In praise of Higher Order Functions and of some friends and heroes

Mary Sheeran, Chalmers

Higher order functions aren't just a part of the Haskell experience, they pretty much are the Haskell experience.

http://learnyouahaskell.com/higher-order-functions

1959

MAPLIST (L,f)

maplist (L,f) constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L. The address portion of the element of the new list at J, corresponding to the element at L contains f(car(L)). The value of maplist is the address of the new list.

a) "fast" maplist

```
maplist(L,f)=/L=0→return(0)
                maplist = cons(f(L),0)
                M = maplist
            al  L = cdr (L)
                cdr(M) = cons(f(L),0)
                cdr(L) = 0→return(maplist)
                M = cdr(M)
               \go(al)
```

b) "slow" maplist

```
maplist(L,f) =(L=0→0,1→cons(f(L),maplist(cdr(L),f)))
```

Status: Both maplists have been checked out. In compiling, the fast maplist is used, as it saves about 1.3 milliseconds per list element of L. (75 % saving)

March 3, 1959
Author: J. McCarthy

Modification number 7
Makes obsolete:

maplist (L,f) constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L. The address portion of the element of the new list at J, corresponding to the element at L contains f (car (L)).

```
        cdr(M) = cons(f(L),0)
        cdr(L) = 0→return(maplist)
        M = cdr(M)
       \go(a1)
```

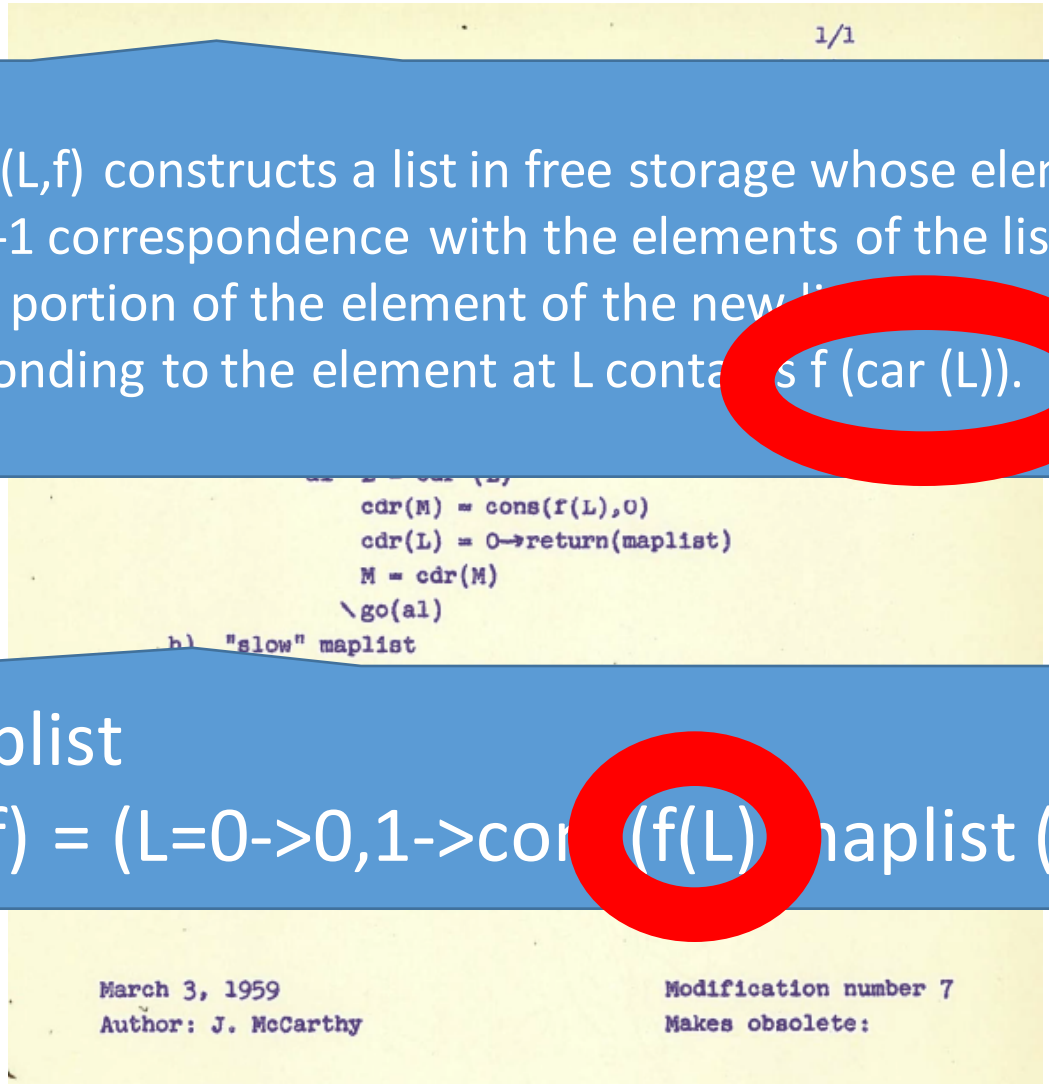
b) "slow" maplist

"slow" maplist

maplist (L,f) = (L=0->0,1->cons(f(L),maplist (cdr(L),f)))

March 3, 1959
Author: J. McCarthy

Modification number 7
Makes obsolete:


1/1
maplist (L,f) constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L. The address portion of the element of the new li
corresponding to the element at L conta s f (car (L)).
cdr(M) = cons(f(L),0)
cdr(L) = 0→return(maplist)
M = cdr(M)
\go(a1)
b) "slow" maplist
"slow" maplist
maplist (L,f) = (L=0->0,1->cor (f(L) aplist (cdr(L),f)))
March 3, 1959
Author: J. McCarthy
Modification number 7
Makes obsolete:

MAPLIST (L,f)

maplist (L,f) constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L. The address portion of the element of the new list at J, corresponding to the element at L contains f(L). The value of maplist is the address of the new list.

a) "fast" maplist

```
maplist(L,f)=/L=0→return(0)
                maplist=cons(f(L),0)
                M=maplist
             al L=cdr(L)
                cdr(M)=cons(f(L),0)
                cdr(L)=0→return(maplist)
                M=cdr(M)
               \go(al)
```

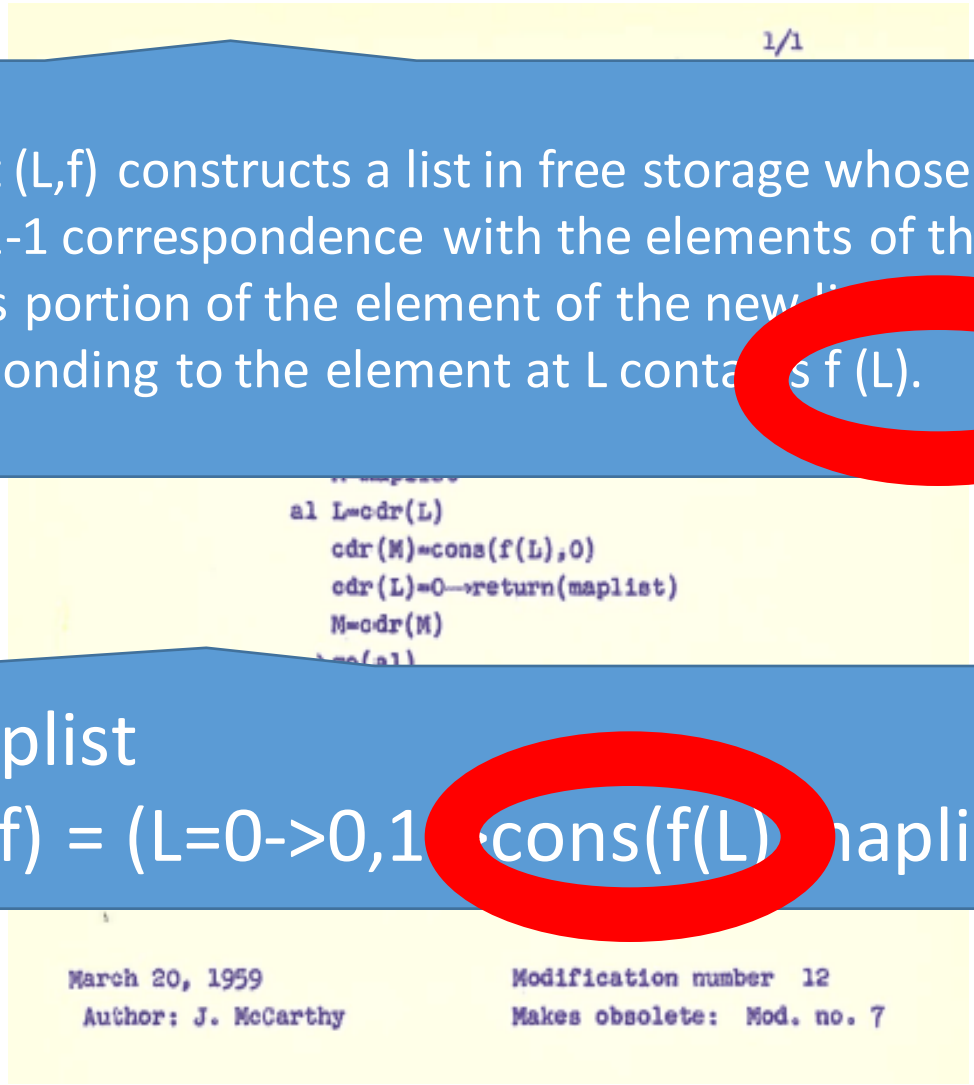
b) "slow maplist"

```
maplist(L,f)=(L=0→0,1→cons(f(L),maplist(cdr(L),f)))
```

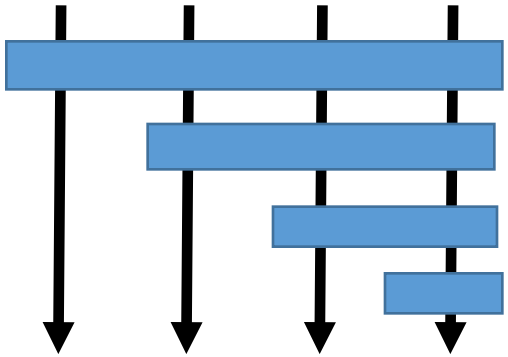
Status: Both maplists have been checked out. In compiling, the fast maplist is used, as it saves about 1.3 milliseconds per list element of L. (75% saving)

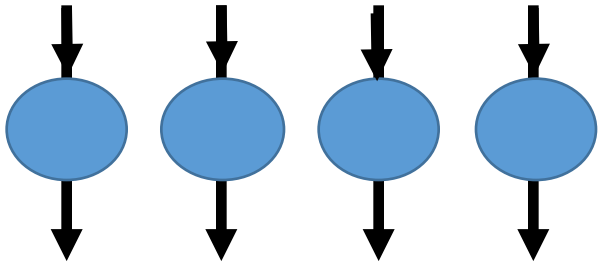
March 20, 1959
Author: J. McCarthy

Modification number 12
Makes obsolete: Mod. no. 7


1/1
maplist (L,f) constructs a list in free storage whose elements are in 1-1 correspondence with the elements of the list L. The address portion of the element of the new li
corresponding to the element at L conta s f (L).
al L=cdr(L)
cdr(M)=cons(f(L),0)
cdr(L)=0→return(maplist)
M=cdr(M)
"slow" maplist
maplist (L,f) = (L=0->0,1 cons(f(L) aplist (cdr(L),f)))
March 20, 1959
Author: J. McCarthy
Modification number 12
Makes obsolete: Mod. no. 7

