -> Bool
prop_Insert p ps = ordered (insert p ps)

check :: (a -> b -> Bool) -> (a,b)
check prop | not (prop x y) = (x,y)
  where
   x, y free

Main> check prop_Insert
(Zero,(Succ _a:Zero:_b))
(Succ Zero,(Succ _a:Zero:_b))
⋮
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]
```

```
{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{ _ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], _:_ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], { _ }:_ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], {Zero, Succ _ }:_ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], {Zero, Succ { _ }}:_ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], {Zero, Succ {Zero, ...}}:_ }
```

Haskell ND vs. Curry ND

Haskell ND

```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], {Zero, Succ {Zero, ...}}:{ _ }}
```

Haskell ND vs. Curry ND

Haskell ND

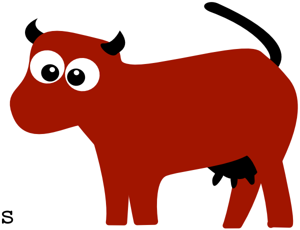
```
data Peano = Zero | Succ Peano
data [a] = [] | a : [a]

{[], [Zero], [Zero,Succ Zero], [Succ Zero,Zero], ...}
```

Curry ND

```
data Peano = Zero | Succ {Peano}
data [a] = [] | {a} : {[a]}

{[], {Zero, Succ {Zero, ...}}:{[], ...}}
```

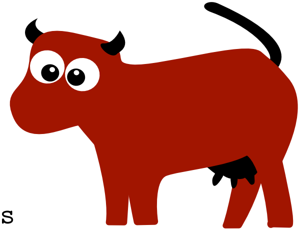
characteristics explicit-sharing

- library, available via hackage
- Sebastian Fischer, Oleg Kiselyov, Chung-chieh Shan
- non-determinism
- "target language": monadic Haskell
- search strategy: arbitrary, any `MonadPlus`
- Purely Functional Lazy Non-deterministic Programming, ICFP 2009



characteristics explicit-sharing

- library, available via hackage
- Sebastian Fischer, Oleg Kiselyov, Chung-chieh Shan
- non-determinism
- "target language": monadic Haskell
- search strategy: arbitrary, any `MonadPlus`
- Purely Functional Lazy Non-deterministic Programming, ICFP 2009



this is not the end of evolution!

Advertisement

Curry

The Flavor in Your Programming

PAKCS `www.informatik.uni-kiel.de/~pakcs`

KiCS `www.informatik.uni-kiel.de/prog/mitarbeiter/bernd-brassel/projects`

explicit sharing `sebfisch.github.com/explicit-sharing`

give it a try!

have fun!

have logical fun!