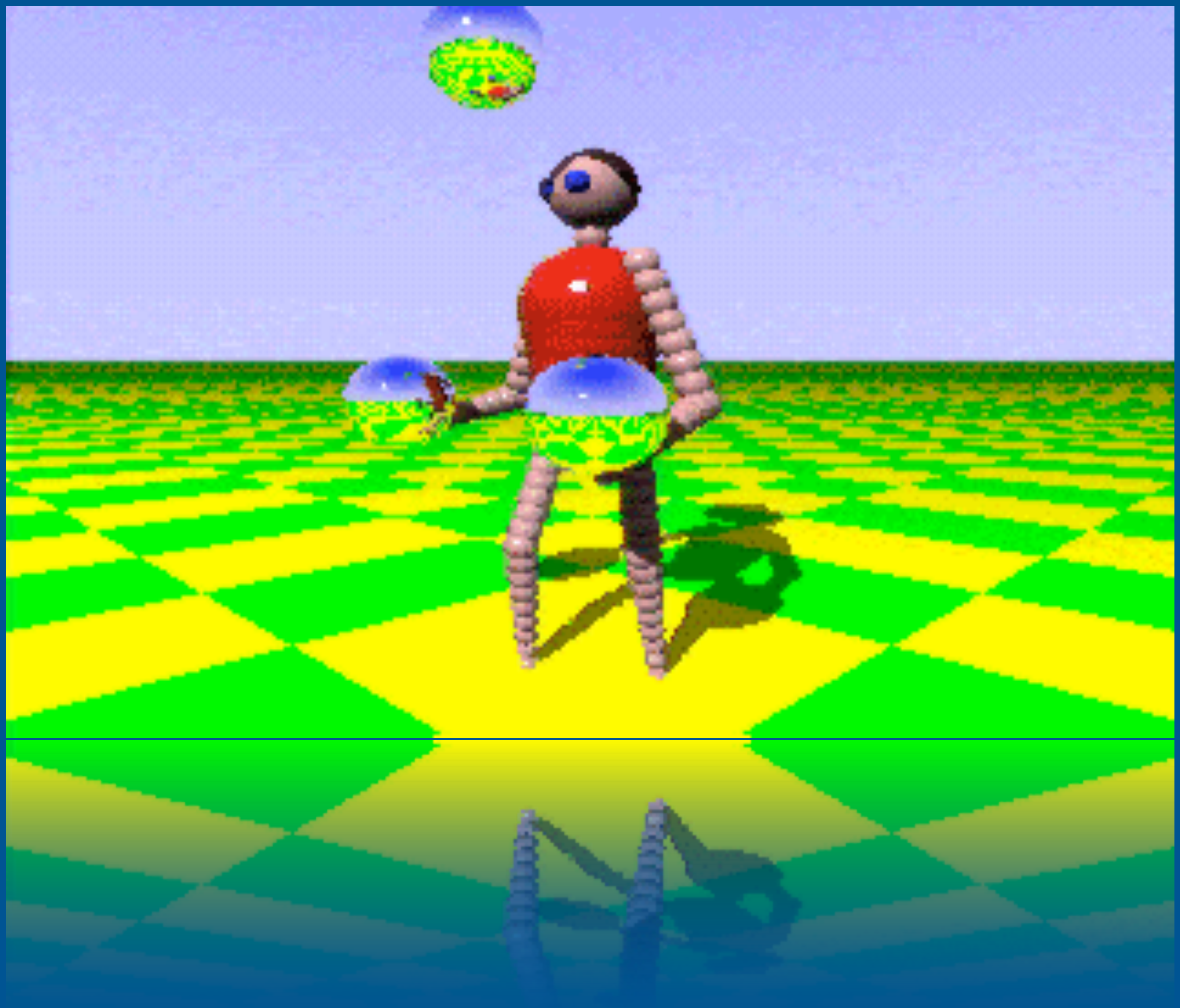
Memoisation and Hylomorphism



"Memo" Functions

"It would be useful if computers could learn from experience and thus automatically improve the efficiency of their own programs during execution." Donald Michie

Created recursive functions for:

- Factorial
- Highest Common Factor
- Member
- Reverse

Memoisation

“Memo” Functions

We define the rule part of factorial as follows:

if n < 0 or if not (n. isinteger) then undef else
*if n = 0 then 1 else n * fact (n - 1) close*
end

To endow *fact* with the “memo” facility, using Popplestone’s routines, we merely write

newmemo (fact, 100, nonop =) → fact;

...rote has an upper fixed limit of 100 entries...the symbol nonop warns the machine not to try to operate the “=” function at this stage...