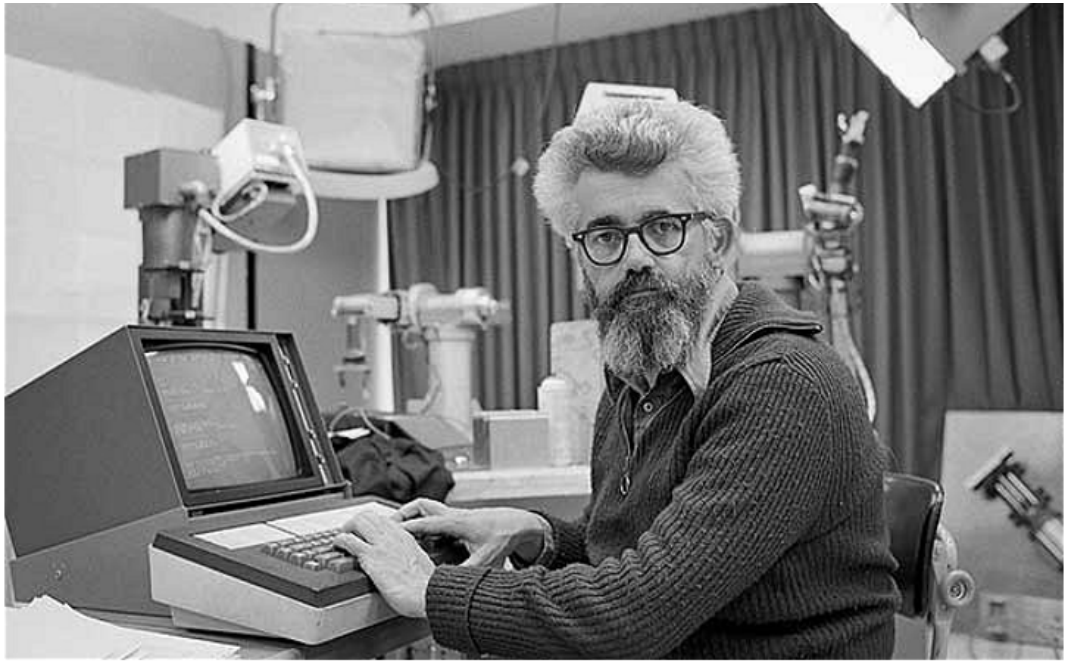
maplist wasn't the map we wanted





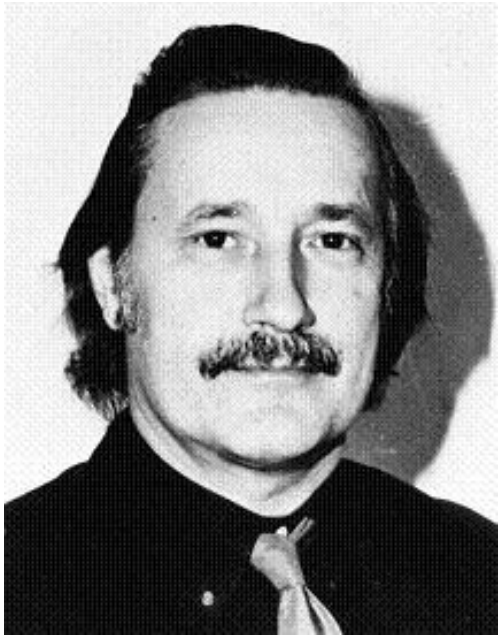
though maplist can be used to define map (called mapcar in LISP)

John McCarthy Turing Award 1971



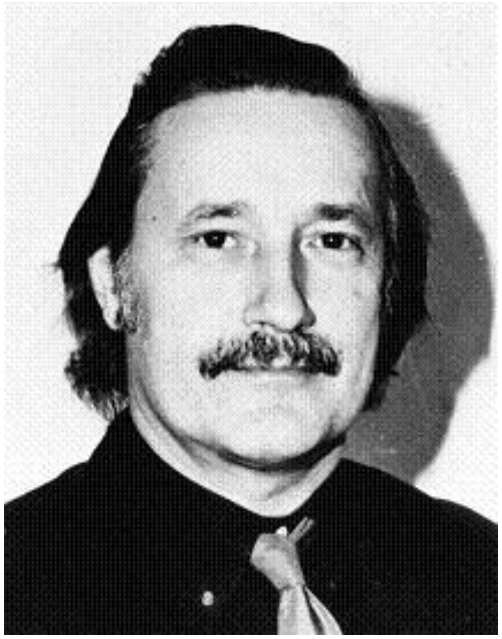
Ferranti Pegasus 1959
Science Museum London

Christopher Strachey



Renaissance Man

1961



David W. Barron and Christopher Strachey.

Programming.

In Leslie Fox, editor, Advances in Programming and NonNumerical Computation, pages 49–82. Pergammon Press, 1966.

Danvy & Spivey,
ICFP 2007

Map

```
let Map[f,L] = Null[L] -> NIL,
               Cons[f[Hd[L]], Map[f,Tl[L]]]
```

map

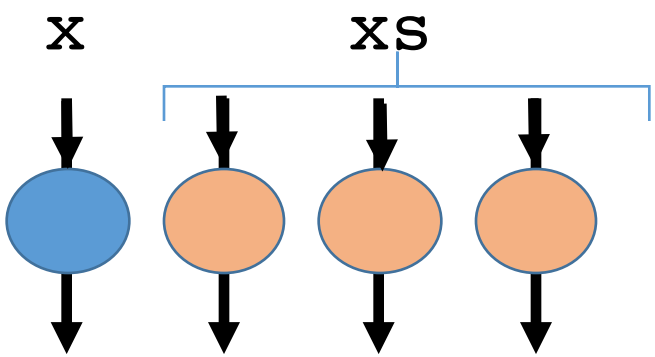
```
let Map[f,L] = Null[L] -> NIL,
               Cons[f[Hd[L]], Map[f,Tl[L]]]
```

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs
```

```
map :: (a -> b) -> [a] -> [b]
map f []          = []
map f (x:xs)      = f x : map f xs
```

*Main> map (*2) [1..8]
[2,4,6,8,10,12,14,16]

```
map f []          = []
map f (x:xs)      = f x : map f xs
```



List iteration (fold)

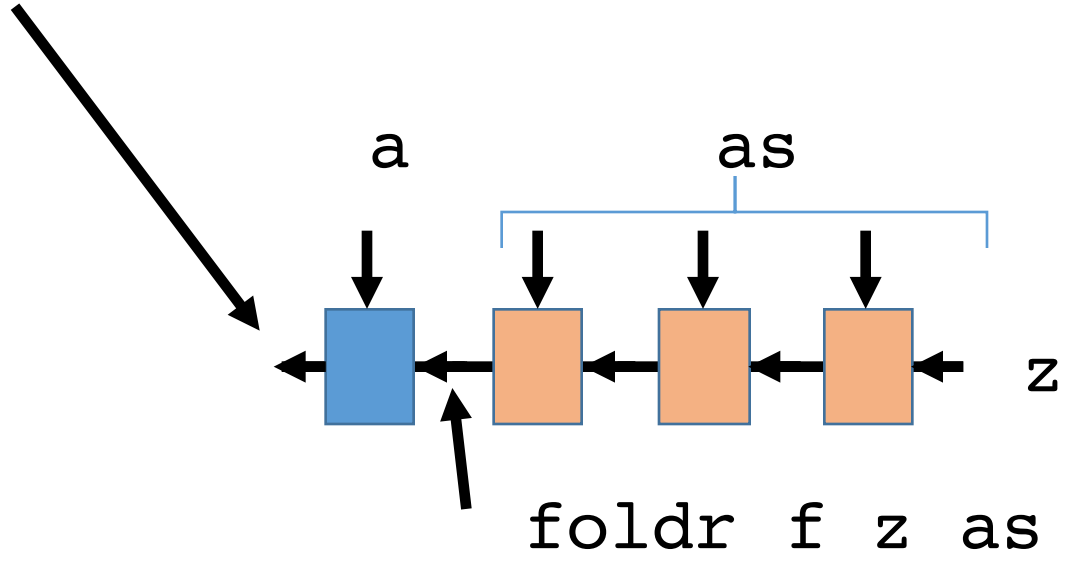
```
let Lit[F,z,L] = Null[L] -> z,
                 F[Hd[L], Lit[F,z,Tl[L]]]
```

List iteration (fold)

```
let Lit[F,z,L] = Null[L] -> z,
                 F[Hd[L], Lit[F,z,Tl[L]]]
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (a:as) = f a (foldr f z as)
```

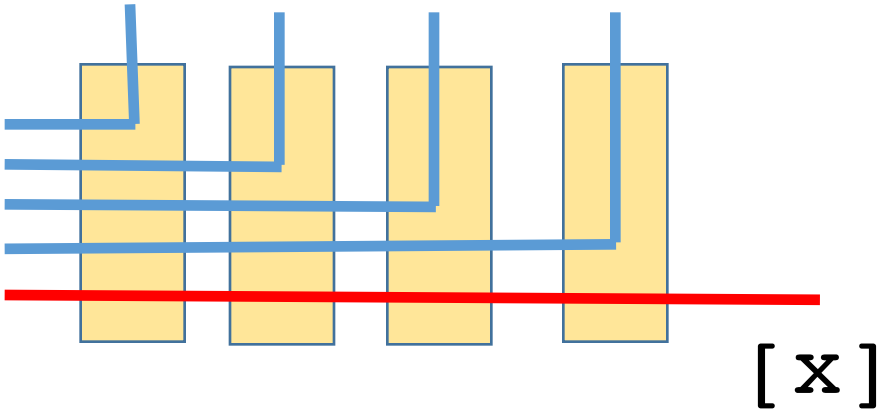
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (a:as) = f a (foldr f z as)
```

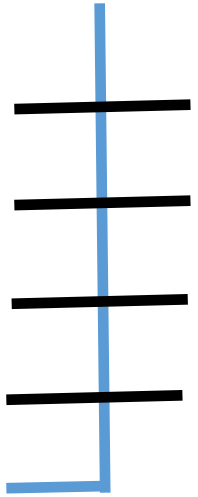


```
*Main> foldr (*) 1 [1..8]
40320
```

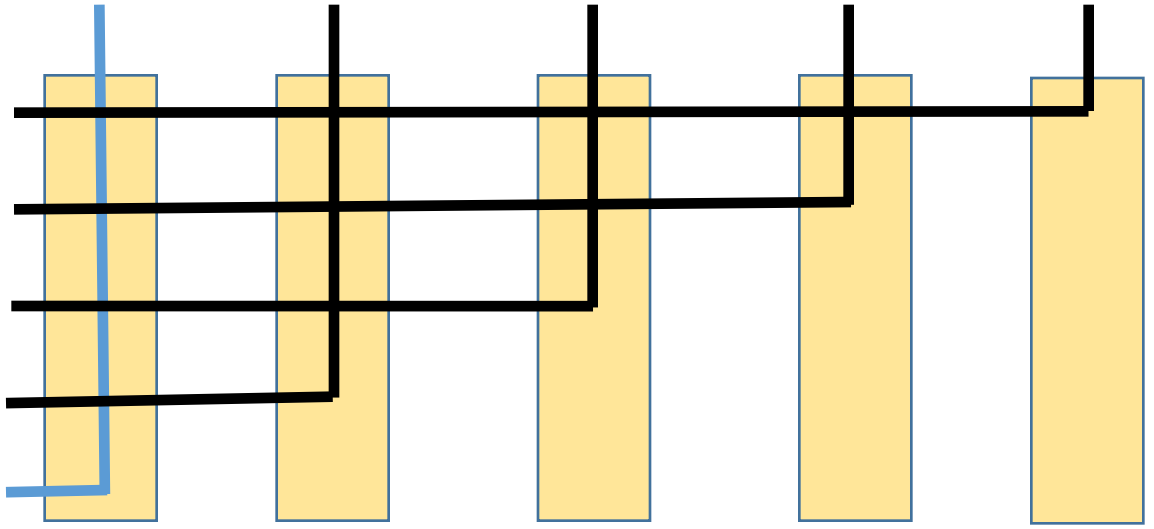
```
*Main> foldr (*) 1 [1..8]
40320

1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : []

1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 1
```

```
ap x ls = foldr (:) [x] ls
```





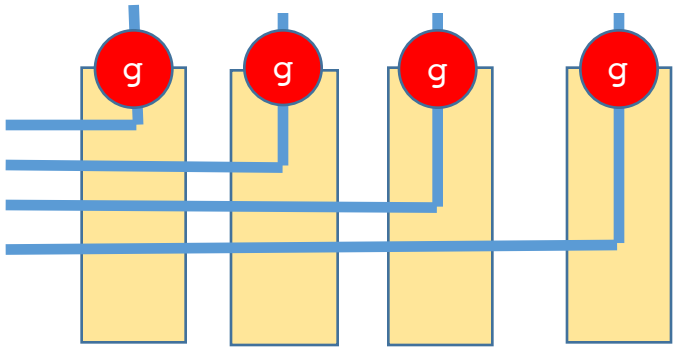
```
foldr ap [] ls
```



```
ap x l = foldr (:) [x] l

rev l  = foldr ap  []  l
```

*Main> rev [1..8]
[8,7,6,5,4,3,2,1]

```
map :: (t -> a) -> [t] -> [a]
map g l = foldr f [] l
   where
      f x y = g x : y
```



[]

```
mapa :: (t->a) -> [a] -> [t] -> [a]
mapa g z l = foldr f z l
   where
      f x y = g x : y
```

```
map g l ++ z
```

Cartesian product

```
Product [[a, b], [p, q, r], [x, y]]
```

```
[[a, p, x], [a, p, y], [a, q, x], [a, q, y], [a, r, x], [a, r, y],
 [b, p, x], [b, p, y], [b, q, x], [b, q, y], [b, r, x], [b, r, y]]
```

Barron and Strachey. 1966. Programming.

p0

```
product :: [[t]] -> [[t]]
product [ ]          = [[ ]]
product ([ ] : _) = [ ]
product ((x : xs) : xss)
  = map (x:) (product xss)
      ++ product (xs : xss)
```

p0

```
product :: [[t]] -> [[t]]
product [ ]          = [[ ]]
product ([ ] : _) = [ ]
product ((x : xs) : xss)
  = map (x:) (product xss)
        ++ product (xs : xss)
```

introduce `h xs xss = product (xs : xss)`

p1

```
product [ ]          = [[ ]]
product (xs : xss) = h xs xss
  where
    h [ ]       xss = [ ]
    h (x : xs) xss
      = map (x:) (product xss)
          ++ h xs xss
```

introduce `h xs xss = product (xs : xss)`

p1

```
product [ ]          = [[ ]]
product (xs : xss) = h xs xss
  where
    h [ ]       xss = [ ]
    h (x : xs) xss
      = map (x:) (product xss)
          ++ h xs xss
```

```
f xs (product xss) = h xs xss
```

replace h by f

p2

```
product [ ]          = [[ ]]
product (xs : xss)
  = f xs (product xss)
  where
    f [ ] yss        = [ ]
    f (x : xs) yss
      = map (x:) yss ++ f xs yss
```

```
f xs (product xss) = h xs xss
```

replace h by f

p2

```
product []          = [[]]
product (xs : xss)
  = f xs (product xss)
  where
    f [] yss = []
    f (x : xs) yss
      = map (x:) yss ++ f xs yss
```

```
f xs (product xss) = h xs xss
```

replace h by f

p3

```
product []          = [[]]
product (xs : xss)
  = f xs (product xss)
  where
    f [] yss = []
    f (x : xs) yss
      = mapa (x:) (f xs yss) yss
```

```
product []          = [[]]
product (xs : xss)
  = f xs (product xss)
  where
    f [] yss = []
    f (x : xs) yss
      = mapa (x:) (f xs yss) yss
```

```
product xss = foldr f [[]] xss
  where f xs yss = foldr g [] xs
    where g x zss = foldr h zss yss
      where h ys qss = (x : ys) : qss
```

1962

APL

An operation (such as summation) which is applied to all components of a vector to produce a result of a simpler structure is called a reduction. The $\odot$-reduction of a vector x is denoted by /x and defined as

$z \leftarrow /x \leftrightarrow z = (\ldots((x_1 \odot x_2) \odot x_3) \odot \ldots) \odot x_v)$,

where is any binary operator with a suitable domain.

http://www.jsoftware.com/papers/APL.htm


PF1
PF2
PF3
PF4
PF5
PF6
PF7
PF8
RESET
ALT
ENTER
DUP
ATTN
PF9
PF10
PF11
PF12
FIELD MARK
ERASE EOF
CURSR SEL
CURSR BLINK

Notation as a Tool of Thought

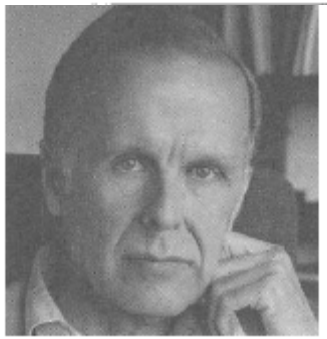
Kenneth E. Iverson
IBM Thomas J. Watson Research Center

Turing Award 1979

1977

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



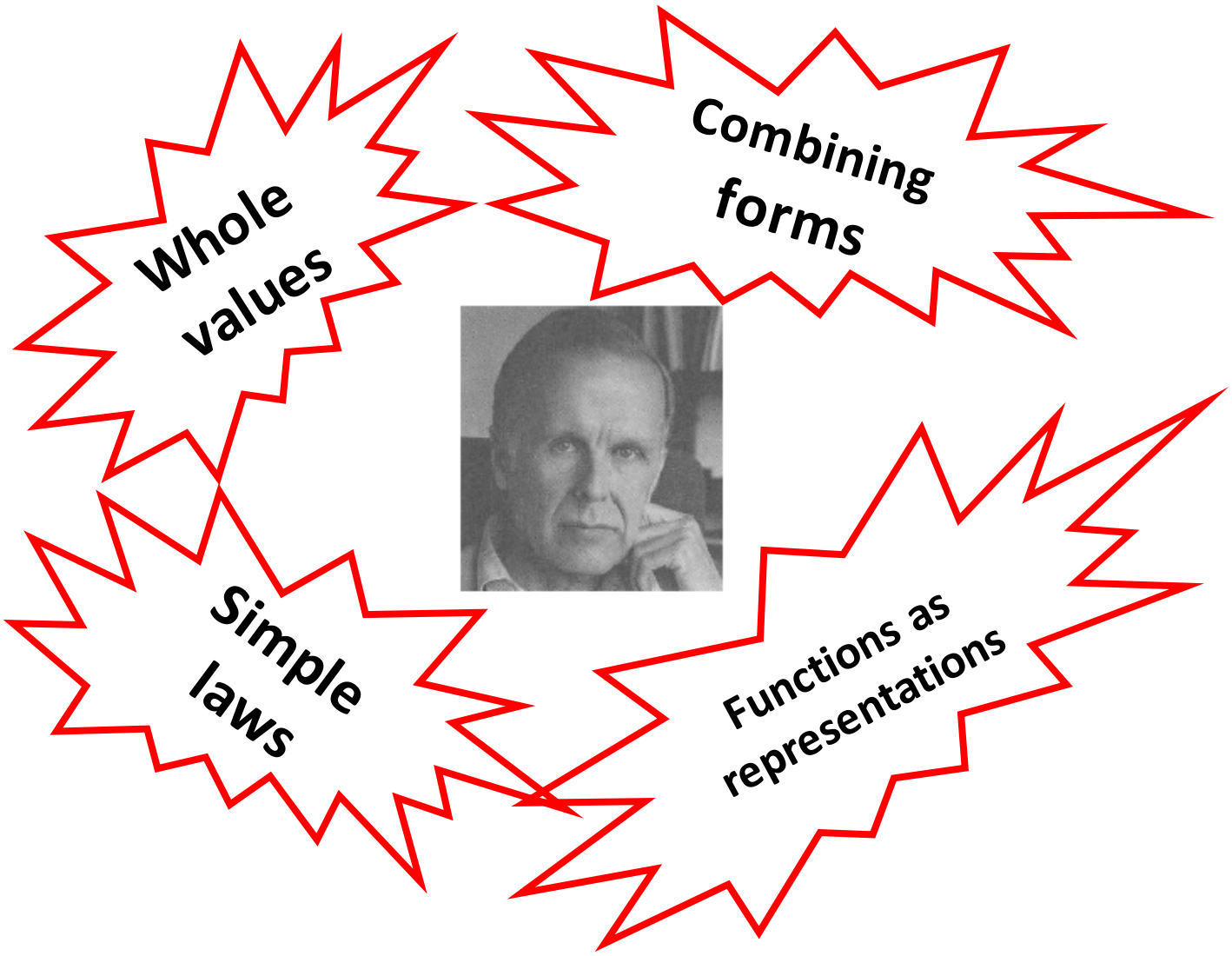
Turing award 1977

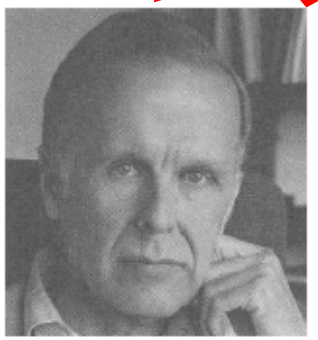
Paper 1978

Conventional programming languages are growing ever more enormous,
but not stronger.

Inherent defects at the most basic level cause them to be both fat and weak:

their inability to effectively use powerful combining forms for building new programs from existing ones