Memoisation

Ruby Factorial

```
def fact(n)
  n < 1 ? 1 : n * fact(n - 1)
end

require 'method_decorators/decorators/memoize'
+Memoize
def ffact(n)
  n < 1 ? 1 : n * ffact(n - 1)
end
```

Memoisation

Coffeescript Factorial

```
fact = (n) ->
  if n < 1 then 1 else n * fact(n - 1)

memos = []
ffact = (n) ->
  memos[n] ?=
    if n < 1
      1
    else
      n * ffact(n - 1)
```

Memoisation

Coffeescript Factorial

```
ffact:
  do ->
    memos = {}
    (n) ->
      memos[n] ?=
        if n < 1
          1
        else
          n * ffact(n - 1)
```

Memoisation

Haskell Factorial

```
fac :: Int -> Int
fac 0 = 0
fac 1 = 1
fac n = n * fac (n - 1)

facs :: [Int]
facs = scanl (*) 1 [1..]
ffac n = facs !! n
```

Memoisation

Fibonacci Example

Named after a guy called Fibonacci, well actually Leonardo of Paris.

$$F_n = F_{n-1} + F_{n-2}$$

Memoisation

Letters and Numbers

Based on the popular television series "Countdown" or French series "*La Conte est Bou*".

Given a list of numbers: 2, 5, 8, 10, 11, 17, 24, 50

Target: 53280

Use: +, *, /, -

Answer: 11 + 5 * 8 - 17 * 24 * 10 * 2

Memoisation

Ingredients

```
data Op = Add | Sub | Mul | Div
data Expr = Num Int | App Op Expr Expr
type Value = Int

subseqs [x] = [[x]]
subseqs (x:xs) = xss ++ [x] : map (x:) xss
        where xss = subseqs xs

value :: Expr -> Value
value (Num x) = x
value (App op e1 e2) = apply op (value e1)
  (value e2)
```

Memoisation

Ingredients

```
legal :: Op -> Value -> Value -> Bool
legal Add v1 v2 = (v1 <= v2)
legal Sub v1 v2 = (v2 < v1)
legal Mul v1 v2 = (1 < v1) && (v1 <= v2)
legal Div v1 v2 = (1 < v2) && (v1 `mod` v2 == 0)
```

Ingredients

```
mkExprs :: [Int] -> [(Expr, Value)]
mkExprs [x] = [(Num x, x)]
mkExprs xs = [ev | (ys, zs) <- unmerges xs,
                   ev1 <- mkExprs ys,
                   ev2 <- mkExprs zs,
                   ev <- combine ev1 ev2]
```

Memoisation

Ingredients

```
unmerges :: [a] -> [([a], [a])]
unmerges [x, y] = [([x], [y])]
unmerges (x:xs) = [([x], xs)] ++
  concatMap (add x) (unmerges xs)
    where
      add x (ys, zs) = [(x : ys, zs), (ys, x : zs)]
```

Adding Memoisation

```
data Trie a = Node a [(Int, Trie a)]
type Memo = Trie [(Expr, Value)]

memoise :: [[Int]] -> Memo
memoise = foldl insert empty
insert memo xs = store xs (mkExprs memo xs) memo
```

Memoisation

Adding Memoisation

```
mkExprs :: Memo -> [Int] -> [(Expr, Value)]
mkExprs memo [x] = [(Num x, x)]
mkExprs memo xs = [ev | (ys, zs) <- unmerges xs,
                        ev1 <- fetch memo ys,
                        ev2 <- fetch memo zs,
                        ev <- combine ev1 ev2]
```

Memoisation

Skeleton Tree

```
data Trie a = Node a [(Int, Trie a)]
data Tree = Tip Int | Bin Tree Tree
type Memo = Trie [Tree]

memoise :: [[Int]] -> Memo
memoise = foldl insert empty
insert memo xs = store xs (mkTrees memo xs)
 memo
```

Three Examples

What’s Going On?