Whole values
Combining forms
Simple laws
Functions as representations



Whole values

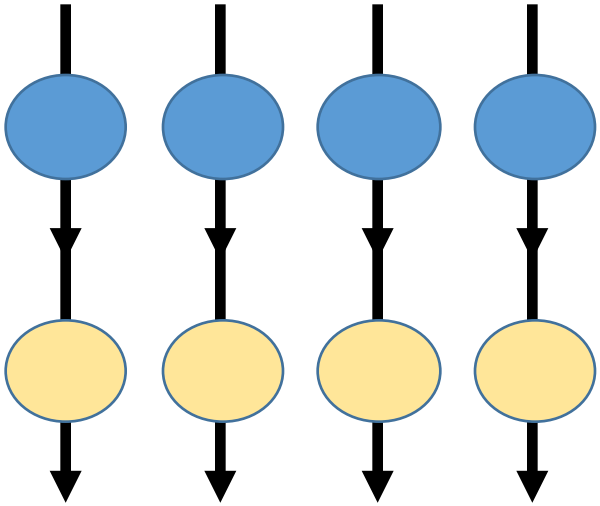
Combining forms

Simple laws

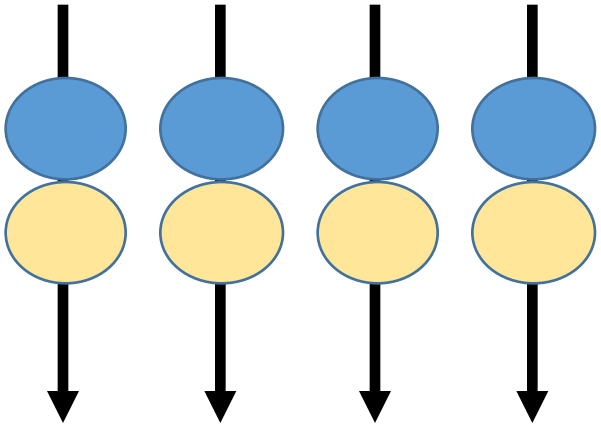
Functions as representations

Fish!
Room 1 15.00

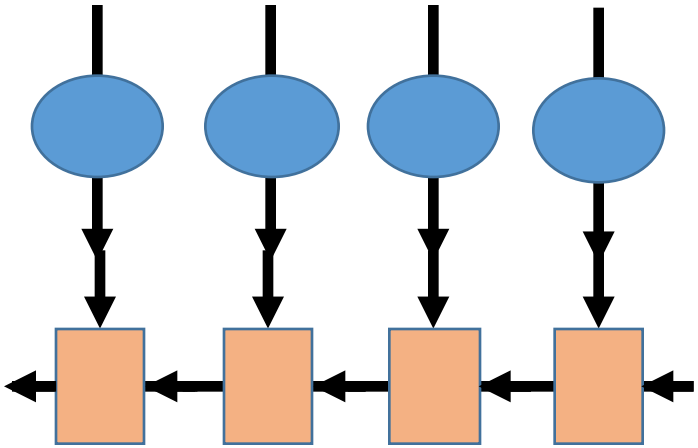
```
map f . map g
```



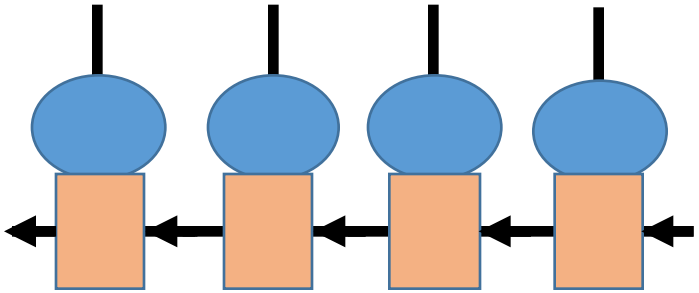
```
map (f . g)
```



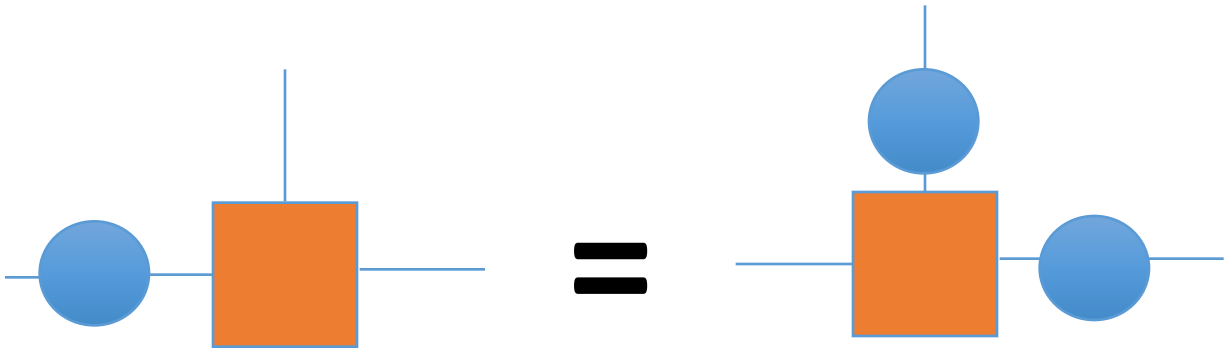
```
foldr f v (map g xs)
```

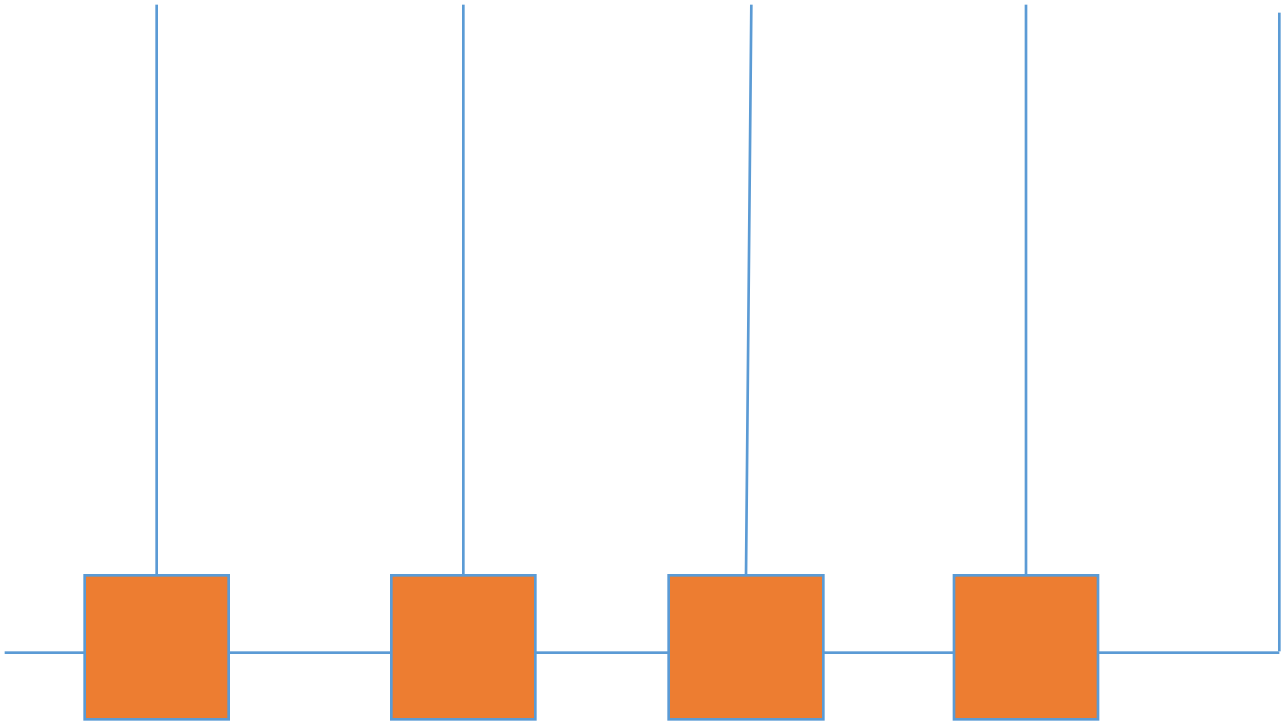


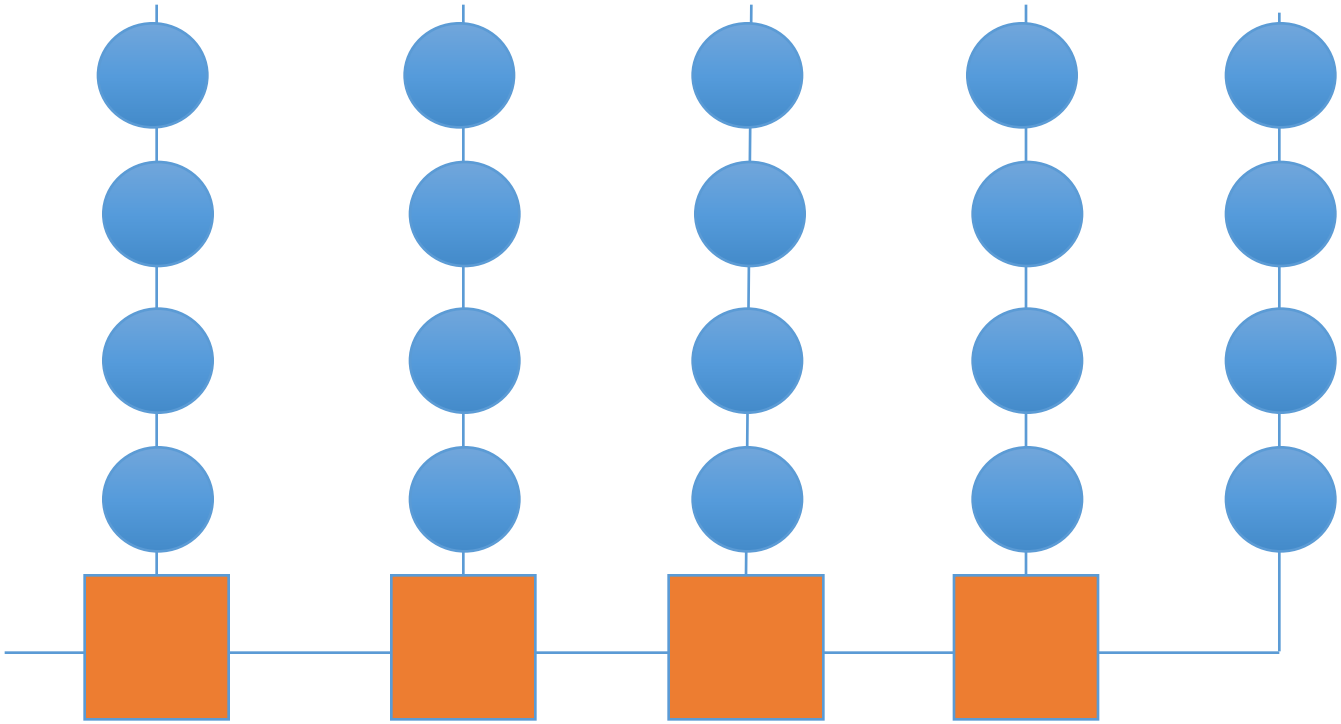
```
foldr (f.g) v xs
```

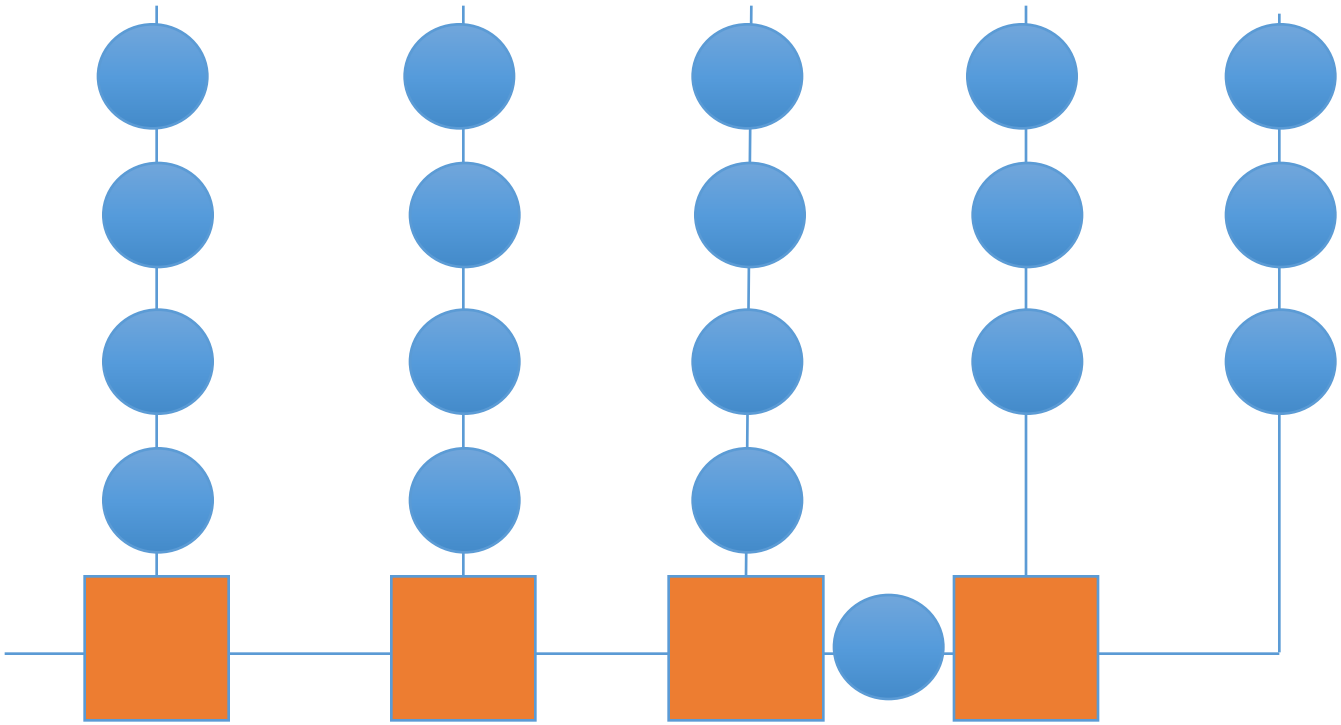


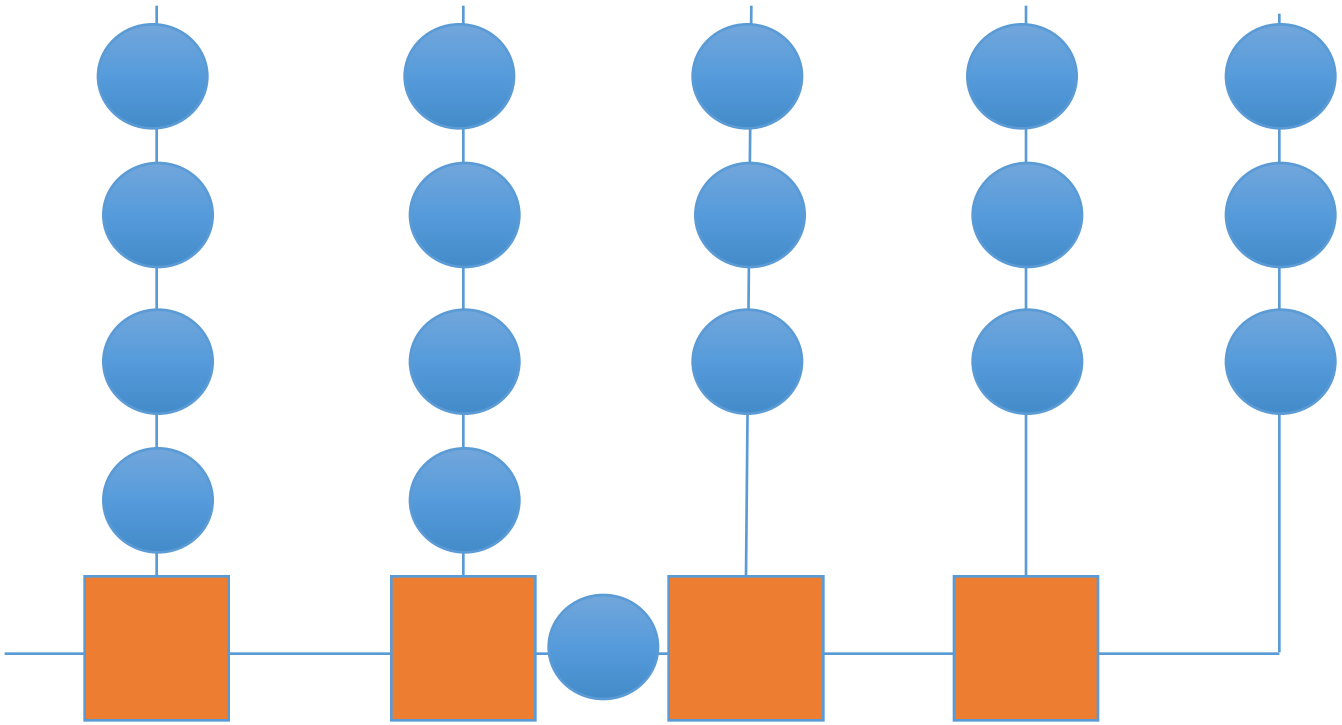
Example law: “retiming”

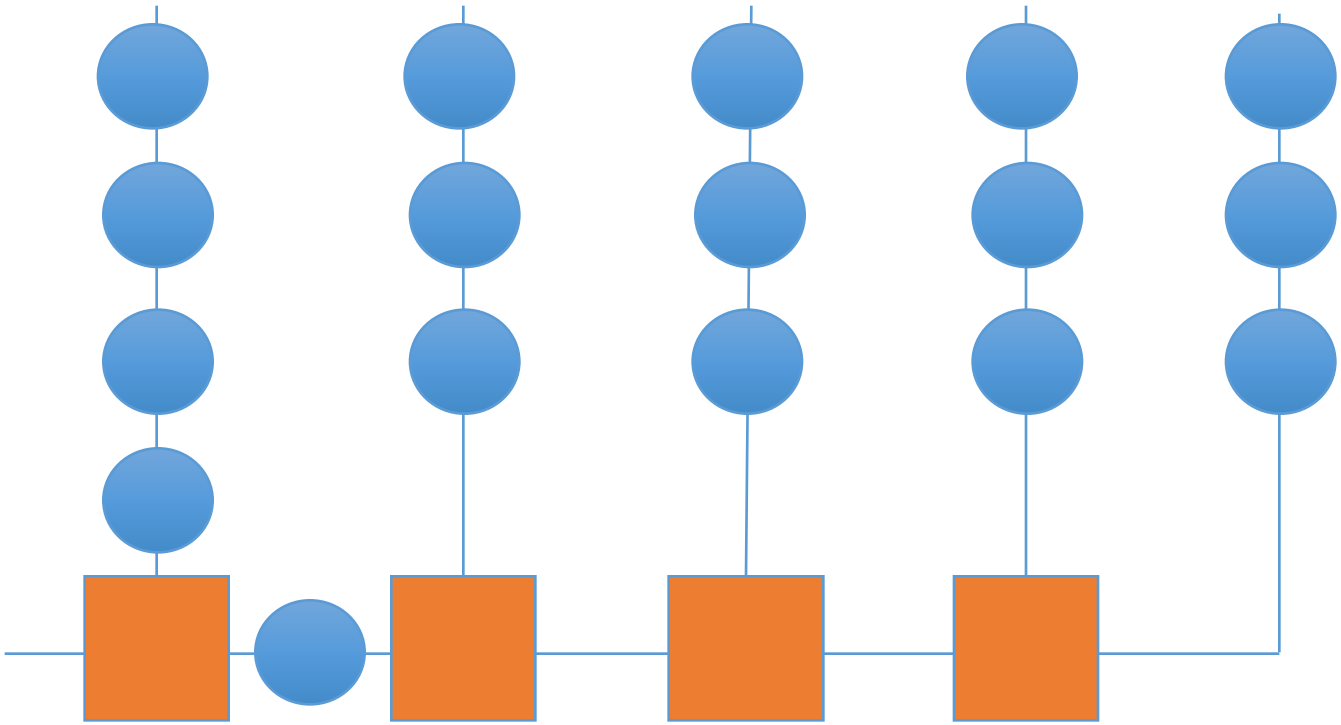


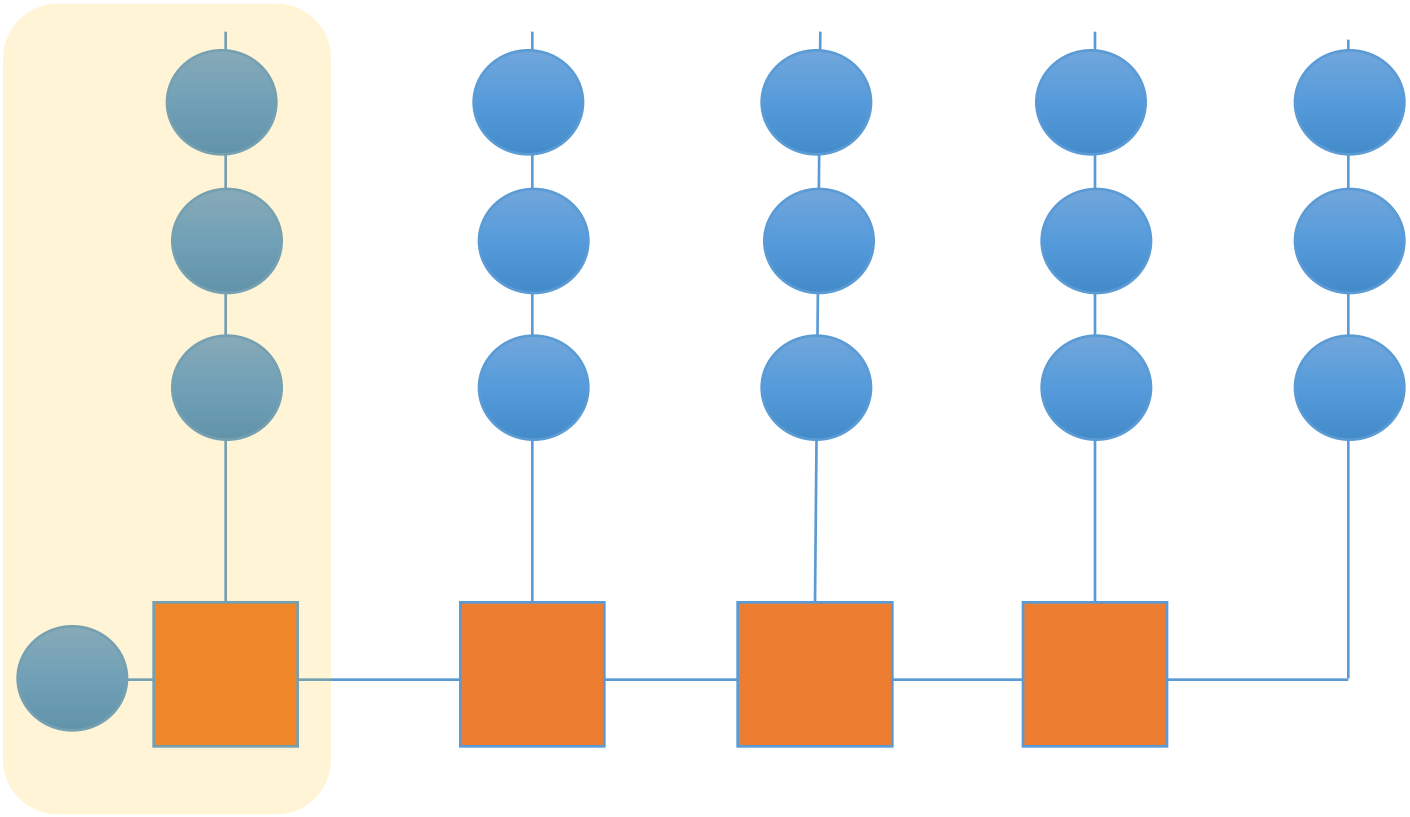


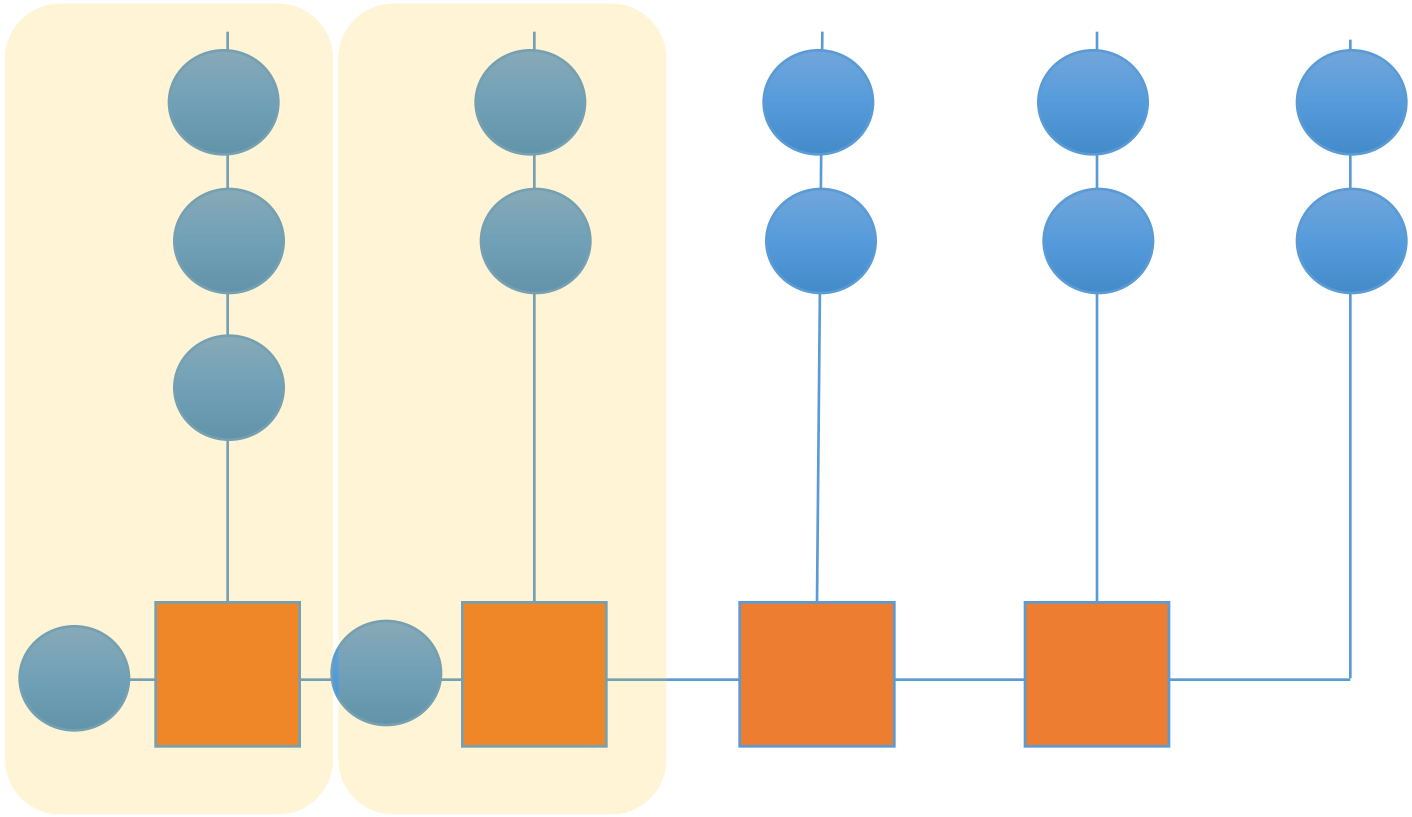


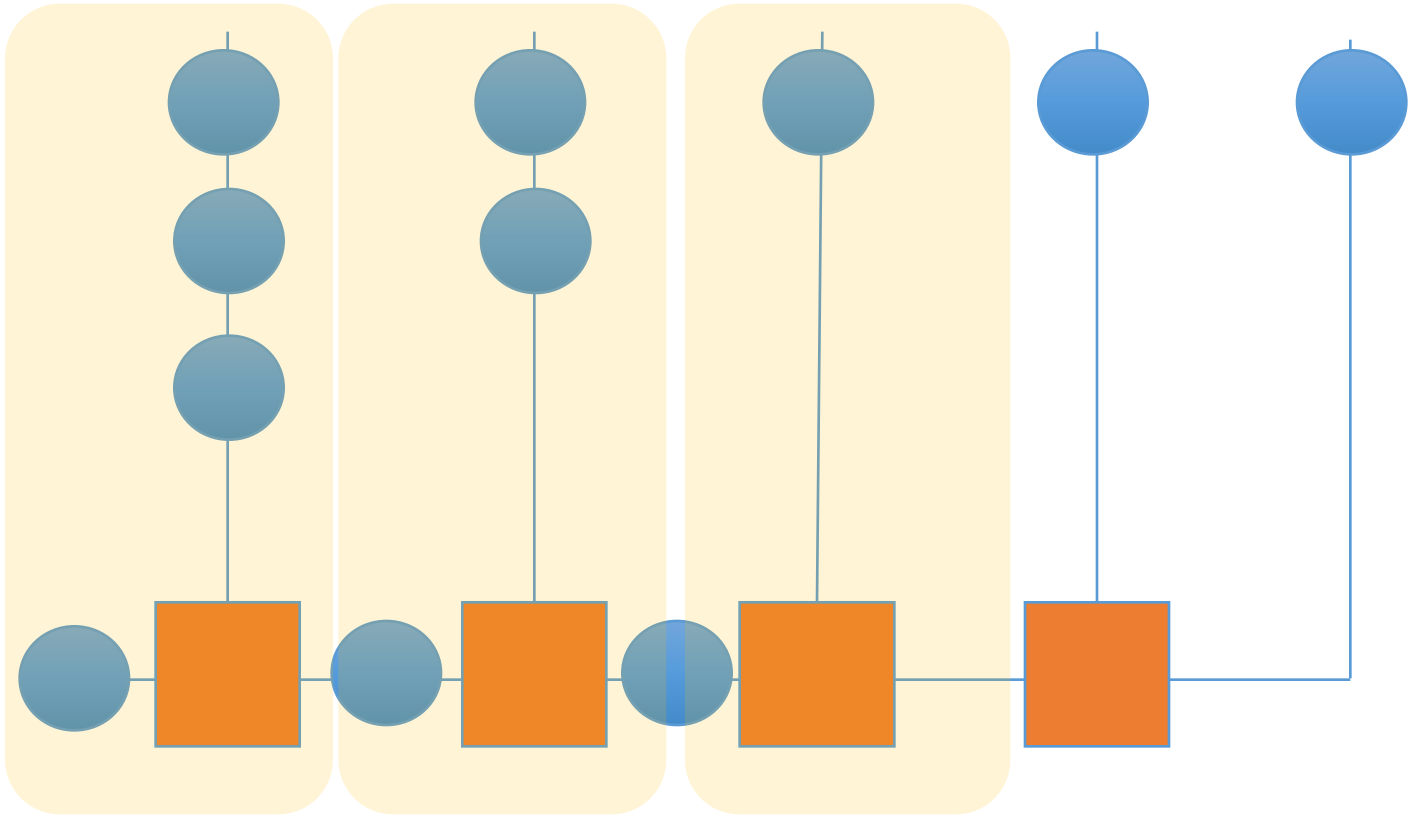


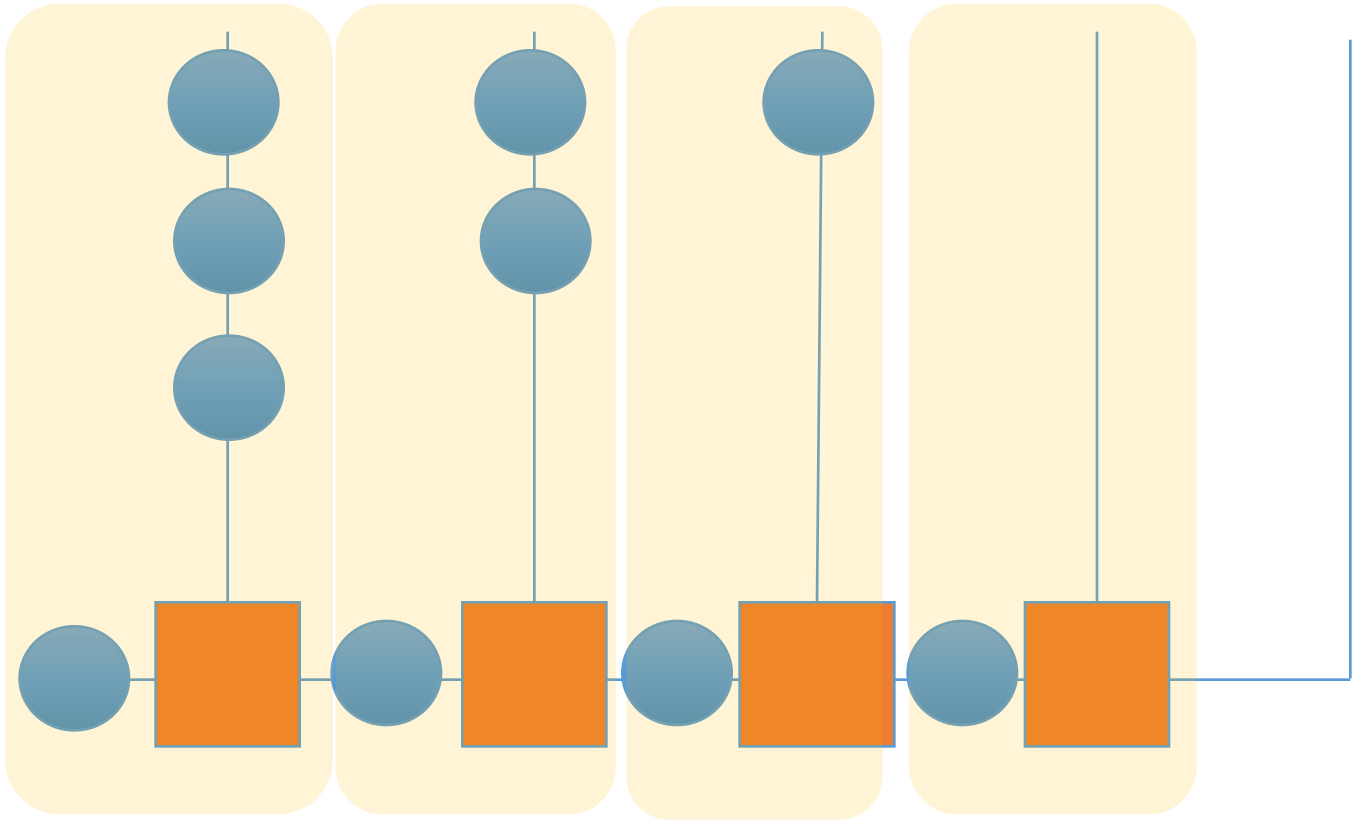












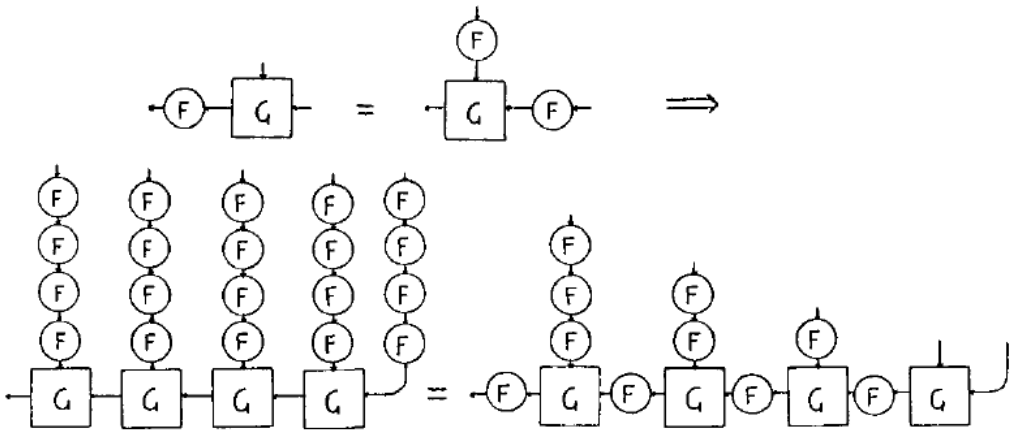


FIG 8. F . G = G . @F => /G . <>F = /(F.G) . mL>F (/>)

Bird Meertens formalism

Squiggol

J. Functional Programming **9** (4): 355–372, July 1999. Printed in the United Kingdom

A tutorial on the universality and expressiveness of fold

GRAHAM HUTTON

University of Nottingham, Nottingham, UK
http://www.cs.nott.ac.uk/~gmh

Universal property of fold

```
g [ ]       =  v
g (x : xs)  =  f x (g xs)
```

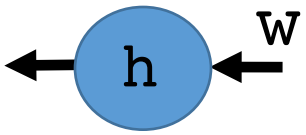
⇔

```
g = fold  f v
```

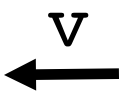
fusion property of fold

```
h w = v
h (g x y) = f x (h y)

=>

h . foldr g w = foldr f v
```

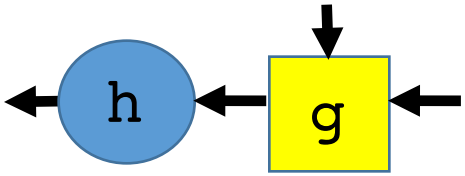
```
h w = v
```



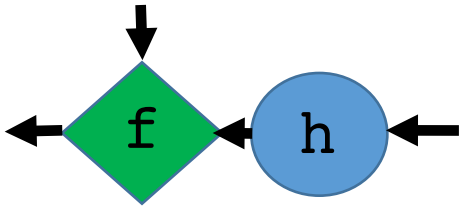
=



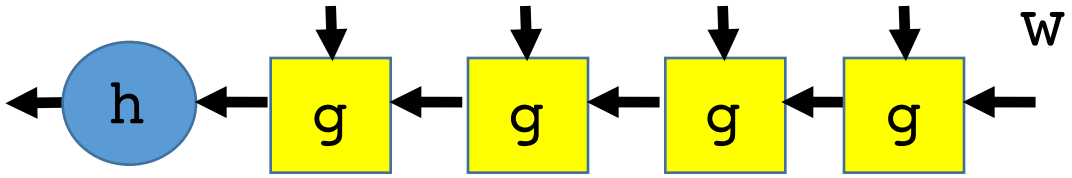

```
h (g x y) = f x (h y)
```



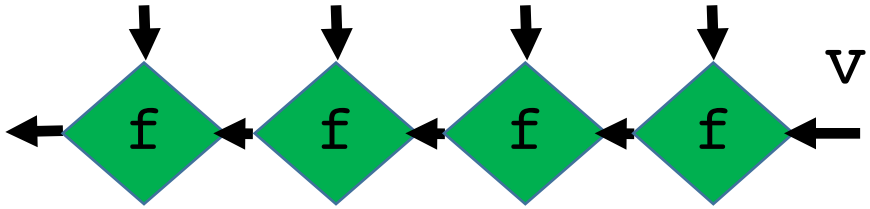
=



```
h . foldr g w = foldr f v