Hypothesis 1:

Haskell Optimisations, GC

Hypothesis 2:

Cost of Memoisation > Recalculation

Memoisation

Results

PROBLEM	STANDARD	TRIE	SKELETON TREE
1,3,7,10,11,12, 14,50 → 12831	0.286079s	13.618973s	13.869919s
2,3,7,10,12,19, 24,50 → 53280	1.998651s	28.124236s	28.863705s
2,5,8,10,11,17, 24,50 → 53280	1.425602s	7.130803s	7.416698s

Problems

Initialising

Cache Miss

Extra Resources

Memoisation

Recalculate Intead?

Littles Law

Concurrency = Bandwidth * Latency

	1975	**2010**	**Improvement**
CPU Speed	3Mhz	3Ghz	x1000
Memory Bandwidth	~10Mb/sec	~10Gb/sec	x1000
Memory Latency	200ns	10ns	x20

Hylomorphisms

A computation that consists of the duals fold and unfold.

$$hylo = fold\ f\ g\ .\ unfold\ p\ v\ h$$

Memoisation

Catamorphism Lists

```
prod [] = 1
prod (x:xs) = x * prod xs

type ListCata x u = (u, x -> u -> u)
listCata :: ListCata x u -> [x] -> u
listCata (a, f) = cata where
        cata [] = a
        cata (x:xs) = f x (cata xs)

prod2 = listCata (1, (*))
rev = listCata ([], (\a b -> b ++ [a]))
```

Anamorphism Lists

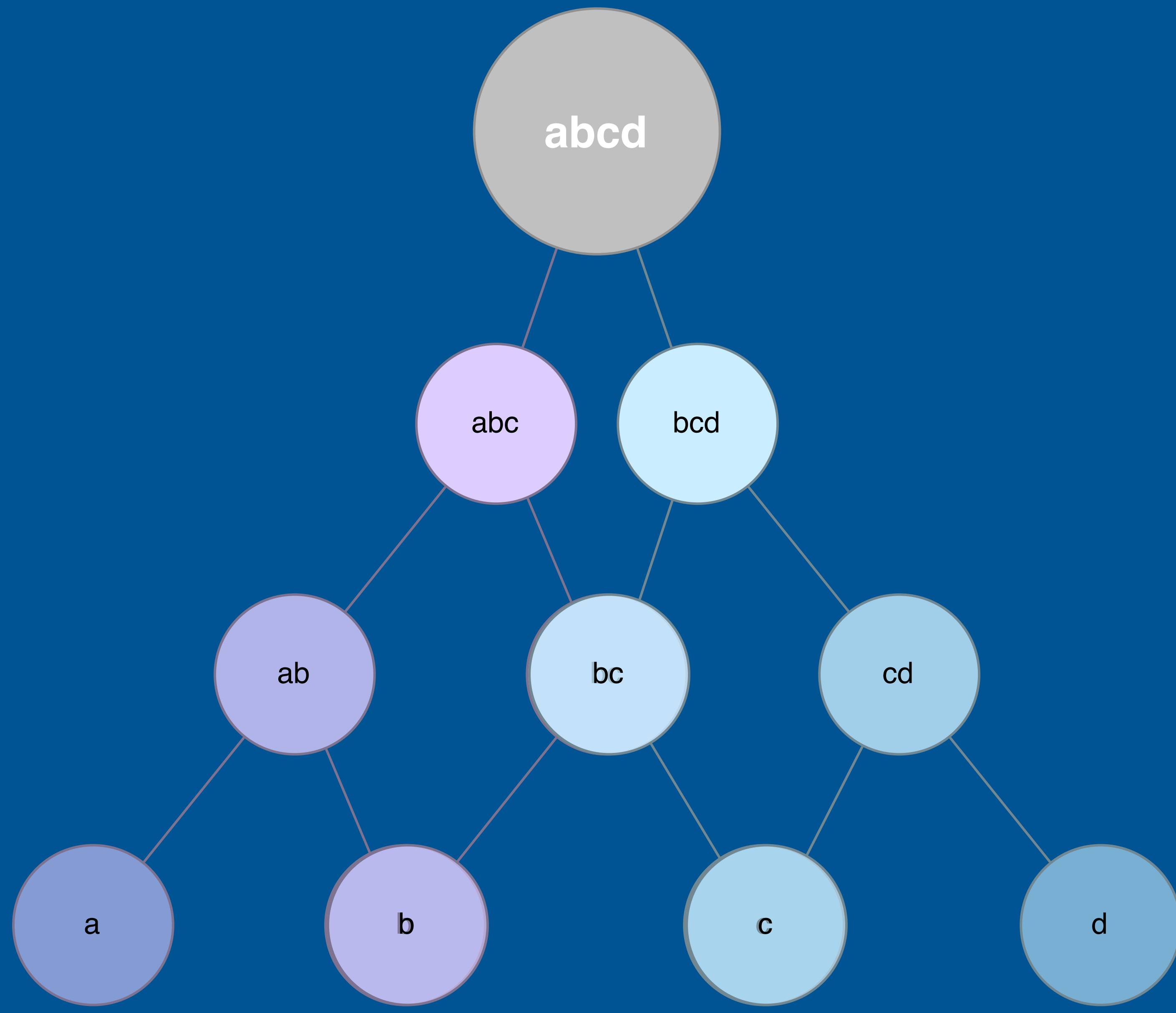
```
count 0 = []
count n  = n : count (n - 1)

type ListAna u x = u -> Either () (x, u)
listAna :: ListAna u x -> u -> [x]
listAna a = ana where
  ana u = case a u of
    Left _ -> []
    Right (x, xs) -> x : ana xs

count2 = listAna destructCount where
  destructCount 0 = Left ()
  destructCount n = Right (n, n - 1)
```