```



=



QuickSpec! Max Algehed Room 3 15.55 today!

```
== Signature ==
 map :: (a -> b) -> [a] -> [b]
fold :: (a -> b -> b) -> b -> [a] -> b
 (.) :: (a -> b) -> (c -> a) -> c -> b
  [] :: [a]
 (:) :: a -> [a] -> [a]
```

QuickSpec! Max Algehed Room 3 15.55 today!

```
== Laws ==
...
  4. (f . g) . h = f . (g . h)
  5. map (f . g) xs = map f (map g xs)
...
  9. fold (f . g) x xs = fold f x (map g xs)
```

Combining forms

capture patterns

support “whole value” programming

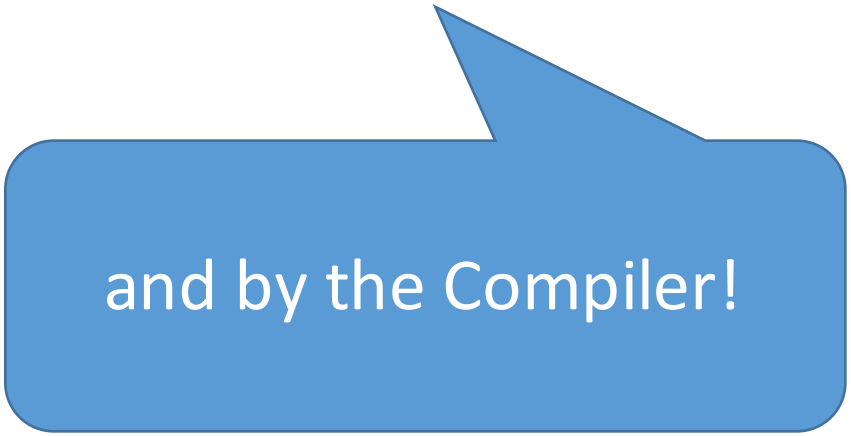
enable reasoning by the programmer

Combining forms

capture patterns

support “whole value” programming

enable reasoning by the programmer

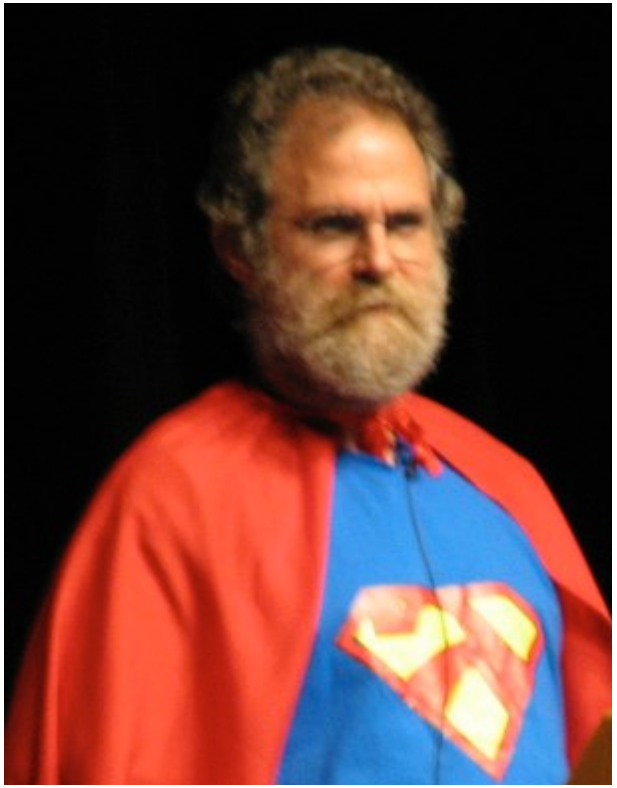


DEFORESTATION: TRANSFORMING PROGRAMS TO ELIMINATE TREES *

Philip WADLER

Department of Computer Science, University of Glasgow, Glasgow G12 8QQ, UK

Abstract. An algorithm that transforms programs to eliminate intermediate trees is presented. The algorithm applies to any term containing only functions with definitions in a given syntactic form, and is suitable for incorporation in an optimizing compiler.

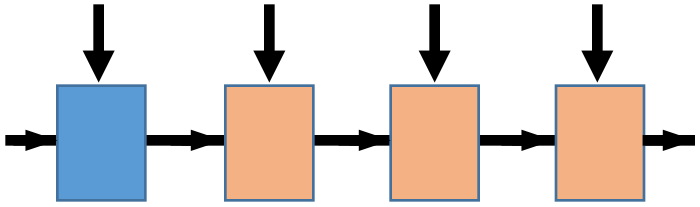


Theoretical Computer Science 73 (1990)

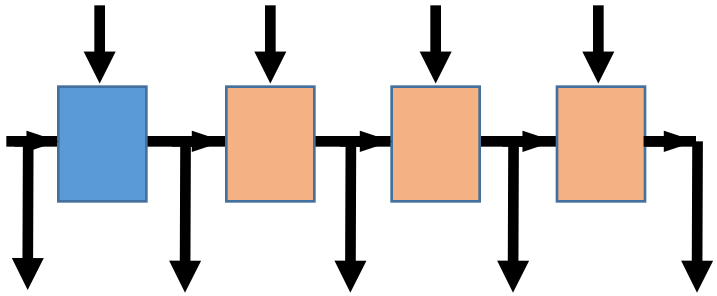
```
build :: (forall  b. (a->b—>b) -> b -> b) -> [a]
build g = g (:) []

{-# RULES
“foldr/build”
   forall k z (g::forall bb. (a->b->b)->b->b) .
   foldr k z (build g) = g k z
#-}
```

Playing by the Rules: Rewriting as a practical optimisation technique in GHC. Peyton Jones, Tolmach, Hoare. Haskell Workshop 2001

foldl



foldl to scanl

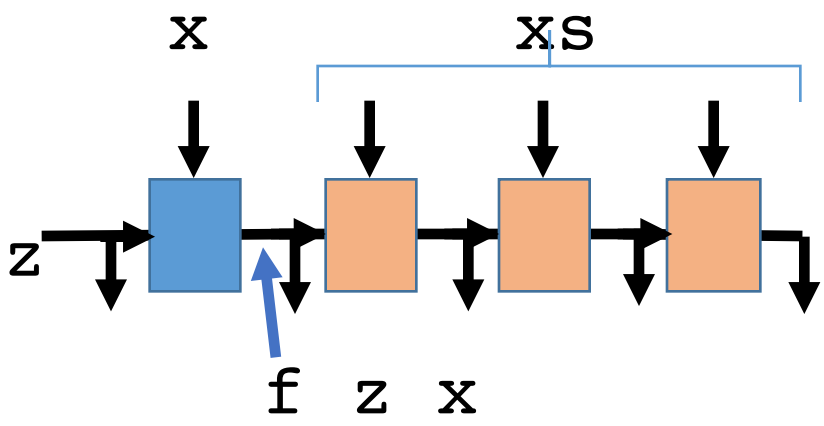


```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f z []          = [z]
scanl f z (x:xs)      =  z : scanl f (f z x) xs)
```

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f z []          = [z]
scanl f z (x:xs)      =  z : scanl f (f z x) xs)
```

```
*Main> scanl (+) 0 [2,4,6,8,10,12]
[0,2,6,12,20,30,42]
```

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f z []          = [z]
scanl f z (x:xs)      =  z : scanl f (f z x) xs)
```



Algebraic Identities for Program Calculation

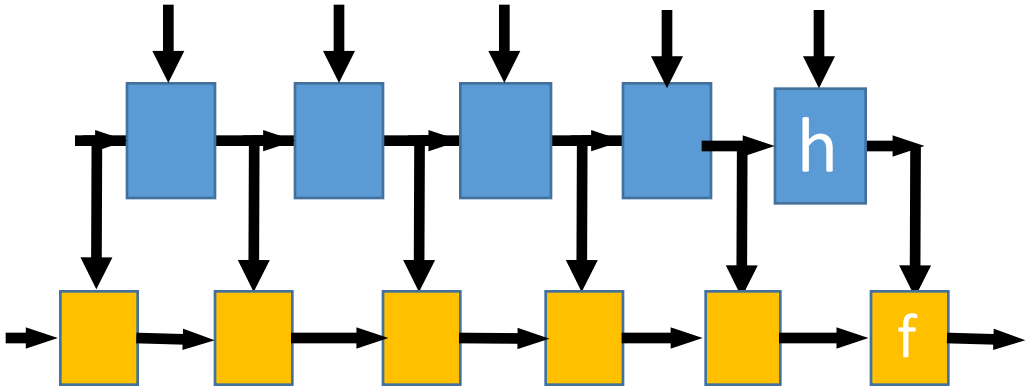
R. S. BIRD

Programming Research Group, University of Oxford, 11 Keble Road, Oxford OX1 3QD

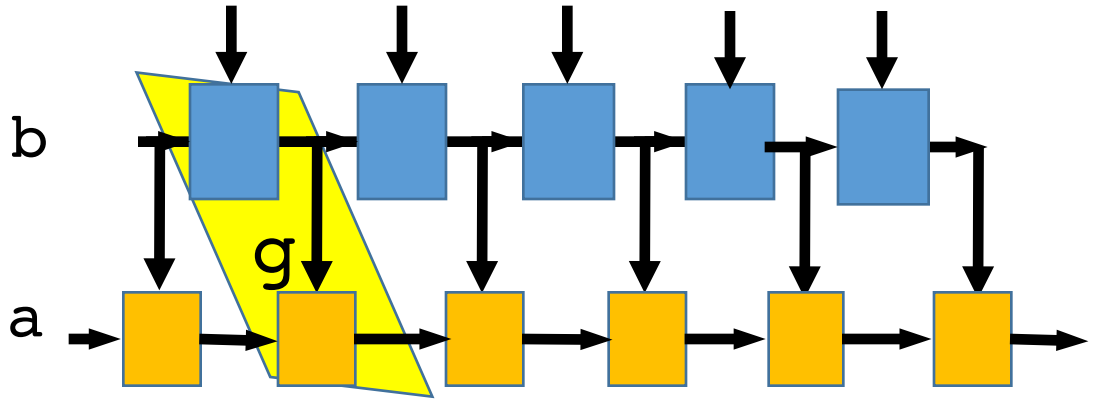
The Computer Journal 32(2) 1989

fold scan fusion

```
foldl f a . scanl h b                    =

fst . foldl g (f a b, b)

g (u,v) x = (f u w, w)
  where  w = h v x
```




```
foldl f a . scanl h b
```




```
fst . foldl g (f a b, b)
```

```
g (u,v) x = (f u w, w) where  w = h v x
```

$$
\begin{aligned}
\mathsf{mss} &= \text{definition} \\
&\quad \mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{segs} \\
&= \text{definition of segs} \\
&\quad \mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{concat} \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
&= \text{map promotion (7)} \\
&\quad \mathsf{max} \cdot \mathsf{concat} \cdot \mathsf{map}(\mathsf{map\,sum}) \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
&= \text{definition of max and fold promotion (8)} \\
&\quad \mathsf{max} \cdot \mathsf{map\,max} \cdot \mathsf{map}\ (\mathsf{map\,sum}) \cdot \mathsf{map\,tails} \cdot \mathsf{inits} \\
&= \text{map distributivity (3)} \\
&\quad \mathsf{max} \cdot \mathsf{map}\ (\mathsf{max} \cdot \mathsf{map\,sum} \cdot \mathsf{tails}) \cdot \mathsf{inits} \\
&= \text{Horner's rule, with } x \otimes y = (x+y)\uparrow 0 \\
&\quad \mathsf{max} \cdot \mathsf{map}\ (\mathsf{foldl}\ (\otimes)\ 0) \cdot \mathsf{inits} \\
&= \text{scan lemma (12)} \\
&\quad \mathsf{max} \cdot \mathsf{scanl}\ (\otimes)\ 0 \\
&= \text{fold-scan fusion (13)} \\
&\quad \mathsf{fst} \cdot \mathsf{foldl}\ (\odot)\ (0,0)
\end{aligned}
$$

where $\odot$ is defined by

$$(u, v) \odot x = (u\uparrow w, w) \quad \text{where} \quad w = (v+x)\uparrow 0$$

The Computer Journal 32(2) 1989

$$
\begin{aligned}
\mathrm{mss} &= \text{definition} \\
&\quad \max \cdot \mathrm{map}\,\mathrm{sum} \cdot \mathrm{segs} \\
&= \text{definition of segs} \\
&\quad \max \cdot \mathrm{map}\,\mathrm{sum} \cdot \mathrm{concat} \cdot \mathrm{map}\,\mathrm{tails} \cdot \mathrm{inits} \\
&= \text{map promotion (7)} \\
&\quad \max \cdot \mathrm{concat} \cdot \mathrm{map}(\mathrm{map}\,\mathrm{sum}) \cdot \mathrm{map}\,\mathrm{tails} \cdot \mathrm{inits}
\end{aligned}
$$

maximum segment sum

the fibonacci of program calculation

$$
\begin{aligned}
&\quad \text{scan fusion (13)} \\
&\quad \mathrm{fst} \cdot \mathrm{foldl}\,(\odot)\,(0,0)
\end{aligned}
$$

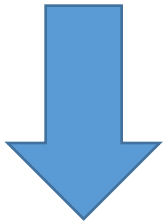
where $\odot$ is defined by

$$(u, v) \odot x = (u \uparrow w, w) \quad \text{where} \quad w = (v + x) \uparrow 0$$

The Computer Journal 32(2) 1989

Maximum segment sum

```
maximum . map sum . segs
```



```
maximum . scanl f 0
```

```
f x y = max (x+y) 0
```

```
fst . foldl g (0,0)
```

```
g (u,v) x = (max u w,w)
w = max (v+x) 0
```

Maximum segment sum

```
maximum . map sum . segs
```

$O(n^3)$

```
maximum . scanl f 0
```

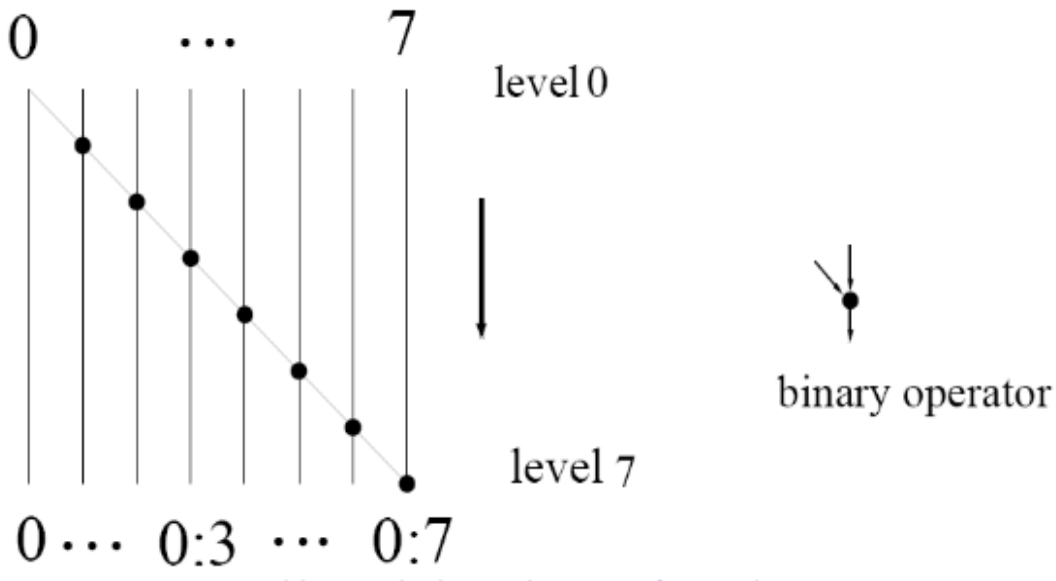
```
f x y = max (x+y) 0
```

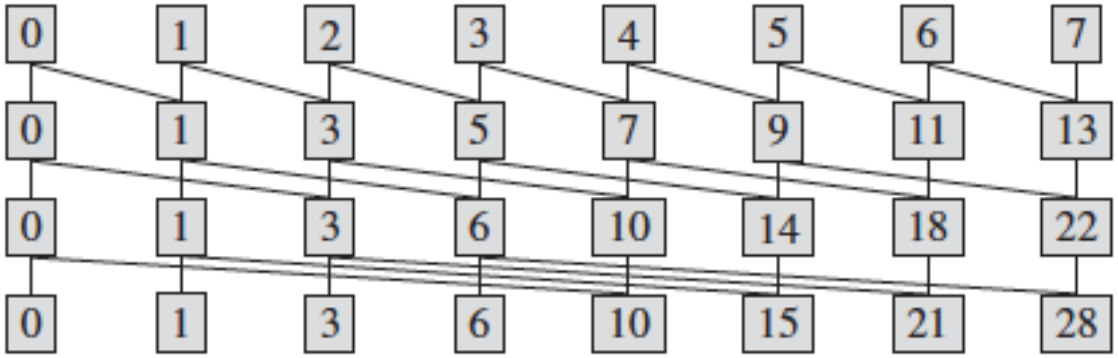
$O(n)$

```
fst . foldl g (0,0)
```

```
g (u,v) x = (max u w,w)
w = max (v+x) 0
```

scan diagram





Kogge & Stone IEEE Transactions on Computers, 1973, *C-22*

Bitwise Operations

C. STRACHEY, *London, England*

Recent contributions on the subject of counting the ones or reversing the digits of a binary word have prompted some rather more general observations on bitwise operations which treat each binary digit in a computer word as a separate entity.

If speed is the main consideration and space is no object, it seems clear that the best way of performing these operations is by some form of table look-up. The word is usually divided into sections of a convenient size—e.g., six bits—and the operation is then reduced effectively to a simple form of digit by a digit process working in the scale of 64. This effective though rather uninteresting process is equally applicable to any bitwise operation.