Memoisation

Nexus



Memoisation

Nexus

```
data Tree a = Leaf a | Node [Tree a]

unfold :: (b -> Bool) -> (b -> a) -> (b ->
[b]) -> b -> Tree a
unfold p v h x = if p x then Leaf (v x) else
  Node (map (unfold p v h) (h x))

mkTree :: ([a] -> [[a]]) -> [a] -> Tree [a]
mkTree h = unfold single id h
```

Memoisation

Nexus

```
data LTree a = LLeaf a | LNode a [LTree a]

treeToNexus :: Tree [a] -> LTree [a]
treeToNexus = fill id recover

fill :: (a -> b) -> ([b] -> b) -> Tree a -> LTree b
fill f g = fold (lleaf f) (lnode g)

fold :: (a -> b) -> ([b] -> b) -> Tree a -> b
fold f g (Leaf x) = f x
fold f g (Node ts) = g (map (fold f g) ts)
```

Memoisation

Nexus and Countdown

```
minors :: [a] -> [[a]]
minors [x, y] = [[x], [y]]
minors (x : xs) = map (x :) (minors xs) ++ [xs]

unmerges :: [a] -> [([a], [a])]
unmerges x = unmerge $ halved $
  [treeToNexus $ mkTree minors x]
```

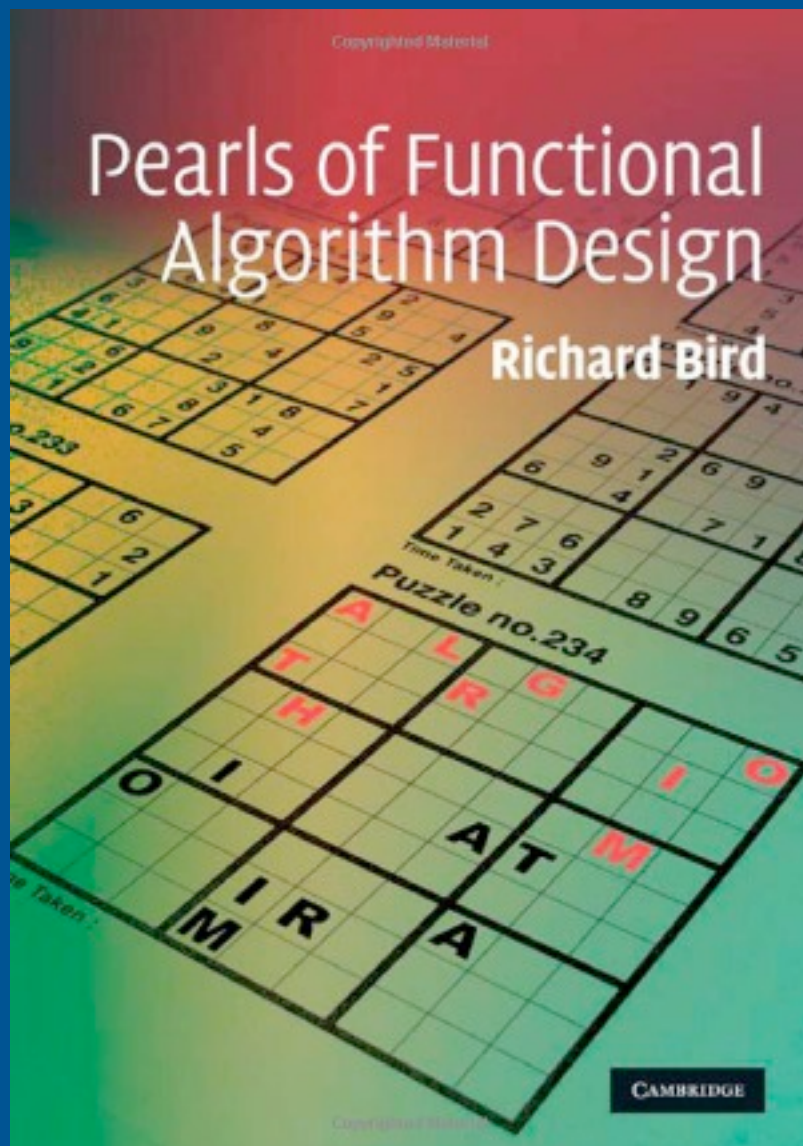
Memoisation

Results

PROBLEM	STANDARD	NEXUS
1,3,7,10,11,12, 14,50 → 12831	0.286079s	0.32372s
2,3,7,10,12,19, 24,50 → 53280	1.998651s	1.322064s
2,5,8,10,11,17, 24,50 → 53280	1.425602s	2.486383s

Memoisation

My Learning



Sorting morphisms

Lex Augusteijn

Philips Research Laboratories, Eindhoven
Email: `lex@natlab.research.philips.com`

Abstract. Sorting algorithms can be classified in many different ways. The way presented here is by expressing the algorithms as functional programs and to classify them by means of their *recursion patterns*. These patterns on their turn can be classified as the natural recursion patterns that destruct or construct a given data-type, the so called *cata-* and *anamorphisms* respectively. We show that the selection of the recursion pattern can be seen as the major design decision, in most cases leaving no more room for more decisions in the design of the sorting algorithm. It is also shown that the use of alternative data structures may lead to new sorting algorithms.
This presentation also serves as a gentle, light-weight, introduction into the various morphisms.

1 Introduction

In this paper we present several well known sorting algorithms, namely *insertion sort*, *straight selection sort*, *bubble sort*, *quick sort*, *heap sort* and *merge sort* (see e.g. [Knu73, Wir76]) in a non-standard way. We express the sorting algorithms as functional programs that obey a certain pattern of recursion. We show that for each of the sorting algorithms, the recursion patterns forms the major design decision, often leaving no more space for additional decisions to be taken. We make these recursion patterns explicit in the form of higher-order functions, much like the well-known `map` function on lists.
In order to reason about recursion patterns, we need to formalize that notion. Such a formalization is already available, based on a category theoretical modeling of recursive data types as can e.g. be found in [Fok92, Mei92]. In [BdM94] this theory is presented together with its application to many algorithms, including selection sort and quicksort. These algorithms can be understood however only after absorbing the underlying category theory. There is no need to present that theory here. The results that we need can be understood by anyone having some basic knowledge of functional programming, hence we repeat only the main results here. These results show how to each recursive data type a number of morphisms is related, each capturing some pattern of recursion which involve the recursive structure of the data type. Of these morphisms, we use the so called *catamorphism*, *anamorphism*, *hylomorphism* and *paramorphism* on linear lists and binary trees. The value of this approach is not so much in obtaining a nice implementation of some algorithm, but in unraveling its structure.
This presentation gives the opportunity to introduce the various morphisms in a simple way, namely as patterns of recursion that are useful in functional programming, instead

1

Memoisation

Links

"'Memo' Function and Machine Learning"
http://www.cs.utexas.edu/users/hunt/research/hash-cons/hash-cons-papers/michie-memo-nature-1968.pdf

"Extending Ruby with Ruby" and method_decorators gem
https://github.com/newhavenrb/conferences/wiki/Extending-Ruby-with-Ruby
https://github.com/michaelfairley/method_decorators

"Memoized, the *practical* method decorator"
https://github.com/raganwald/homoiconic/blob/master/2012/09/memoize-the-practical-method-decorator.md
https://github.com/raganwald/method-combinators

Links

“Functional Pearl Trouble Shared is Trouble Halved”
http://www.cs.ox.ac.uk/ralf.hinze/publications/HW03.pdf

“Sorting Morphisms”
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.3315

Code and Examples:
https://github.com/newmana/memoization