Programmers who are used to working with machines with a very small store, however, are reluctant to use a method which seems so crude and which involves using so large a table. It seems therefore of some interest to consider whether it is possible to devise an "efficient" process for bitwise operations. In this context "efficient" is taken to mean a process in which both the space occupied and the time taken are functions which increase only approximately logarithmically with the number of bits in the word. One such process, for counting ones, has been known to exist for some time and is published in the second edition of Wilkes, Wheeler and Gill. The purpose of this note is to indicate how similarly "efficient" processes may be designed for other bitwise operations. As an example let us consider the problem of reversing the bits in a word.

The requirement of efficiency means that if we double the word length we are only allowed one more stage of the operation. As this single stage of operation has to double the amount of information in the answer, it clearly cannot invoke any form of table look-up or simple transformation of the bits of the word. It follows therefore that the final answer must be formed from the bits of the original word by means of some form of shunting or shuffling. Figure 1 shows how this can be done using an 11-bit word.

The word to be shifted ($a, b, c, \cdots, k$) is written in the least significant half of a double length register, and the reversed word which it is desired to obtain ($k'j'i'\cdots a'$) is then derived in the most significant half of the same register. Underneath each digit of the original word is written the number of left shifts which this digit requires to reach its final position. The rows below the shift numbers represent the binary decomposition of these numbers; the reversal can then be carried out by successive shifts of the numbers indicated by the 16, 8, 4, 2 and 1 places. The remaining lines show the various steps of the process.

This operation requires the digits of a double length register to be extracted and shifted. The same effect can be obtained easily in machines with only a single length accumulator by treating separately the two halves of the word to be reversed. Not all rearrangements of the digits of a word are possible by this means because of the possibility of overlapping at some of the intermediate stages. Reversal, however, presents no difficulties. The actual details of the program for any particular computer depend critically on the exact form of the logical operations available. As most computers are designed to perform arithmetic and *not* bitwise operations, these instructions are frequently few and inconvenient with the result that the "efficient" method sometimes turns out to be both more cumbersome and slower than cruder methods.

k'	j'	i'	h'	g'	f'	e'	d'	c'	b'	a'	a	b	c	d	e	f	g	h	i	j	k	Initial Position
											1	3	5	7	9	11	13	15	17	19	21	Shift Numbers
											0	0	0	0	0	0	0	0	1	1	1	Shift 16
											0	0	0	0	1	1	1	1	0	0	0	Shift 8
											0	0	1	1	0	0	1	1	0	0	1	Shift 4
											0	1	0	1	0	1	0	1	0	1	0	Shift 2
											1	1	1	1	1	1	1	1	1	1	1	Shift 1
.	.	.	.	.	.	.	.	.	.	.	a	b	c	d	e	f	g	h	i	j	k	Initial Position
.	.	.	i	j	k	.	.	.	.	.	a	b	c	d	e	f	g	h	.	.	.	(Shift 16)
.	.	.	i	j	k	.	e	f	g	h	a	b	c	d	.	.	.	.	.	.	.	(Shift 8)
.	k	.	i	j	g	h	e	f	c	d	a	b	.	.	.	.	.	.	.	.	.	(Shift 4)
.	k	j	i	h	g	f	e	d	c	b	a	.	.	.	.	.	.	.	.	.	.	(Shift 2)
k	j	i	h	g	f	e	d	c	b	a	.	.	.	.	.	.	.	.	.	.	.	Final Position

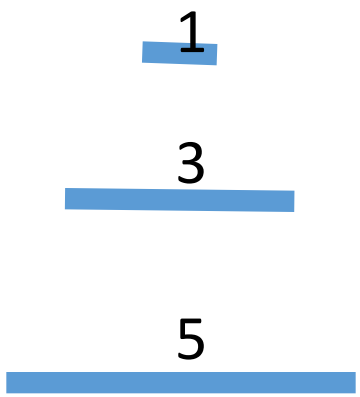
FIG. 1

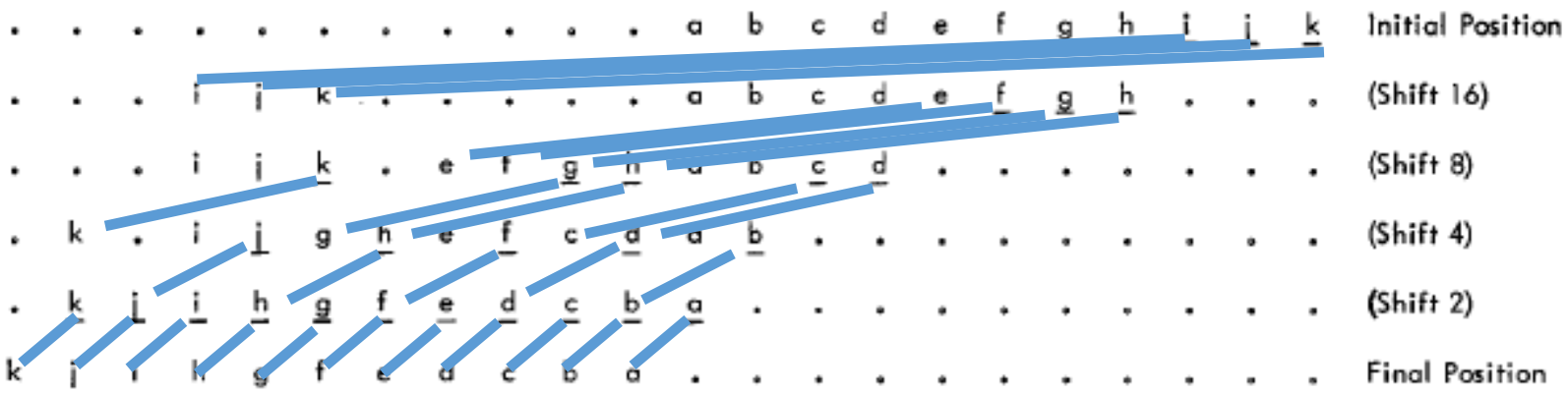
CACM 4(3) March 1961

a b c d e f g h i j k

k j l h g f e d c b a a b c d e f g h i j k

k j I h g f e d c b a a b c d e f g h i j k




. . . . . a b c d e f g h i j k Initial Position
. . . i j k . . . . . a b c d e f g h . . . (Shift 16)
. . . i j k . e f g h a b c d . . . . . (Shift 8)
. k . i j g h e f c d a b . . . . . (Shift 4)
. k j i h g f e d c b a . . . . . (Shift 2)
k j i h g f e d c b a . . . . . Final Position

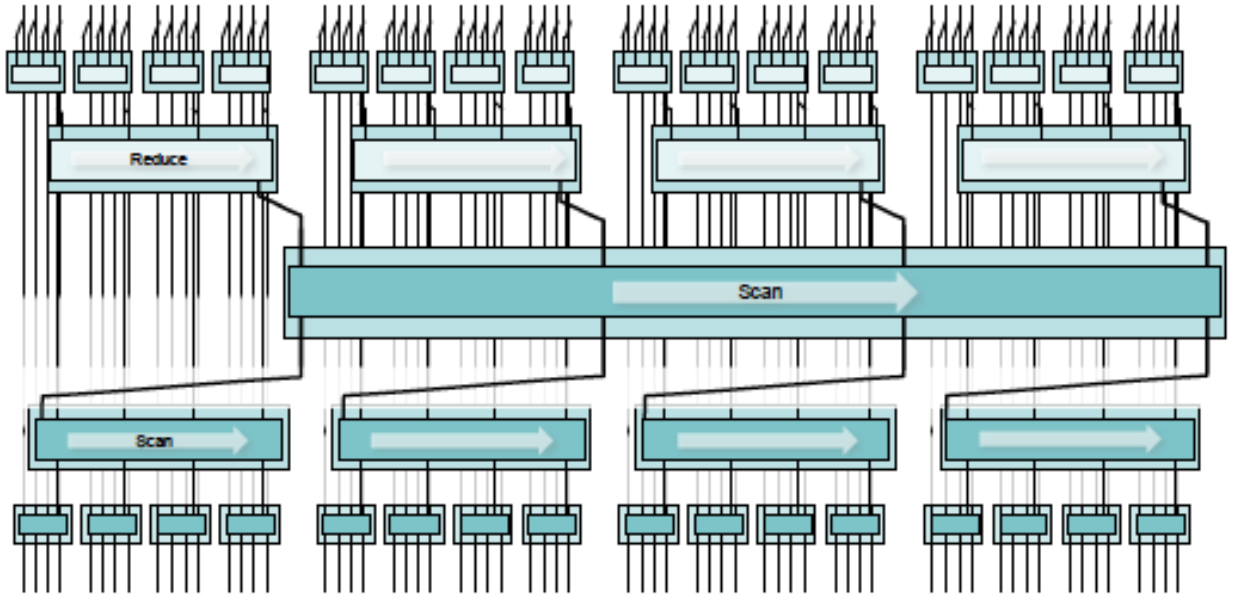
the power of scan Blelloch

With scan you can do nearly anything!

radix sort, evaluate polynomials, solve recurrences, dynamically allocate processors…

Prefix sums and their applications.
CMU-CS-90-190 tech report

GPU



Parallel scan for stream architectures.
Merrill & Grimshaw. U. of Virginia 2009

skeletons Murray Cole

capture generic patterns of parallelism

Algorithmic Skeletons: Structured Management of Parallel Computation
Murray I. Cole MIT Press 1989 (based on PhD thesis)

skeletons

capture generic patterns of parallelism

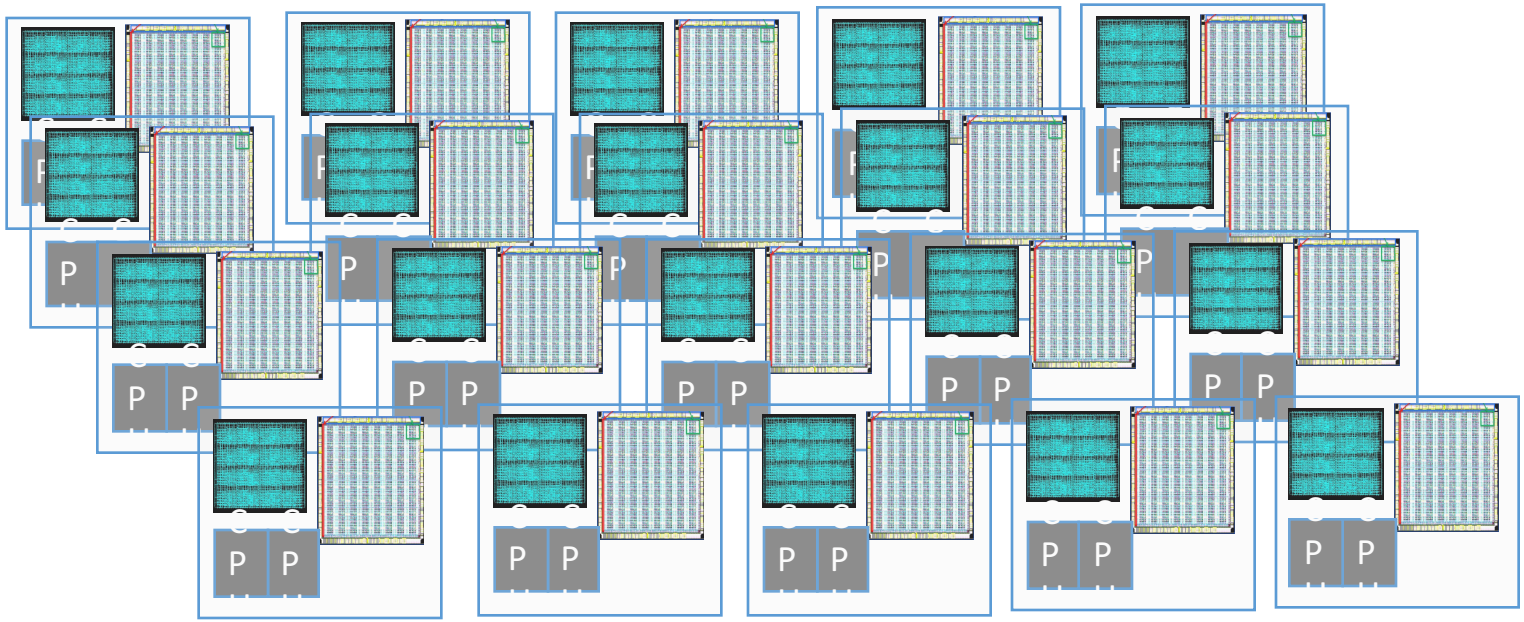
Stencils Wavefronts Divide & Conquer

Task Farm Pipeline …

González-Vélez and Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers , Software: Practice and Experience 40(12) 2010.

skeletons

closely related to higher order functions


P
P P

Backus

the question of it still seems that programming is a pretty low-level enterprise, and that somebody ought to be thinking about how to make it higher; really higher level than it is

Help!

Expressing and controlling locality of data

Ensuring security

Ensuring correctness

Controlling power consumption

Help!

Expressing and controlling locality of data

Ensuring securit

Octopi Safe progra

Chalmers

Information Security (

Functional Programmi n)

31 MSEK (SSF)
hiring doctoral students!

Strachey:
It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing