Why Functional Programming Matters

John Hughes
Mary Sheeran

Functional Programming à la 1940s

- Minimalist: who needs booleans?
- A boolean just *makes a choice!*

```
true  x y = x
false x y = y
```

- We can *define* if-then-else!

```
ifte bool t e =
  bool t e
```

Who needs integers?

- A (positive) integer just *counts loop iterations!*

```
two  f x = f (f x)
one  f x = f x
zero f x = x
```

- To recover a "normal" integer…

```
*Church> two (+1) 0
2
```

Look, Ma, we can add!

- Addition by *sequencing* loops

```
add m n f x = m f (n f x)
```

m

n

- Multiplication by *nesting* loops!

```
mul m n f x = m (n f) x
```

nm

```
*Church> add one (mul two two) (+1) 0
5
```

Factorial à la 1940

```
fact n =
  ifte (iszero n)
       one
       (mul n (fact (decr n)))
```

```
*Church> fact (add one (mul two two)) (+1) 0
120
```

A couple more auxiliaries

- Testing for zero

```
iszero n =
  n (\_ -> false) true
```

- Decrementing…

```
decr n =
  n (\m f x-> f (m incr zero))
    zero
    (\x->x)
    zero
```

Booleans, integers, (and other data structures) *can be entirely replaced by functions!*

"Church encodings"

Early versions of the Glasgow Haskell compiler actually implemented data-structures this way!

Alonzo Church

Before you try this at home…

```
Church.hs:27:35:
    Occurs check: cannot construct the infinite type:
        t ~ t -> t -> t
```

Actual type:

```
(((((t -> t -> t) -> t -> t)
  -> (t -> t -> t) -> t -> t -> t)
 -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
 -> ((t -> t -> t) -> t -> t)
 -> (t -> t -> t)
 -> t
 -> t
 -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> ((t -> t -> t) -> t -> t)
-> (t -> t -> t)
-> t
-> t
-> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t)
    -> (t -> t -> t)
    -> t
    -> t
    -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> ((((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
      -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
      -> ((t -> t -> t) -> t -> t)
      -> (t -> t -> t)
      -> t
      -> t
      -> t)
     -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> ((t -> t -> t) -> t -> t)
     -> (t -> t -> t)
     -> t
     -> t
     -> t)
    -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
        -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
        -> ((t -> t -> t) -> t -> t)
        -> (t -> t -> t)
        -> t
        -> t
        -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t)
    -> (t -> t -> t)
    -> t
    -> t
    -> t)
-> (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> ((t -> t -> t) -> t -> t)
     -> (t -> t -> t)
     -> t
     -> t
     -> t)
    -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
        -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t)
    -> (t -> t -> t)
    -> t
    -> t
    -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t)
    -> (t -> t -> t)
    -> t
    -> t
    -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> ((t -> t -> t) -> t -> t)
-> (t -> t -> t)
-> t
-> t
-> t
```

Expected type:

```
(((((t -> t -> t) -> t -> t)
   -> (t -> t -> t) -> t -> t -> t)
  -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
  -> ((t -> t -> t) -> t -> t)
  -> (t -> t -> t)
  -> t
  -> t
  -> t)
 -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
 -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
 -> ((t -> t -> t) -> t -> t)
 -> (t -> t -> t)
 -> t
 -> t
 -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
    -> ((t -> t -> t) -> t -> t)
    -> (t -> t -> t)
    -> t
    -> t
    -> t)
-> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
-> ((t -> t -> t) -> t -> t)
-> (t -> t -> t)
-> t
-> t
-> t
```

But wait, there's more…

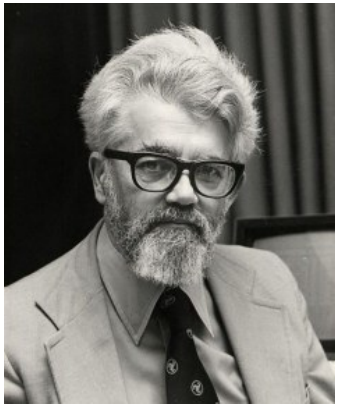
Relevant bindings include

```
n :: (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
       -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
       -> ((t -> t -> t) -> t -> t)
       -> (t -> t -> t)
       -> t
       -> t
       -> t)
      -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
      -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
      -> ((t -> t -> t) -> t -> t)
      -> (t -> t -> t)
      -> t
      -> t
      -> t)
     -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)
         -> t
         -> t
         -> t)
     -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)

         -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
     -> ((((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
           -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
           -> ((t -> t -> t) -> t -> t)
           -> (t -> t -> t)
           -> t
           -> t
           -> t)
          -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> ((t -> t -> t) -> t -> t)
          -> (t -> t -> t)
          -> t
          -> t
          -> t)
         -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t)
             -> (t -> t -> t)
             -> t
             -> t
             -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)
         -> t
         -> t
         -> t)
     -> (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> ((t -> t -> t) -> t -> t)
          -> (t -> t -> t)
          -> t

          -> t
          -> t)
         -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)
         -> t
         -> t
         -> t)
     -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)
         -> t
         -> t
         -> t)
     -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)

         -> t
         -> t
         -> t
    (bound at Church.hs:27:6)
  fact :: ((((((t -> t -> t) -> t -> t)
             -> (t -> t -> t) -> t -> t -> t)
            -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
            -> ((t -> t -> t) -> t -> t)
            -> (t -> t -> t)
            -> t
            -> t
            -> t)
           -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
               -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
           -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
           -> ((t -> t -> t) -> t -> t)
           -> (t -> t -> t)
           -> t
           -> t
           -> t)
          -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t)
              -> (t -> t -> t)
              -> t
              -> t
              -> t)
          -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> ((((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                -> ((t -> t -> t) -> t -> t)
                -> (t -> t -> t)
                -> t
                -> t
                -> t)
               -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                   -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
               -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
               -> ((t -> t -> t) -> t -> t)
               -> (t -> t -> t)
               -> t
               -> t
               -> t)
              -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                  -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                  -> ((t -> t -> t) -> t -> t)
                  -> (t -> t -> t)
                  -> t
                  -> t
                  -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t)
              -> (t -> t -> t)
              -> t
              -> t
              -> t)
          -> (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
               -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
               -> ((t -> t -> t) -> t -> t)
               -> (t -> t -> t)
               -> t
               -> t
               -> t)
              -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                  -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t)
              -> (t -> t -> t)
              -> t
              -> t
              -> t)
-> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t)
              -> (t -> t -> t)
              -> t
              -> t
              -> t)
          -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
          -> ((t -> t -> t) -> t -> t)
          -> (t -> t -> t)
          -> t
          -> t
          -> t)
         -> (((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
              -> ((t -> t -> t) -> t -> t)
              -> (t -> t -> t)
              -> t
              -> t
              -> t)
             -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
                 -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t)
             -> (t -> t -> t)
             -> t
             -> t
             -> t)
         -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t)
             -> (t -> t -> t)
             -> t
             -> t
             -> t)
         -> ((((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
             -> ((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> (((t -> t -> t) -> t -> t) -> (t -> t -> t) -> t -> t -> t)
         -> ((t -> t -> t) -> t -> t)
         -> (t -> t -> t)
         -> t
         -> t
         -> t
      (bound at Church.hs:27:1)
    In the first argument of ‘mul’, namely ‘n’
    In the third argument of ‘ifte’, namely ‘(mul n (fact (decr n)))’
```

The type-checker needs a *little bit* of help

```
fact ::
   (forall a. (a->a)->a->a) ->
   (a->a) -> a -> a
```

Factorial à la 1960



```
(LABEL FACT (LAMBDA (N)
  (COND ((ZEROP N) 1)
        (T (TIMES N (FACT (SUB1 N)))))))
```

Higher-order functions!

```
(MAPLIST FACT (QUOTE (1 2 3 4 5)))

(1 2 6 24 120)
```

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

Factorial in ISWIM

```
fac(5)
  where rec fac(n) =
    (n=1) → 1;
             n*fac(n-1)
```

Laws

`(MAPLIST F (REVERSE L))` ≡ `(REVERSE (MAPLIST F L))`

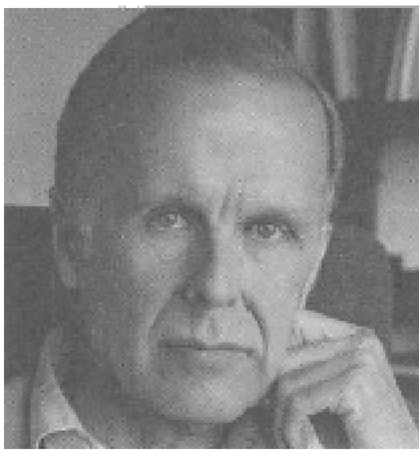
What's the point of two different ways to do the same thing?

Wouldn't *two* facilities be better than one?

Expressive power should be by design, rather than by accident!

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

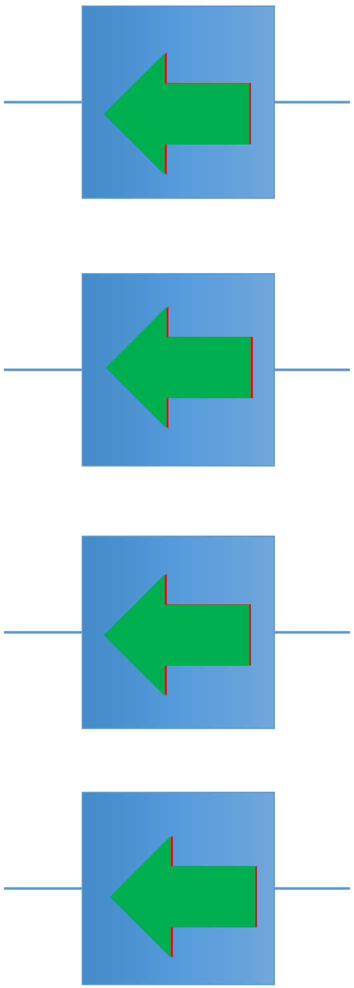
**Conventional programming languages are growing ever more enormous,
but not stronger.**

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

Word-at-a-time

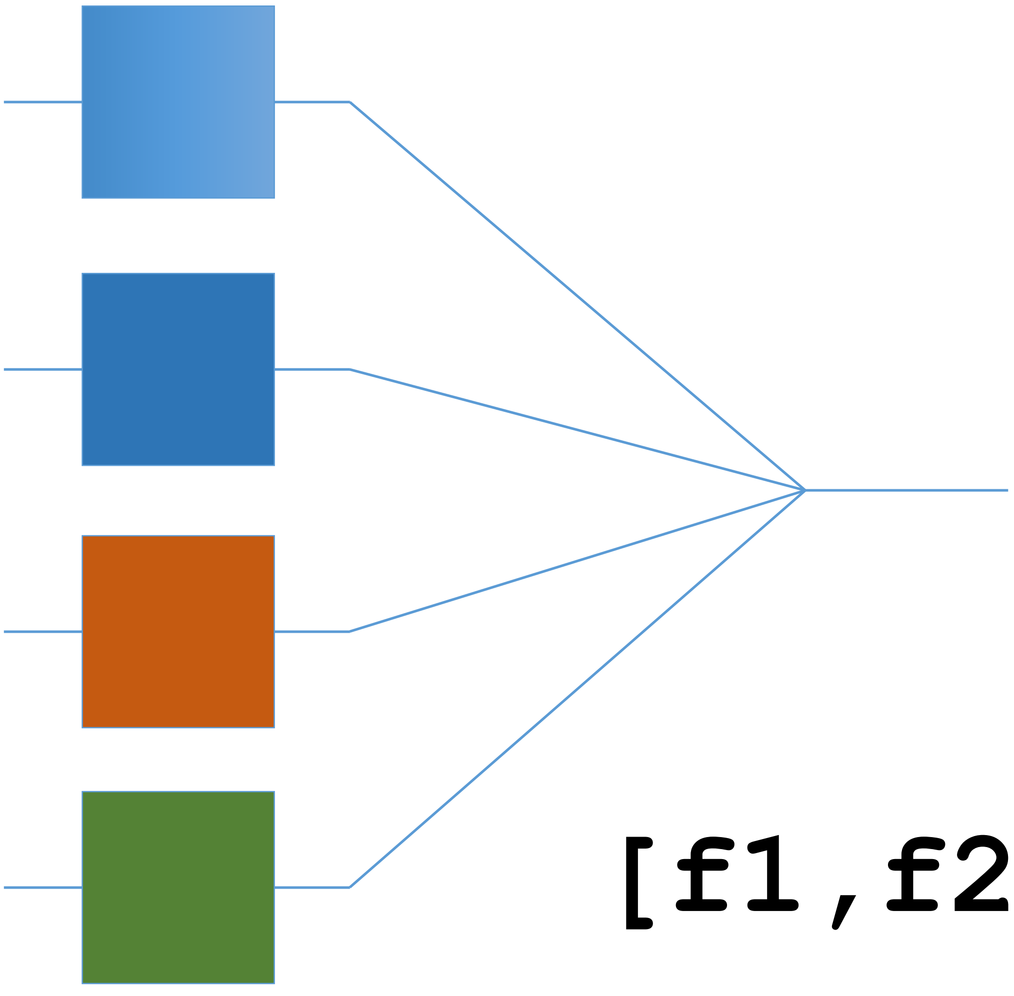
their inability to effectively use
powerful combining forms
for building new programs from
existing ones

apply to all



αf

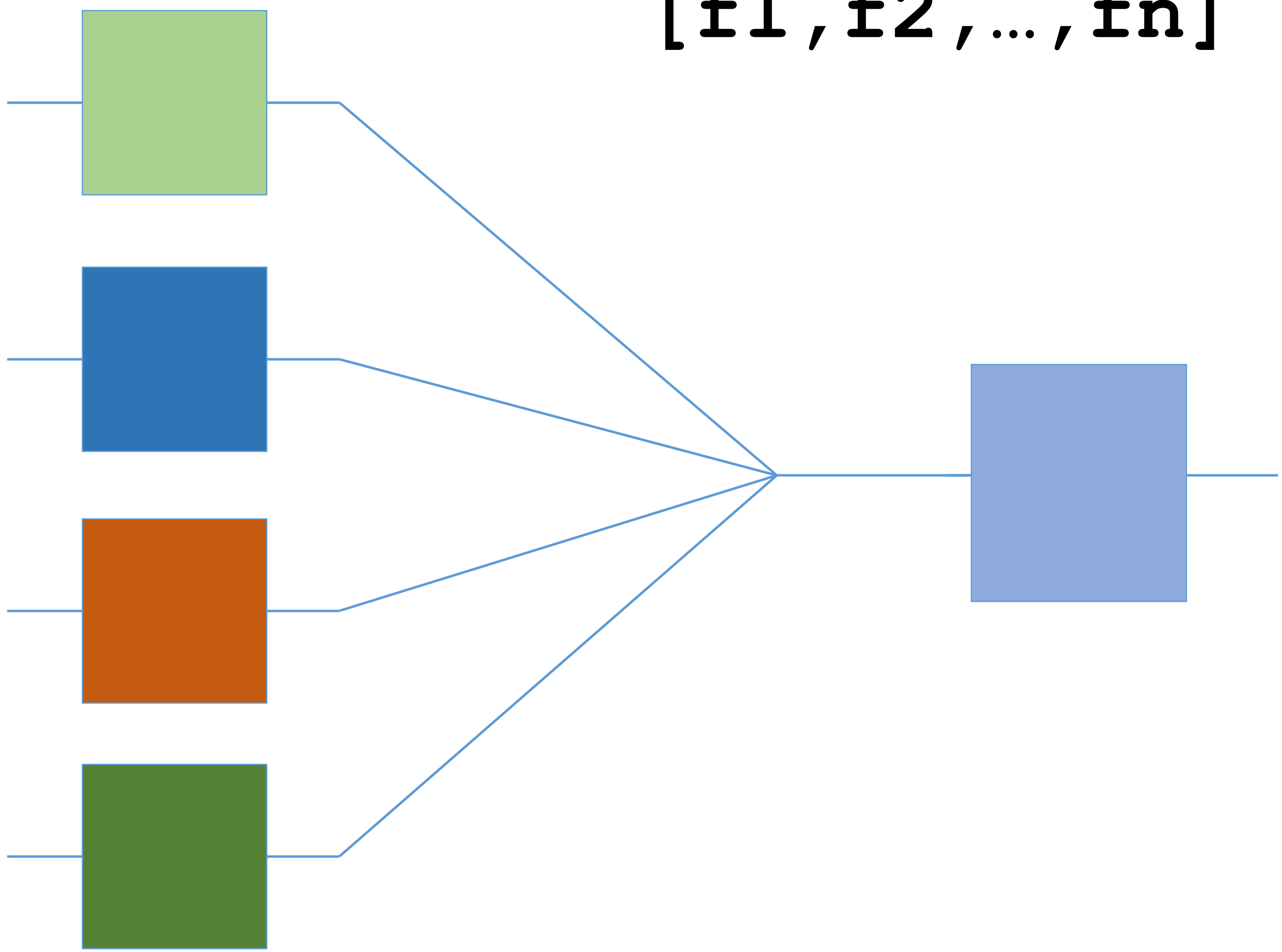
construction



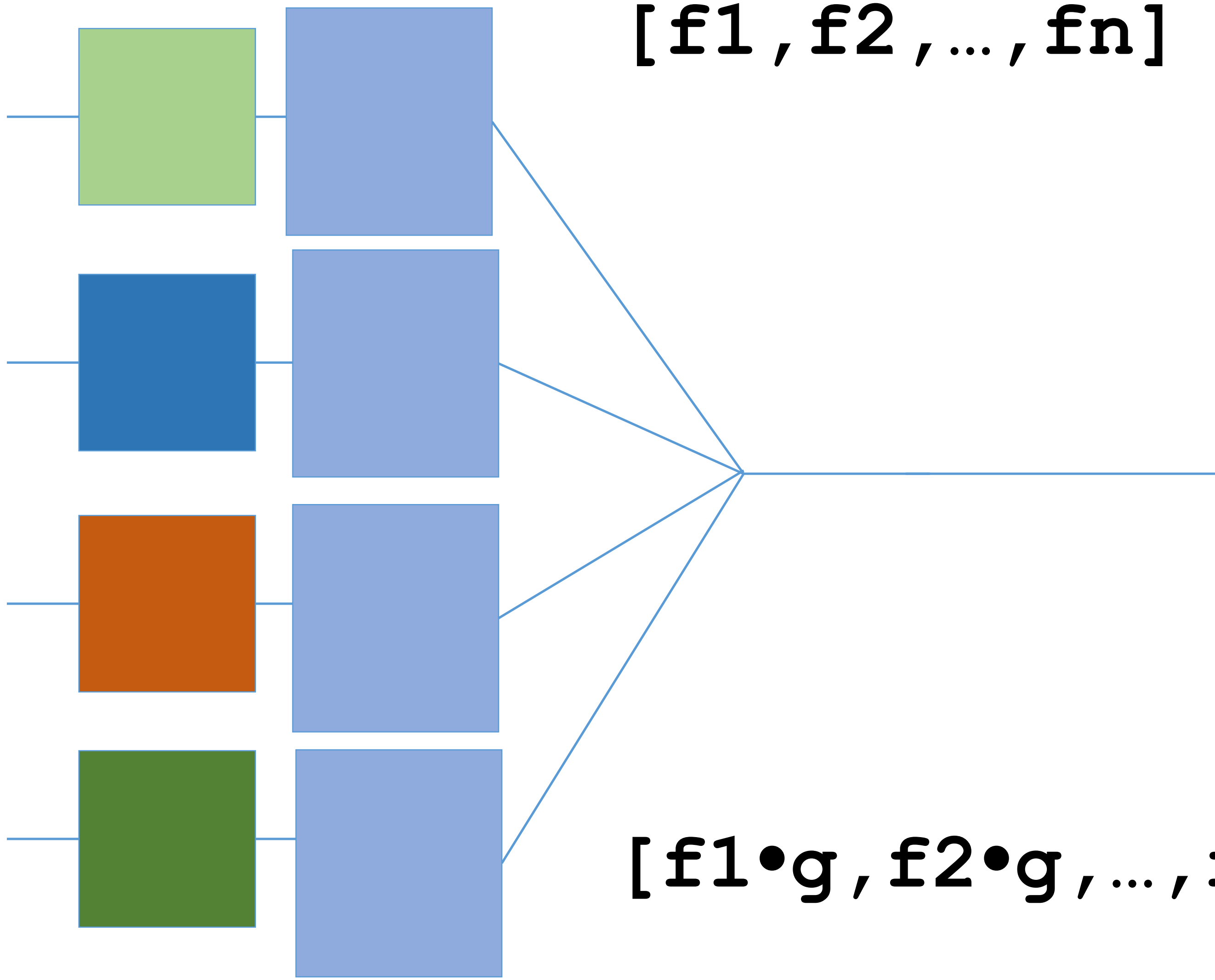
[f1,f2,f3,f4]

their lack of useful mathematical properties for reasoning about programs

[f1,f2,…,fn] • g



[f1,f2,…,fn] • g



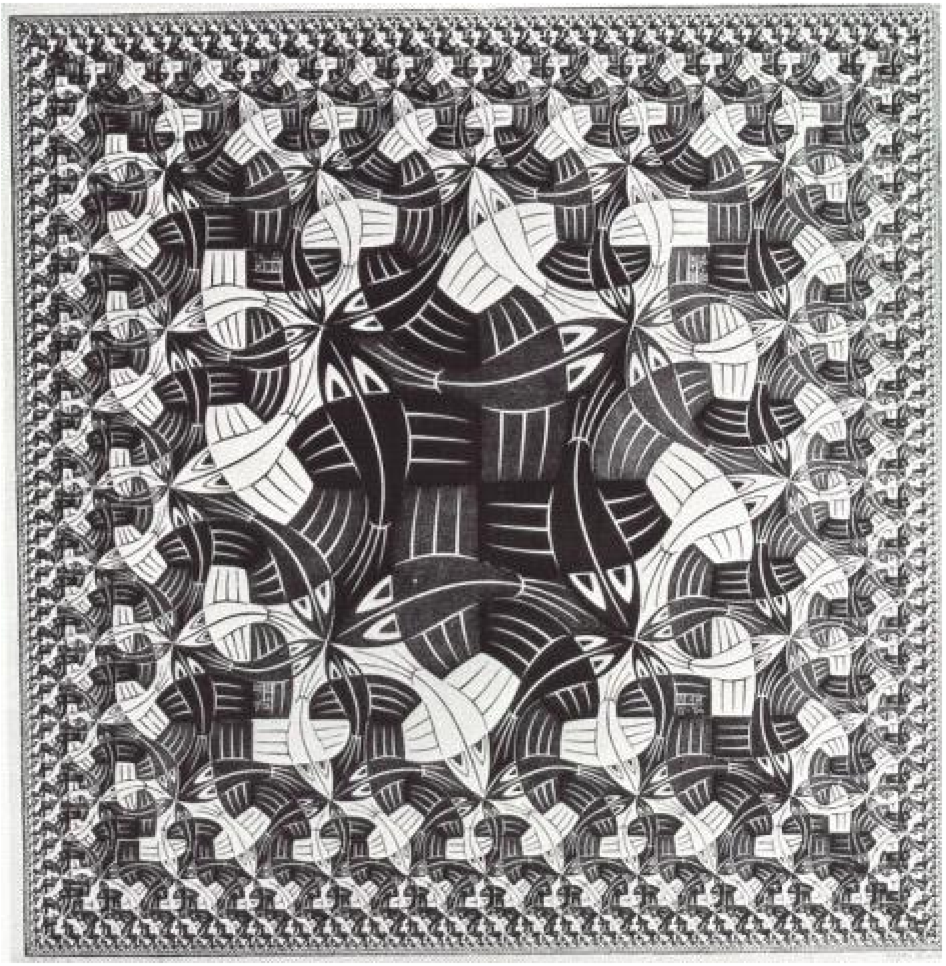
[f1•g,f2•g,…,fn•g]

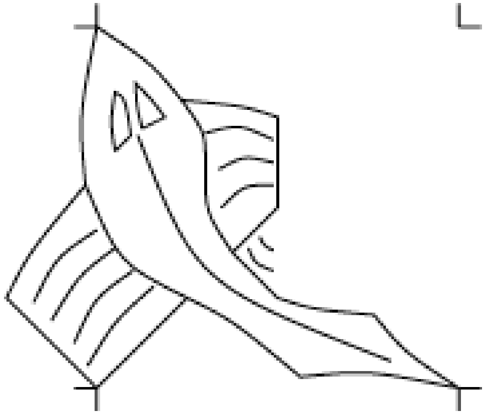
```
c := 0;
for i := 1 step 1 until n do
  c := c + a[i] × b[i]
```

```
Def ScalarProduct  =
   (Insert +) • (ApplyToAll ×) • Transpose
```

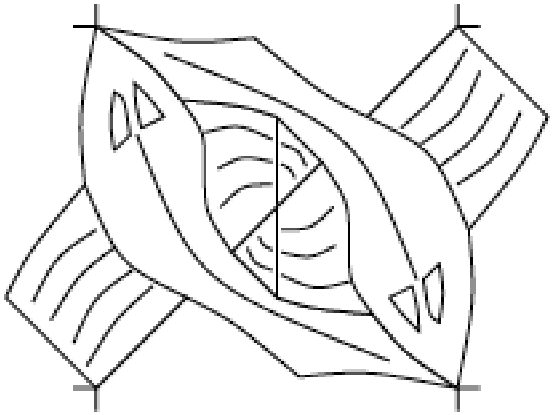
```
Def SP  =  (/ +) • (α ×) • Trans
```

Peter Henderson, Functional Geometry, 1982

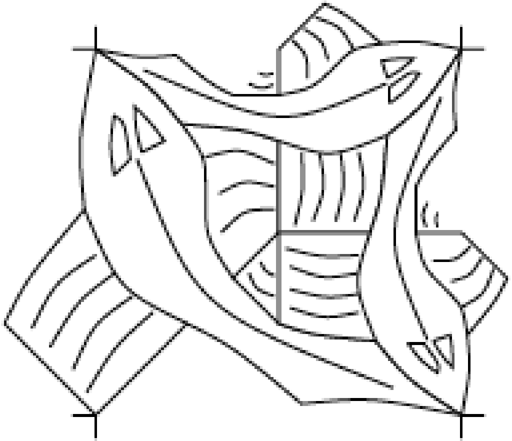




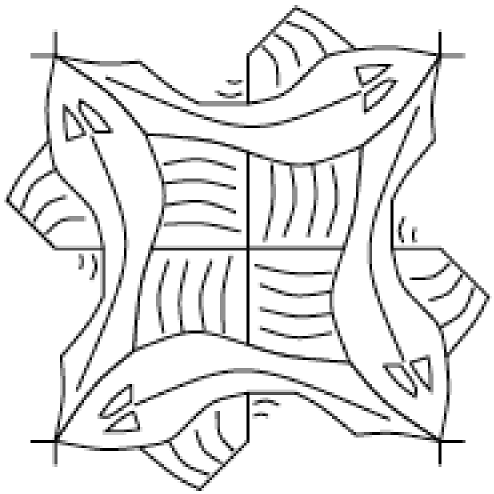
fish



```
over (fish, rot (rot (fish)))
```



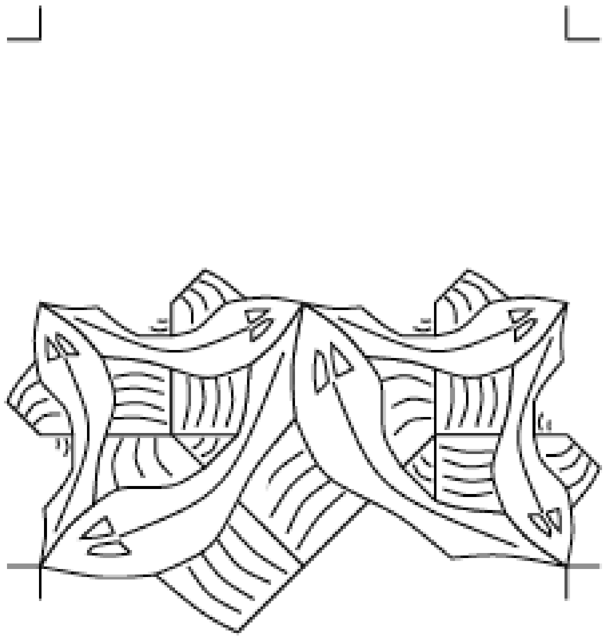
```
t = over (fish, over (fish2, fish3))

fish2 = flip (rot45 fish)
fish3 = rot (rot (rot (fish2)))
```



```
u = over (over (fish2, rot (fish2)),
          over (rot (rot (fish2)),
                rot (rot (rot (fish2)))))
```

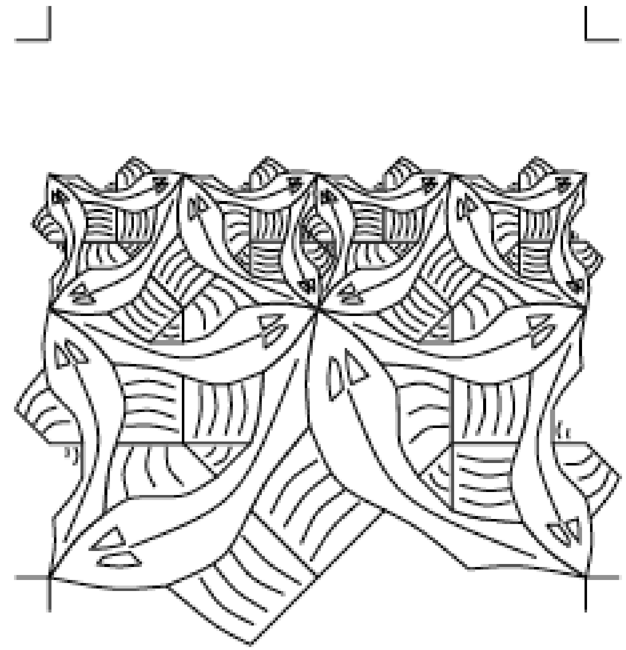
P	Q
R	S

quartet

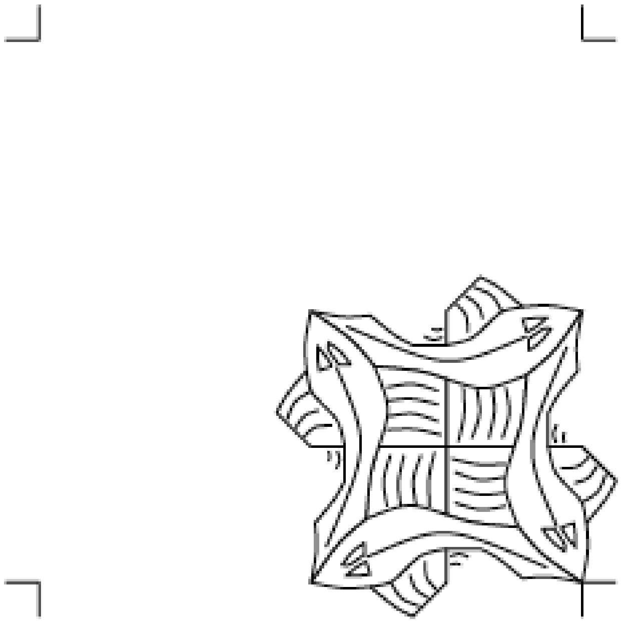


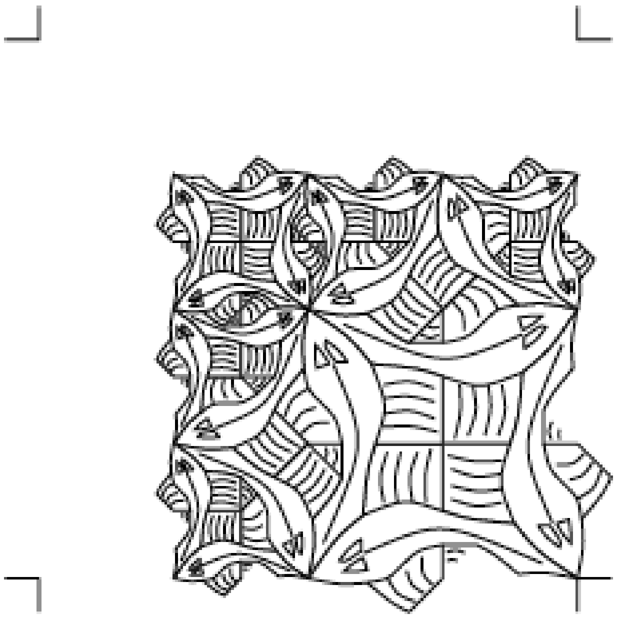
```
quartet(nil,  nil,
        rot(t), t)
```

side1



```
quartet(side1,side1,
        rot(t), t  )
```



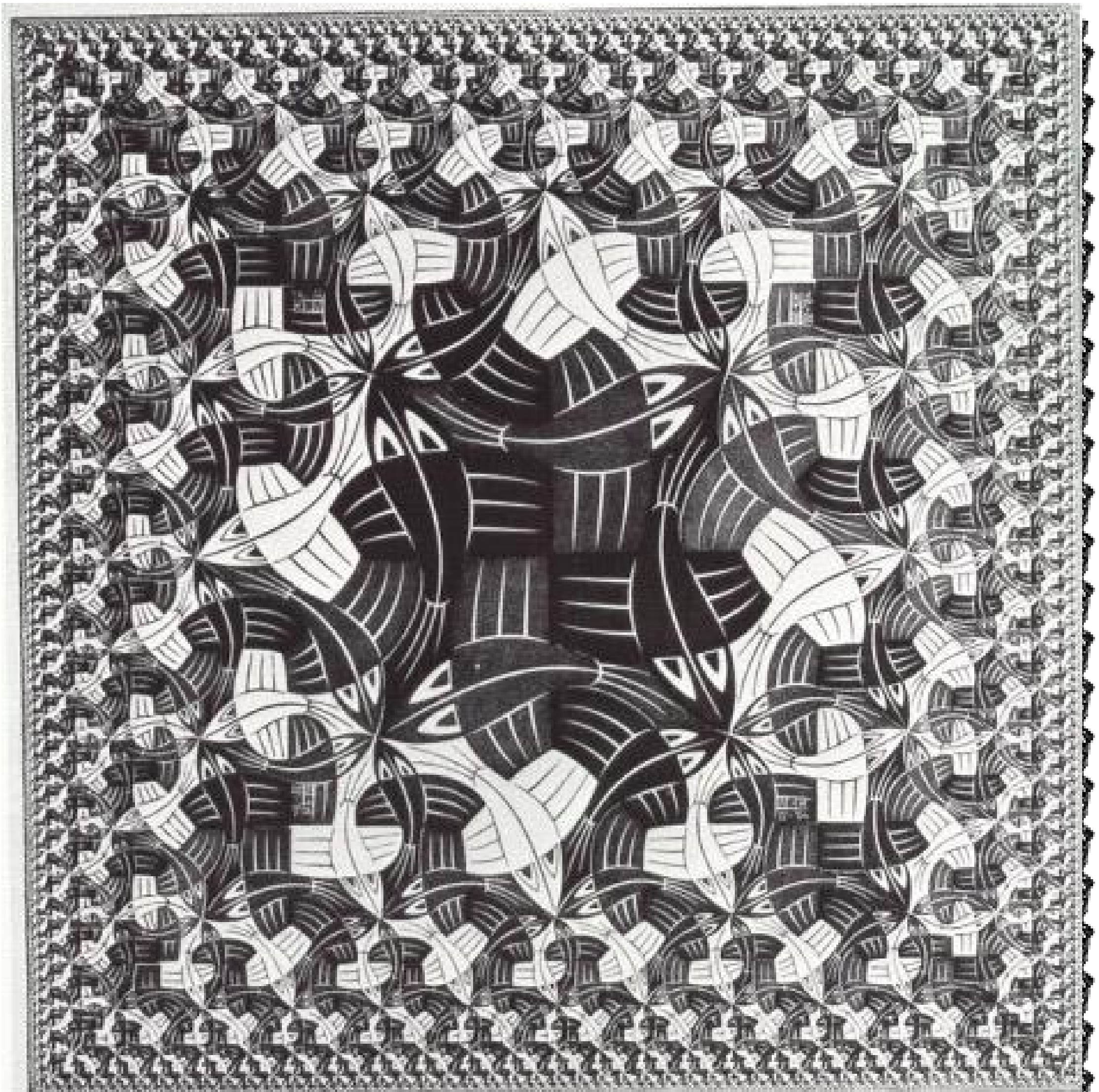


```
quartet (nil,nil,nil,u)
```

corner1

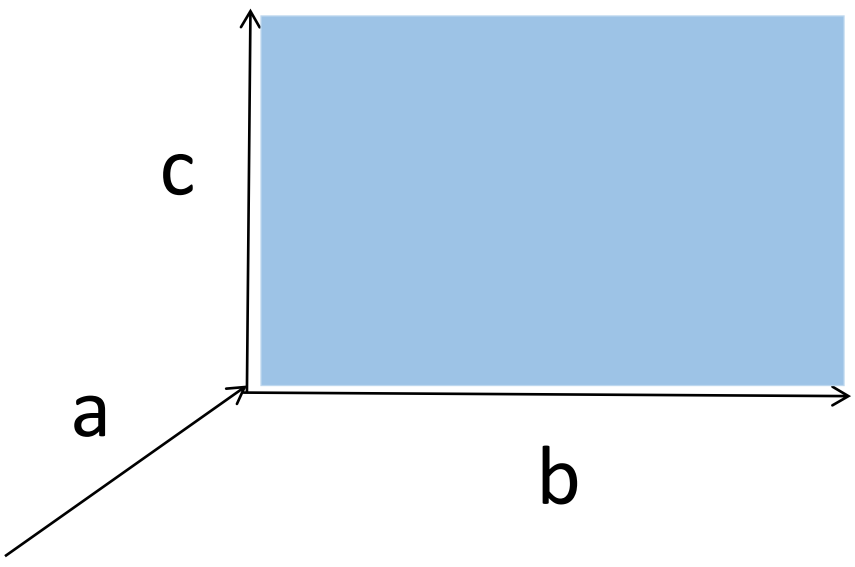
```
quartet(corner1,
        side1,
        rot(side1),
        u)
```

```
squarelimit = nonet(
  corner,        side,                rot(rot(rot(corner))),
  rot(side),     u,                   rot(rot(rot(side))),
  rot(corner), rot(rot(side)), rot(rot(corner)))
```



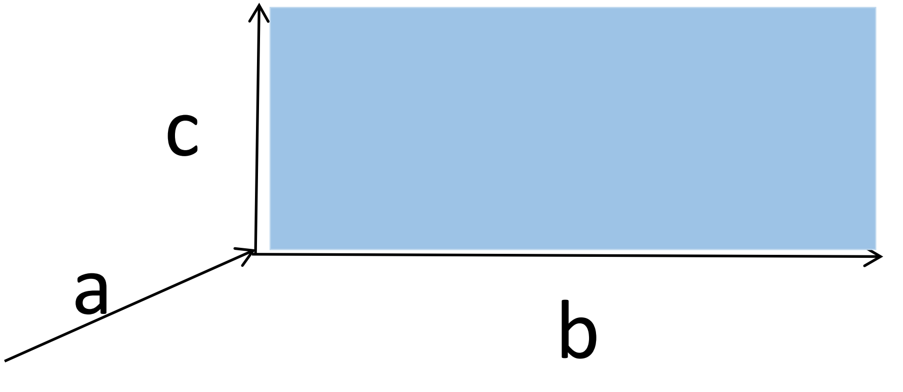
picture = function

picture = function

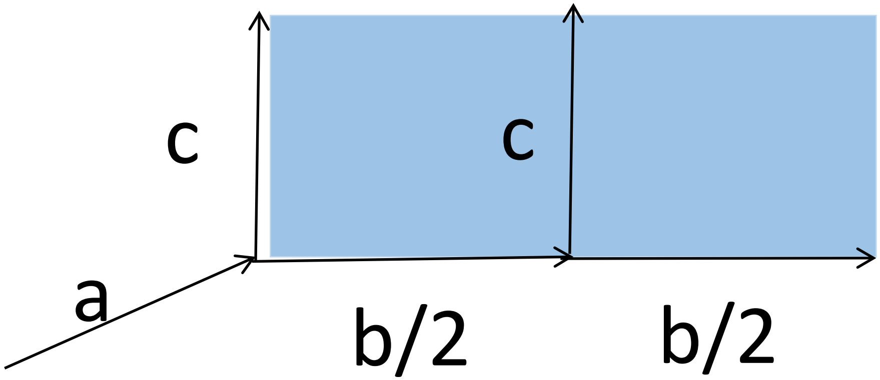


```
over (p,q) (a,b,c) =
  p(a,b,c) U q(a,b,c)
```

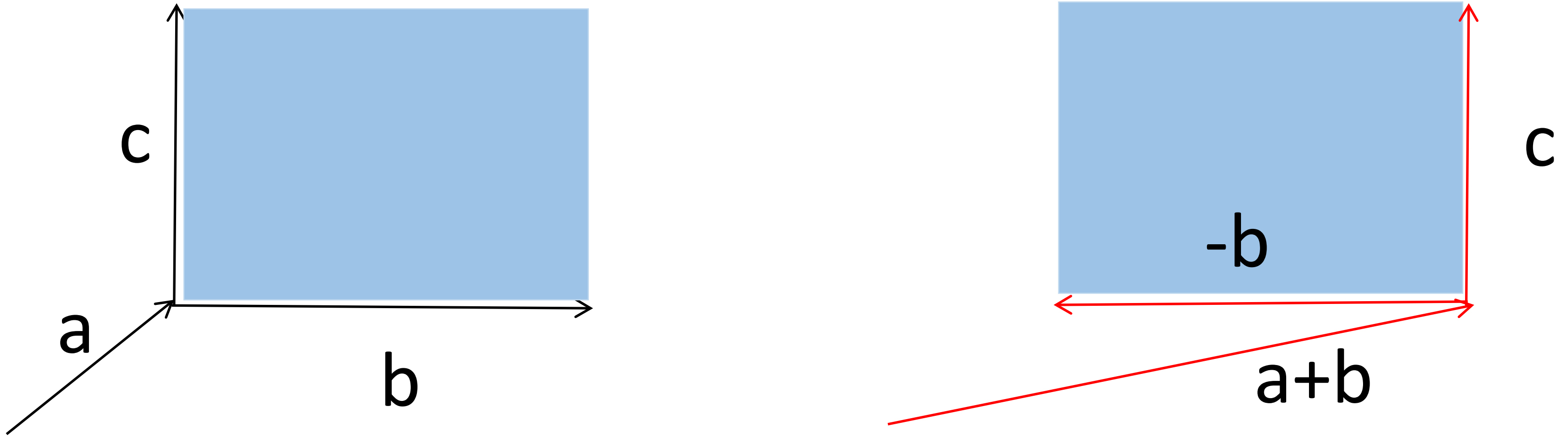
```
over (p,q) (a,b,c) =
  p(a,b,c) U q(a,b,c)

beside (p,q) (a,b,c) =
  p(a,b/2,c) U q(a+b/2,b/2,c)
```



```
over (p,q) (a,b,c) =
  p(a,b,c) U q(a,b,c)

beside (p,q) (a,b,c) =
  p(a,b/2,c) U q(a+b/2,b/2,c)
```



```
rot(p) (a,b,c) = p(a+b,c,-b)
```



Laws

```
rot(above(p,q))
=
beside(rot(p),rot(q))
```

It seems there is a positive correlation between the simplicity of the rules and the quality of the algebra as a description tool.

Whole values

Combining forms

Algebra as litmus test

Whole values

Combining forms

Algebra as litmus test

functions as representations

Haskell vs. Ada vs. C++ vs. Awk vs. ...
An Experiment in Software Prototyping Productivity*

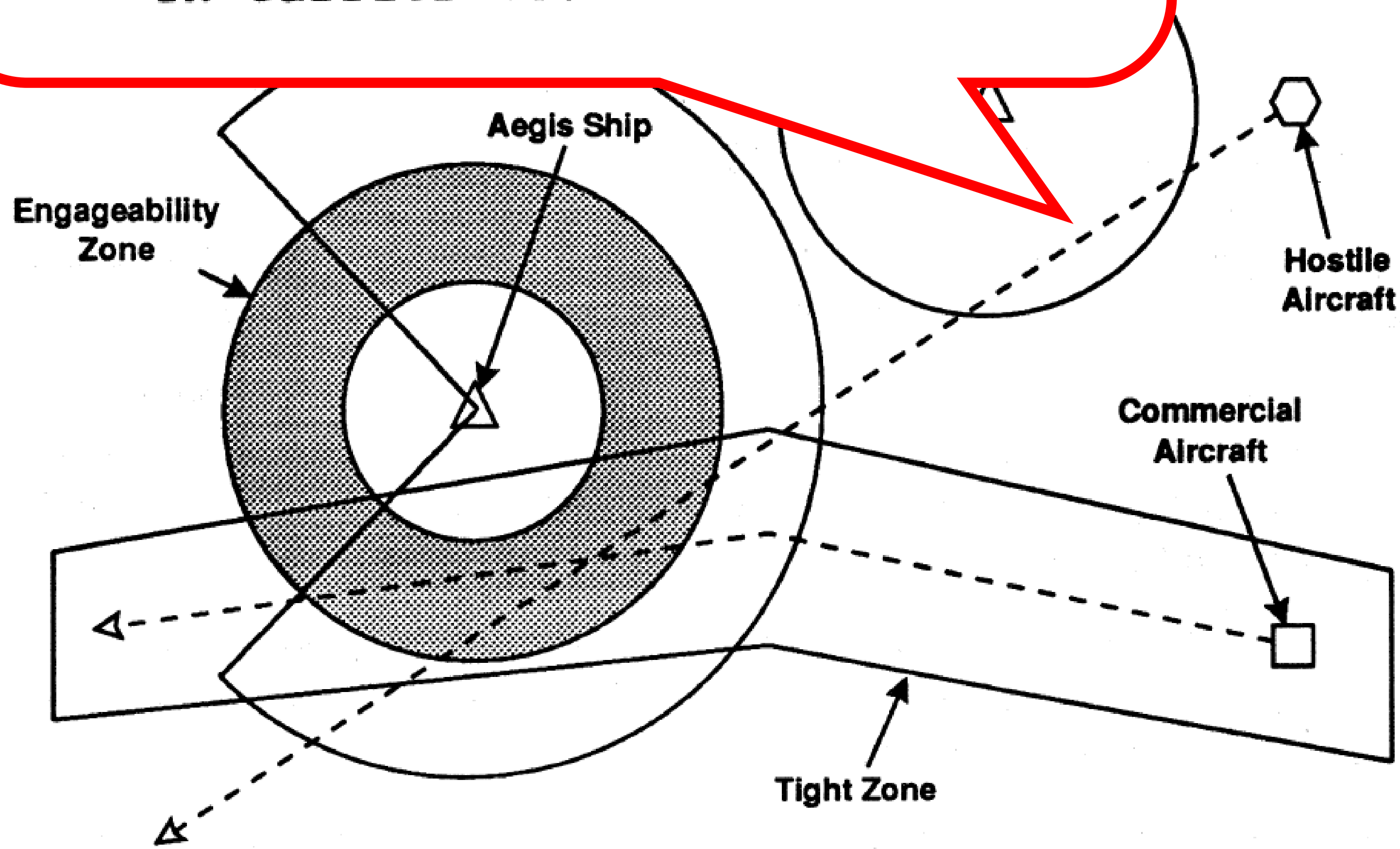
Paul Hudak
Mark P. Jones

Yale University
Department of Computer Science
New Haven, CT 06518
{hudak-paul,jones-mark}@cs.yale.edu

July 4, 1994

er

```
Time 40.0:
commercial aircraft: (100.0,43.0)
 -- In engageability zone
 -- In tight zone
hostile craft: (210.0,136.0)
 -- In carrier slave doctrine
```



Functions as Data

```
> type Region  =  Point -> Bool

> circle    :: Radius -> Region
> outside   :: Region -> Region
> (/\)      :: Region -> Region -> Region

> annulus       :: Radius -> Radius -> Region
> annulus r1 r2 = outside (circle r1) /\ circle r2
```

Including 29 lines of inferable type signatures/synonyms

A student, given 8 days to learn Haskell, w/o knowledge of Yale group

Language	Lines of code	Lines of docum[illegible]	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	7[illegible]	23
(3) Ada9X	800	[illegible]	28
(4) C++	1105	[illegible]30	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Figure 3: Summary of Prototype Software Development Metrics

Reaction…

”too cute for its own good”

…higher-order functions just a trick, probably not useful in other contexts

Lazy Evaluation (1976)


Henderson and Morris

A lazy evaluator


Friedman and Wise

CONS should not evaluate its arguments

”The Whole Value” can be ∞!

- The *infinite list* of natural numbers

  `[0, 1, 2, 3 …]`

Consumer decides how many to compute

- All the iterations of a function

  `iterate f x = [x, f x, f (f x), …]`

- A consumer for numerical methods

  `limit eps xs =`

  <*first element of* **`xs`** *within* **`eps`** *of its predecessor*>

Some numerical algorithms

- Newton-Raphson square root

```
sqrt a = limit eps (iterate next 1.0)
  where next x = (x + a/x) / 2
```

- Derivatives

[1, 1/2, 1/4, 1/8…]

```
deriv f x =
  limit eps (map slope (iterate (/2) 1.0))
  where slope h = (f (x+h) - f x) / h
```

Same convergence check

Different approximation sequences

Speeding up convergence

The smaller *h* is, the better the approximation

Differentiation

Integration

The right answer

$$A + B*h^n$$

An error term

Eliminating the error term

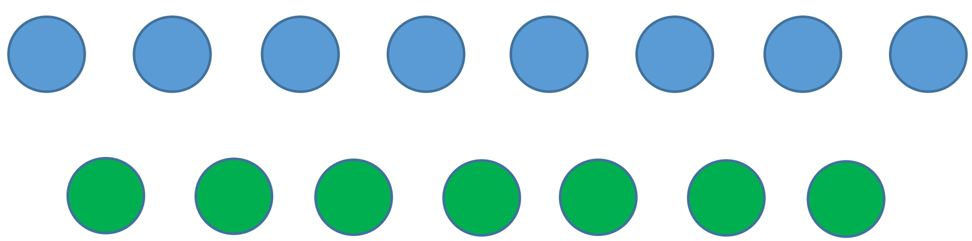
- Given:

$$A + B*h^n$$

$$A + B*(h/2)^n$$

Two successive approximations

- Solve for A and B!



`improve n xs` *converges faster than* **`xs`**

Really fast derivative

The convergence check

```
deriv f x =
  limit eps
    (improve 2
      (improve 1
        (map slope (iterate (/2) 1.0))))
```

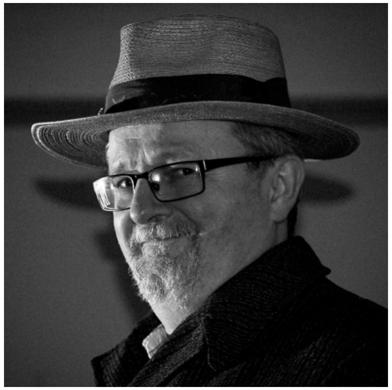
The improvements

The approximations

Everything is programmed *separately* and easy to understand—thanks to ”whole value programming”

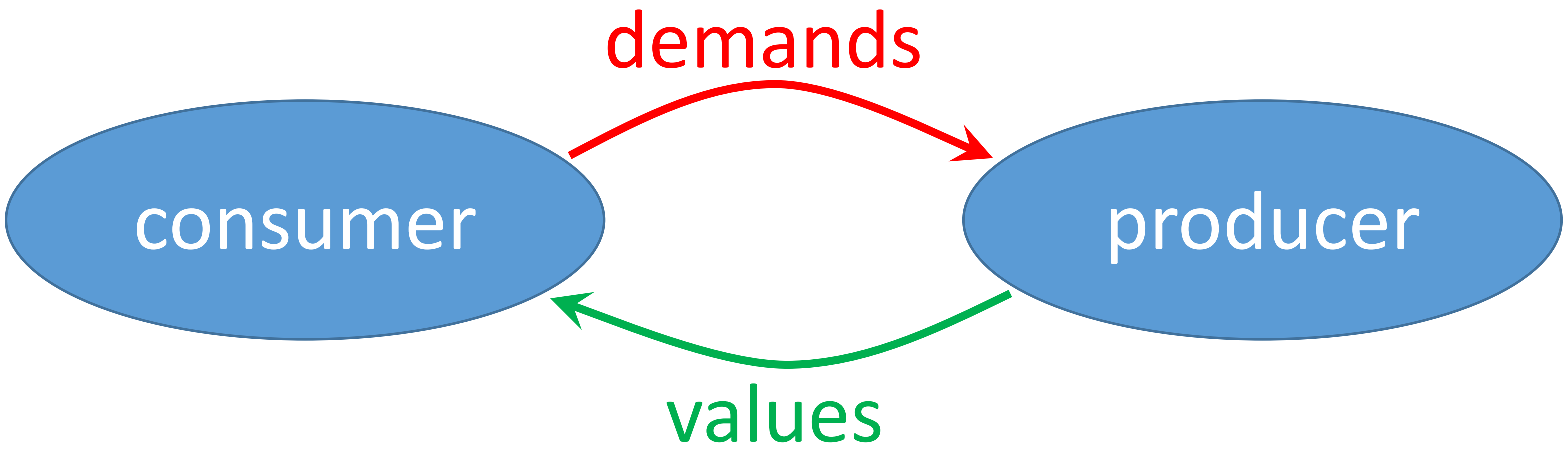
Why Functional Programming Matters

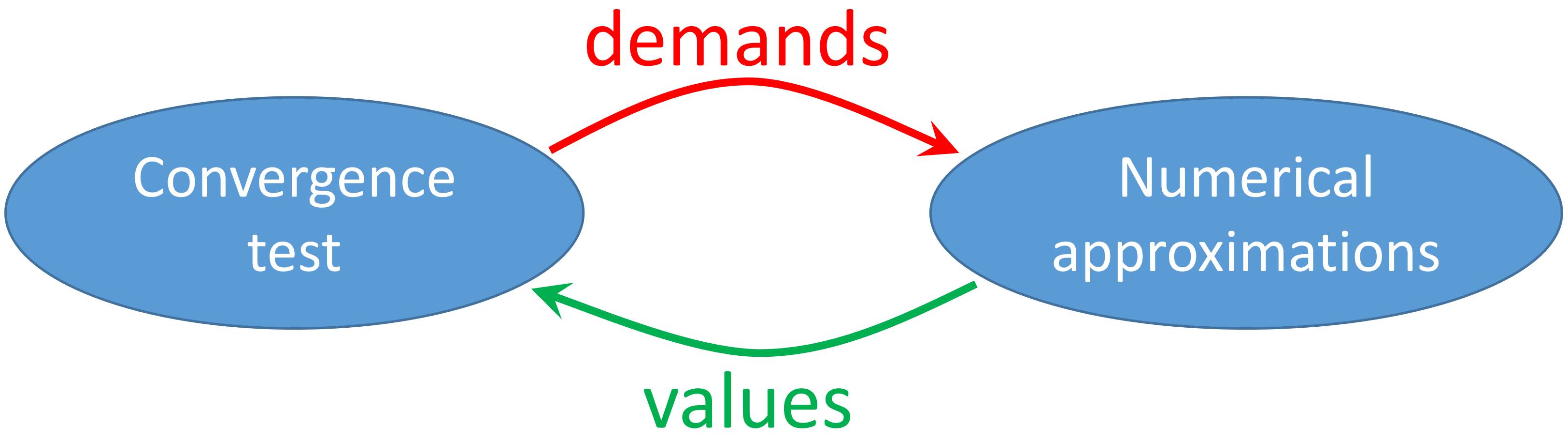
John Hughes
The University, Glasgow



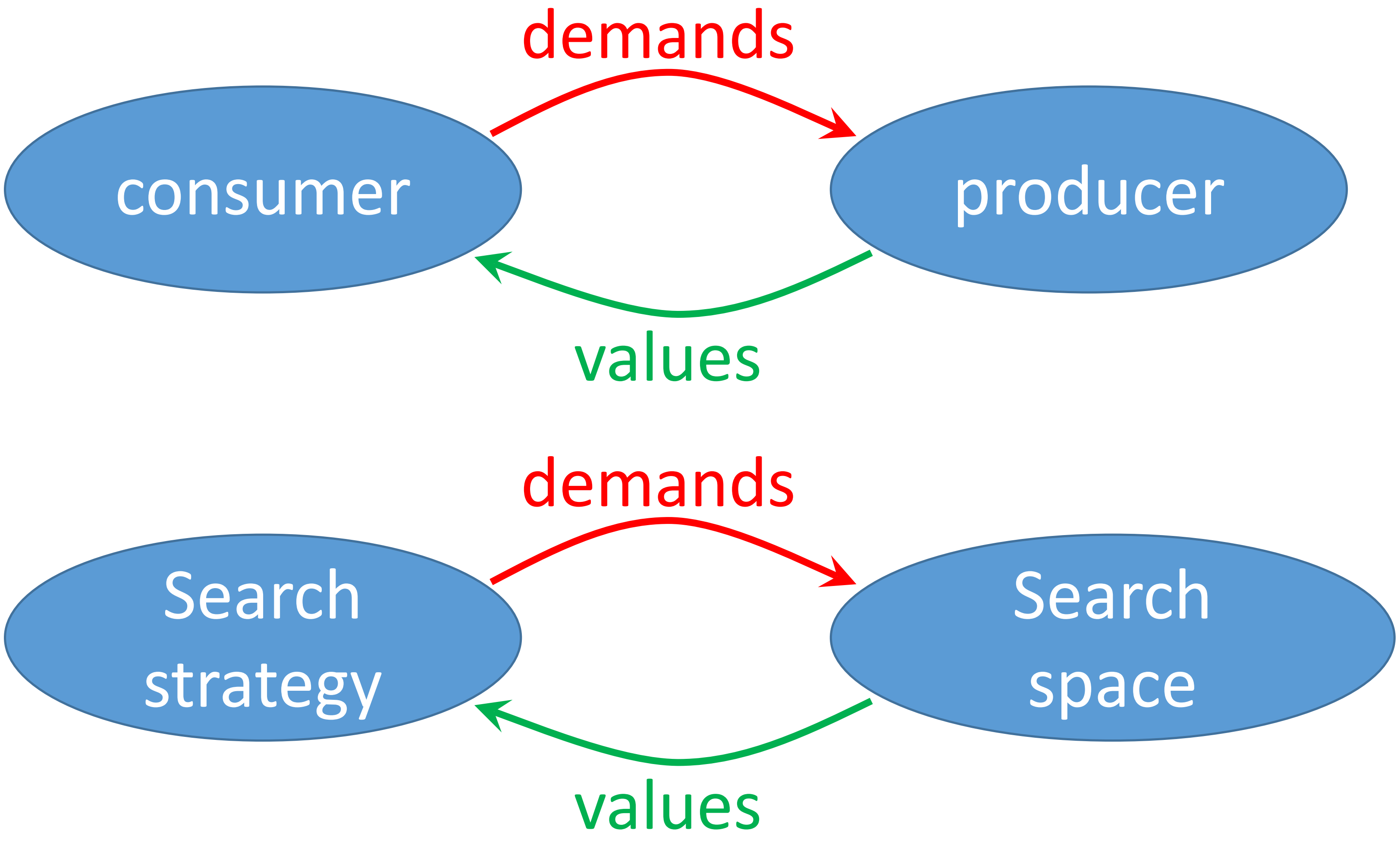
1990

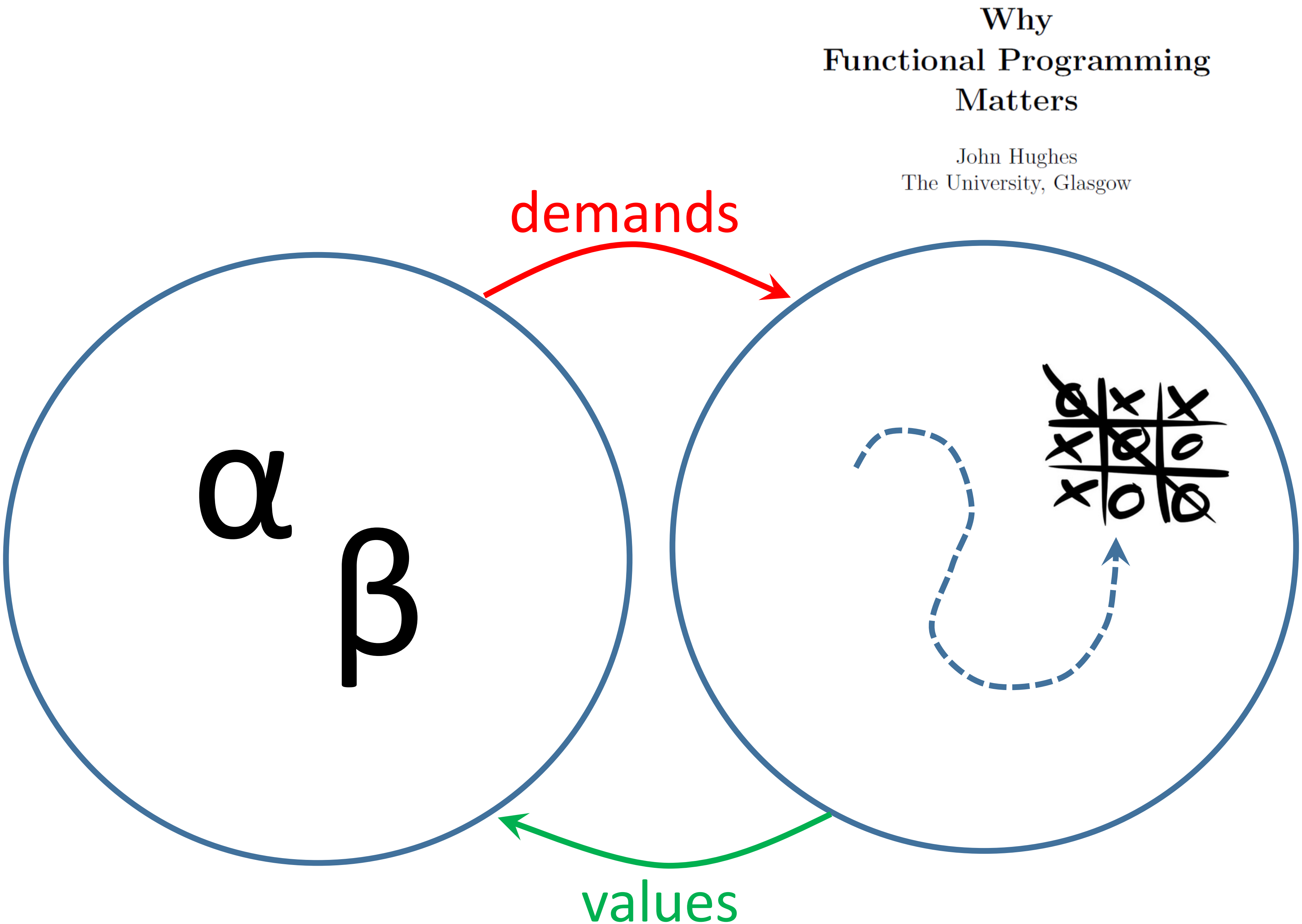
Lazy producer-consumer



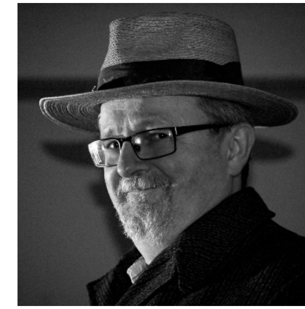


Lazy producer-consumer




Why
Functional Programming
Matters
John Hughes
The University, Glasgow
demands
α
β
values

QuickCheck:
A Lightweight Tool for Random Testing of Haskell Programs



Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

```
prop_reverse() ->
  ?FORALL(Xs,list(int()),
          reverse(reverse(Xs)) == Xs).
```

```
3> eqc:quickcheck(qc:prop_reverse()).
.....................................................
...............................................
OK, passed 100 tests
true
```

2000

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs

Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

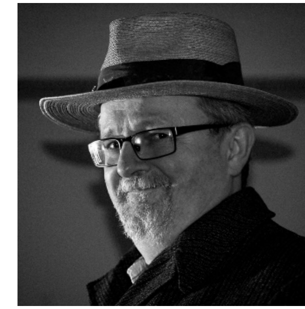
John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se

```
prop_wrong() ->
  ?FORALL(Xs,list(int()),
          reverse(Xs) == Xs).
```

```
4> eqc:quickcheck(qc:prop_wrong()).
Failed! After 1 tests.
[-36,-29,20,31,-47,-63,80,-7,93,-87,-29,33,64,58]
Shrinking xx.x.x..xx(4 times)
[0,1]
false
```

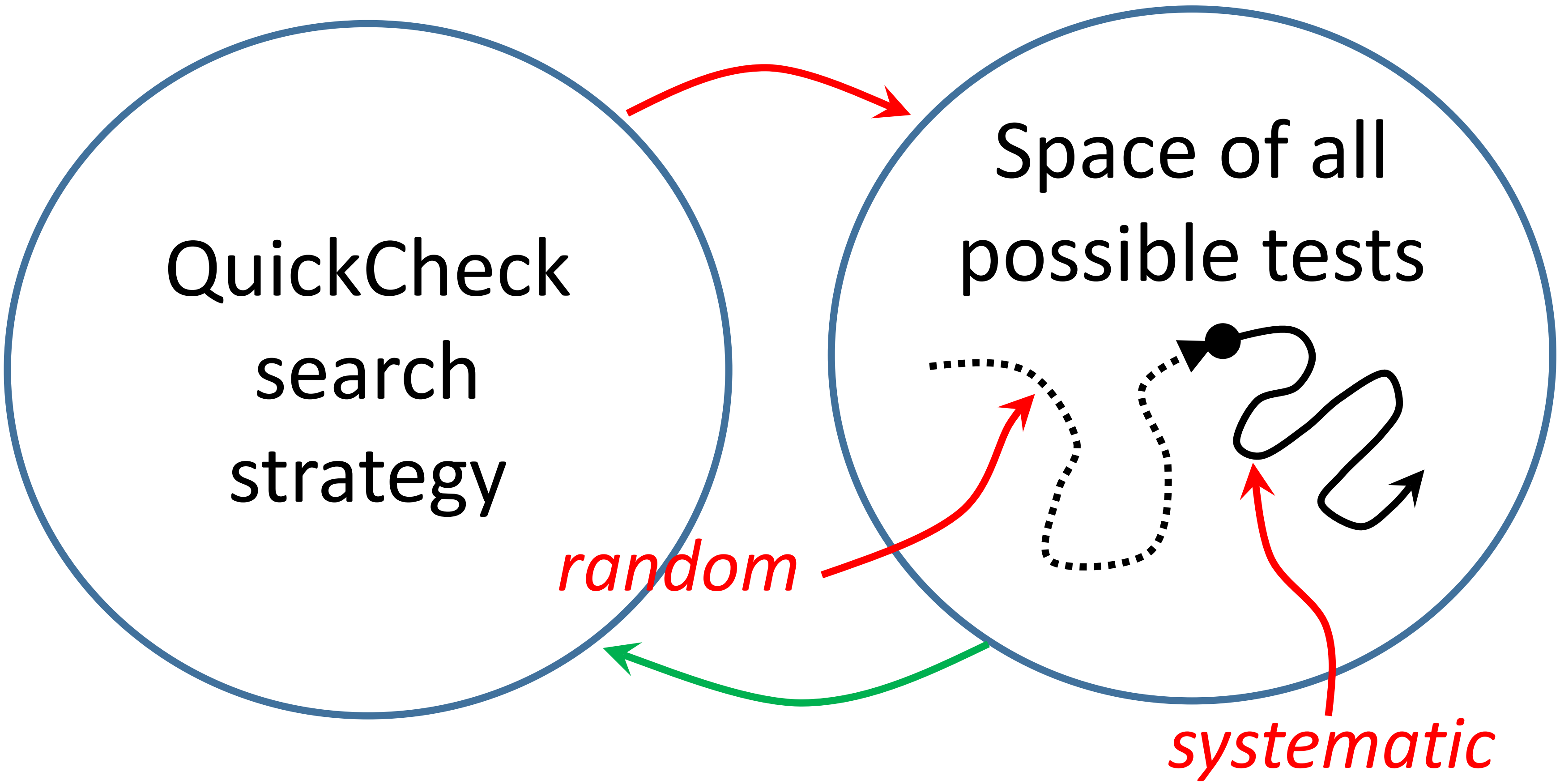
minimal counterexample

QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs



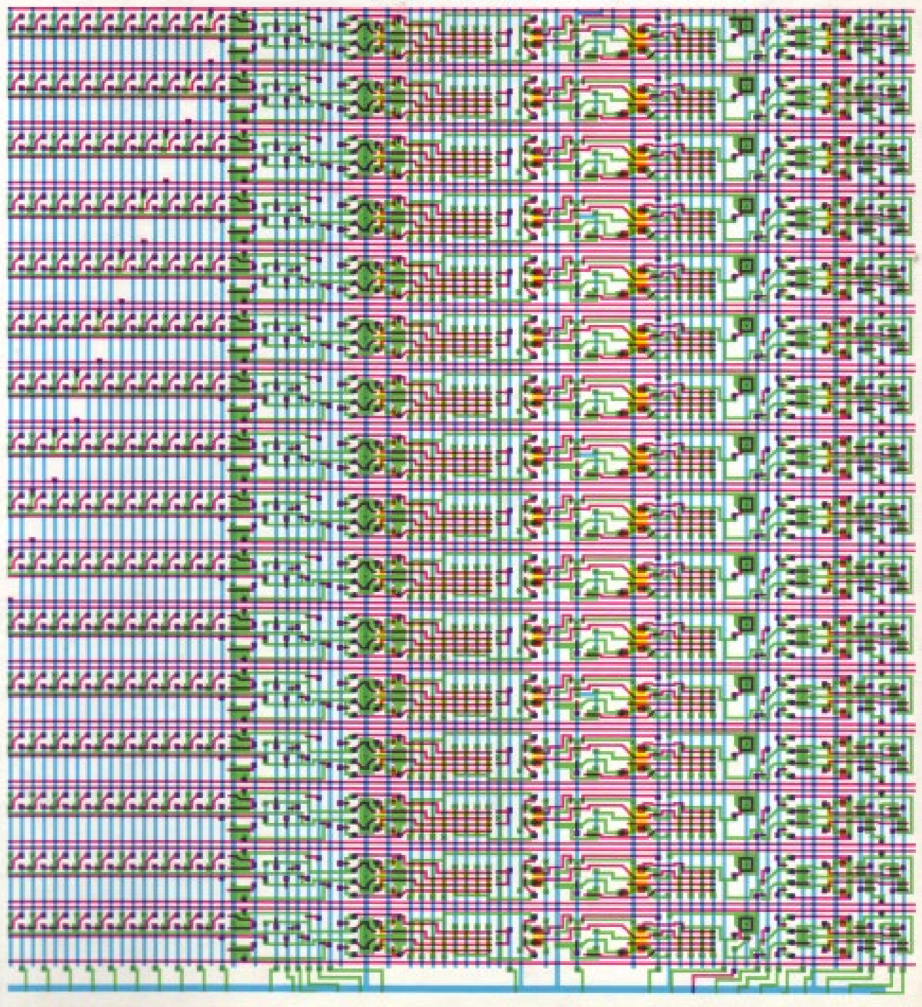
Koen Claessen
Chalmers University of Technology
koen@cs.chalmers.se

John Hughes
Chalmers University of Technology
rjmh@cs.chalmers.se



INTRODUCTION TO VLSI SYSTEMS

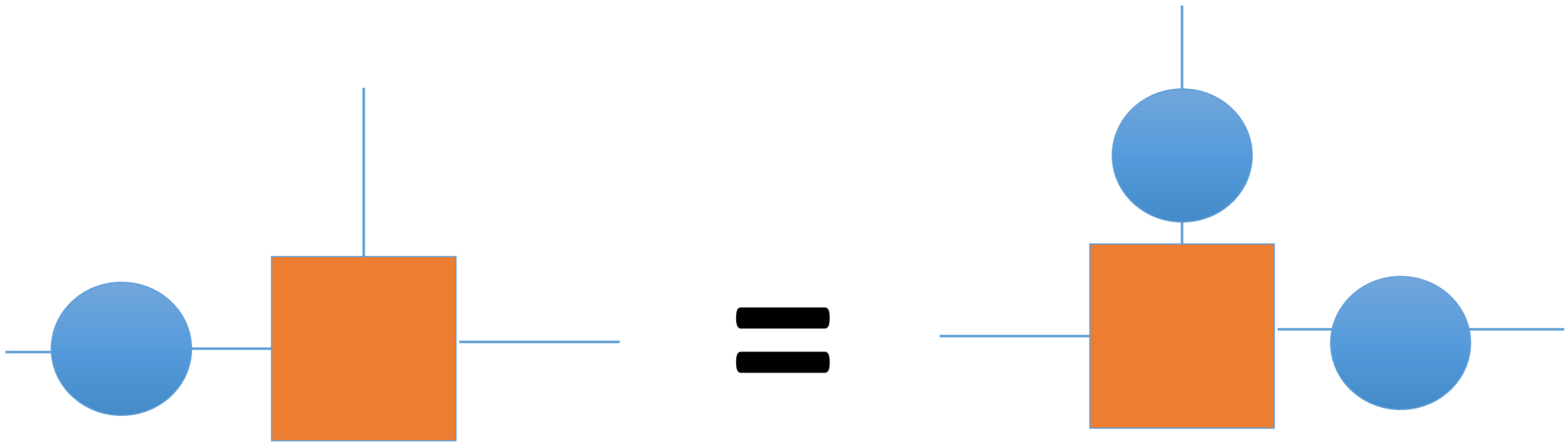
CARVER MEAD • LYNN CONWAY

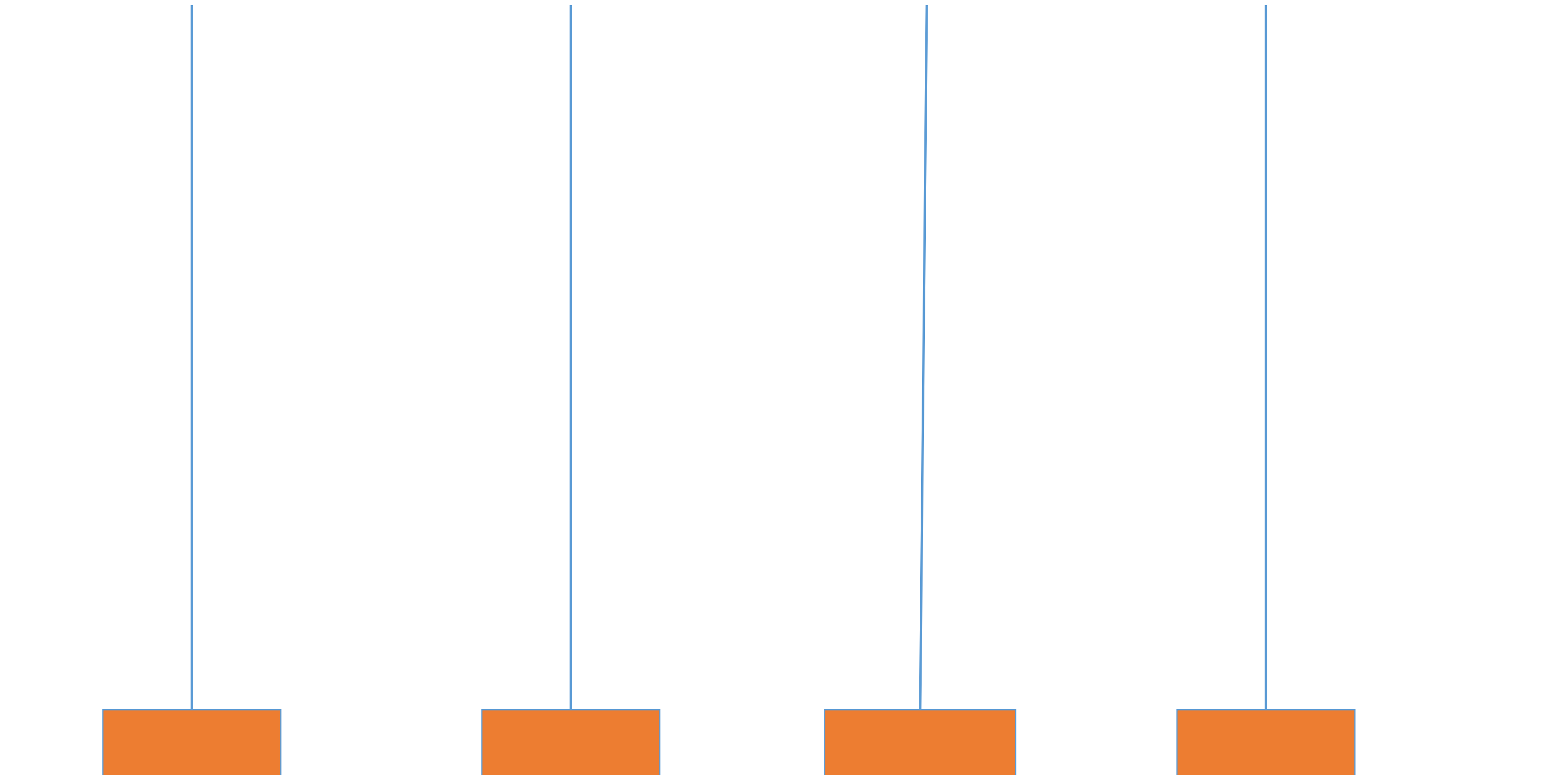


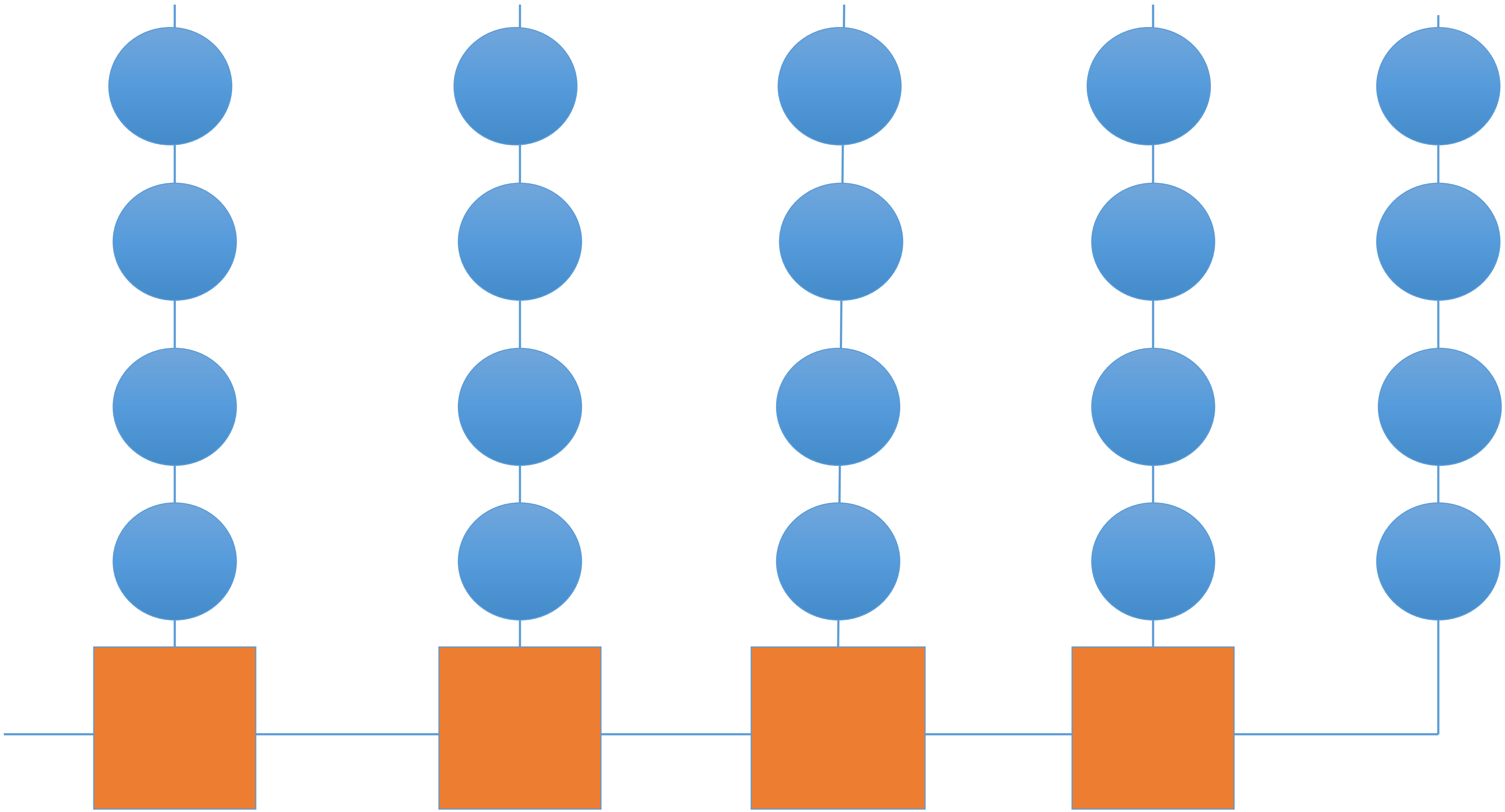
muFP—Circuits as values

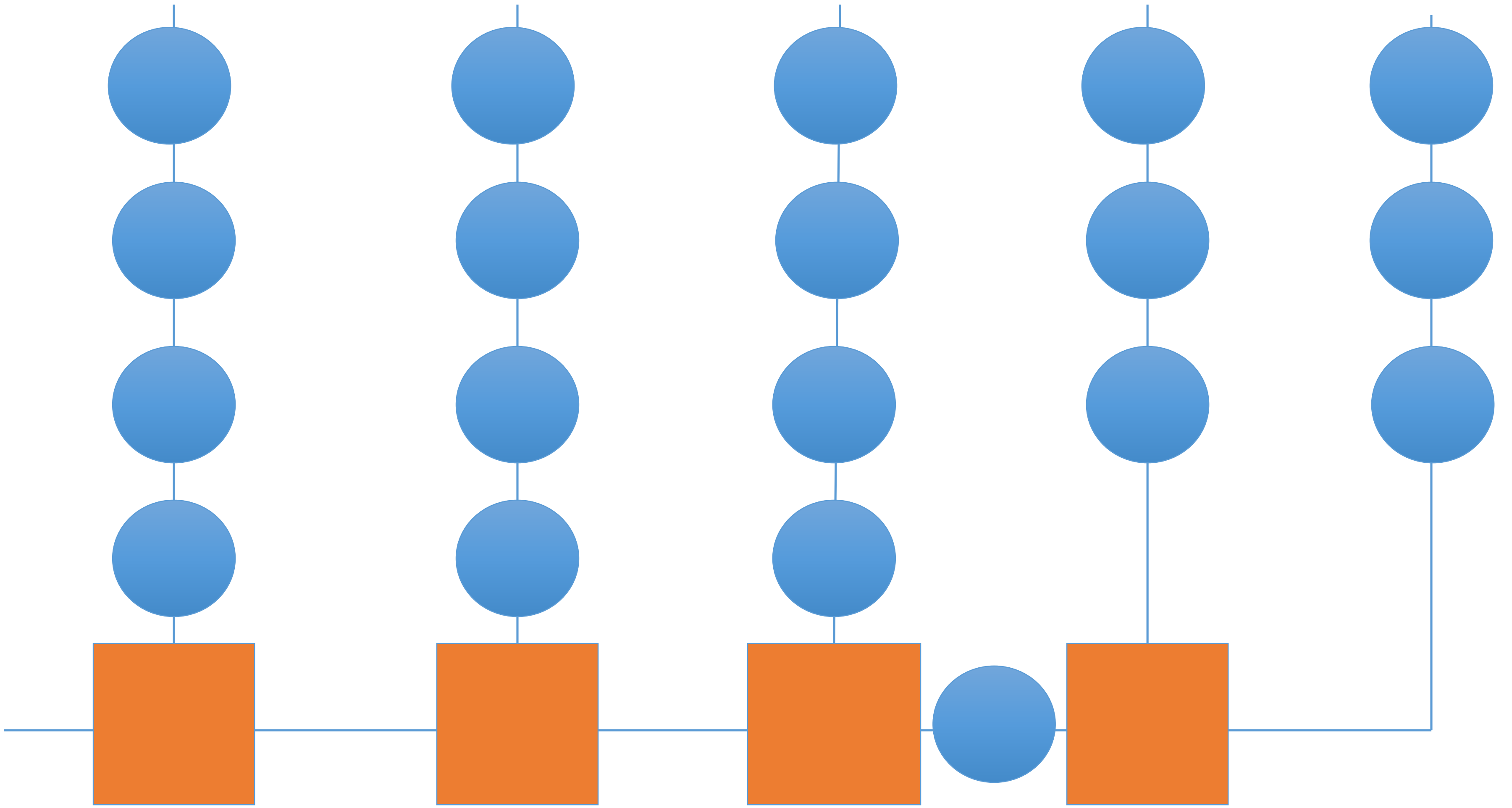
- Backus FP + one-clock-cycle delays
- Inherits many combining forms and laws
- Good for reasoning about alternative designs

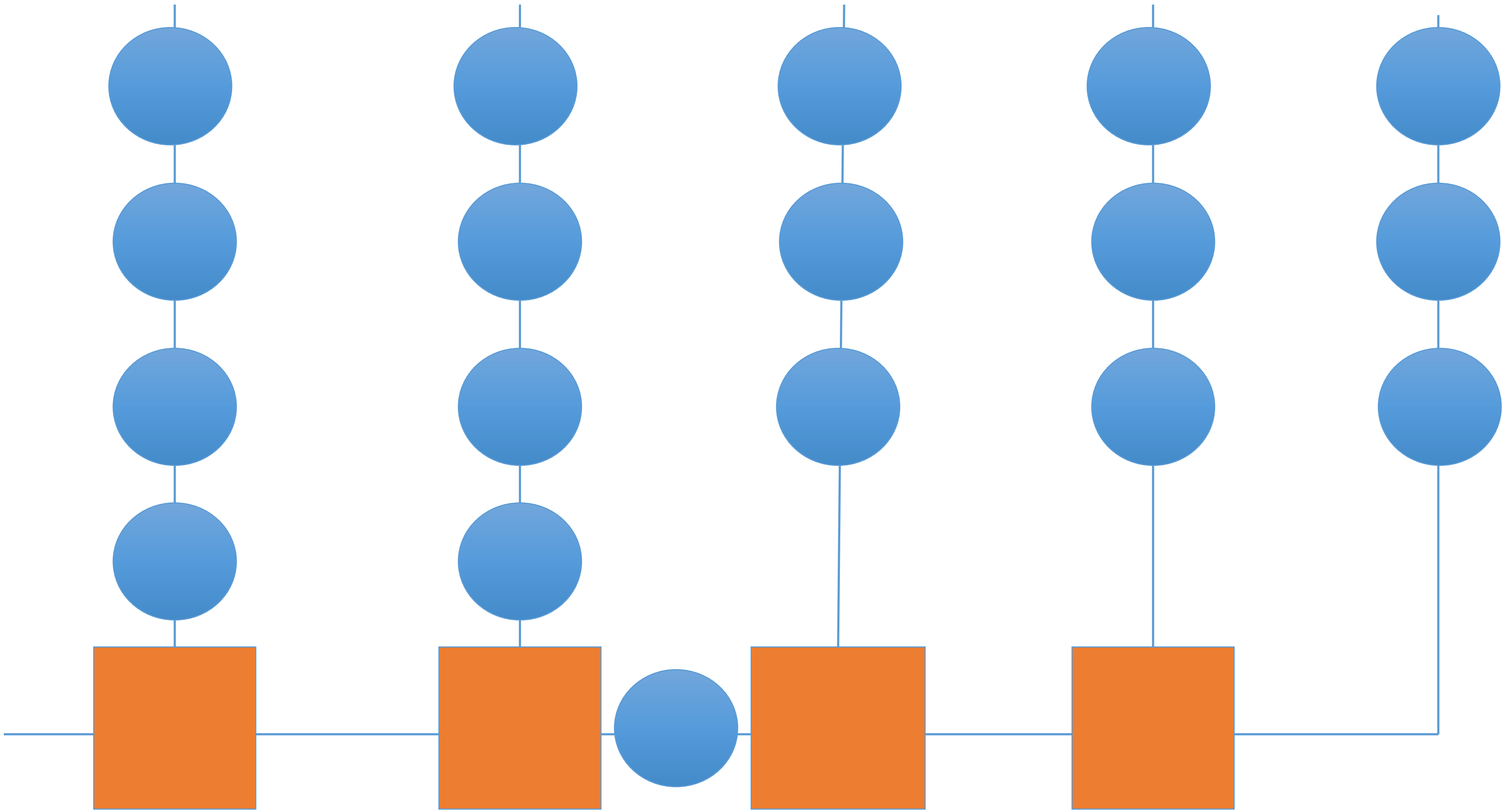
Example law: “retiming”

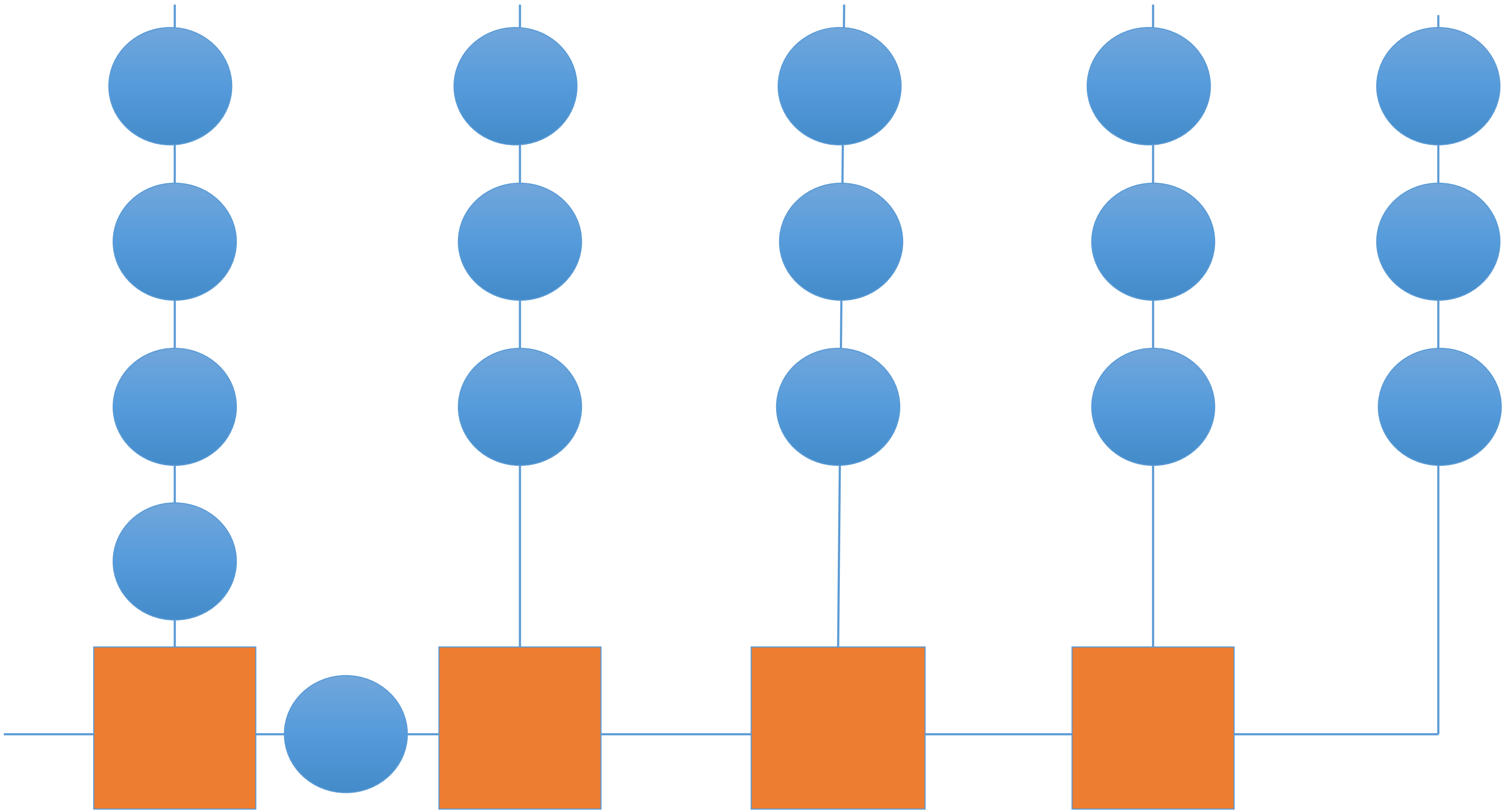


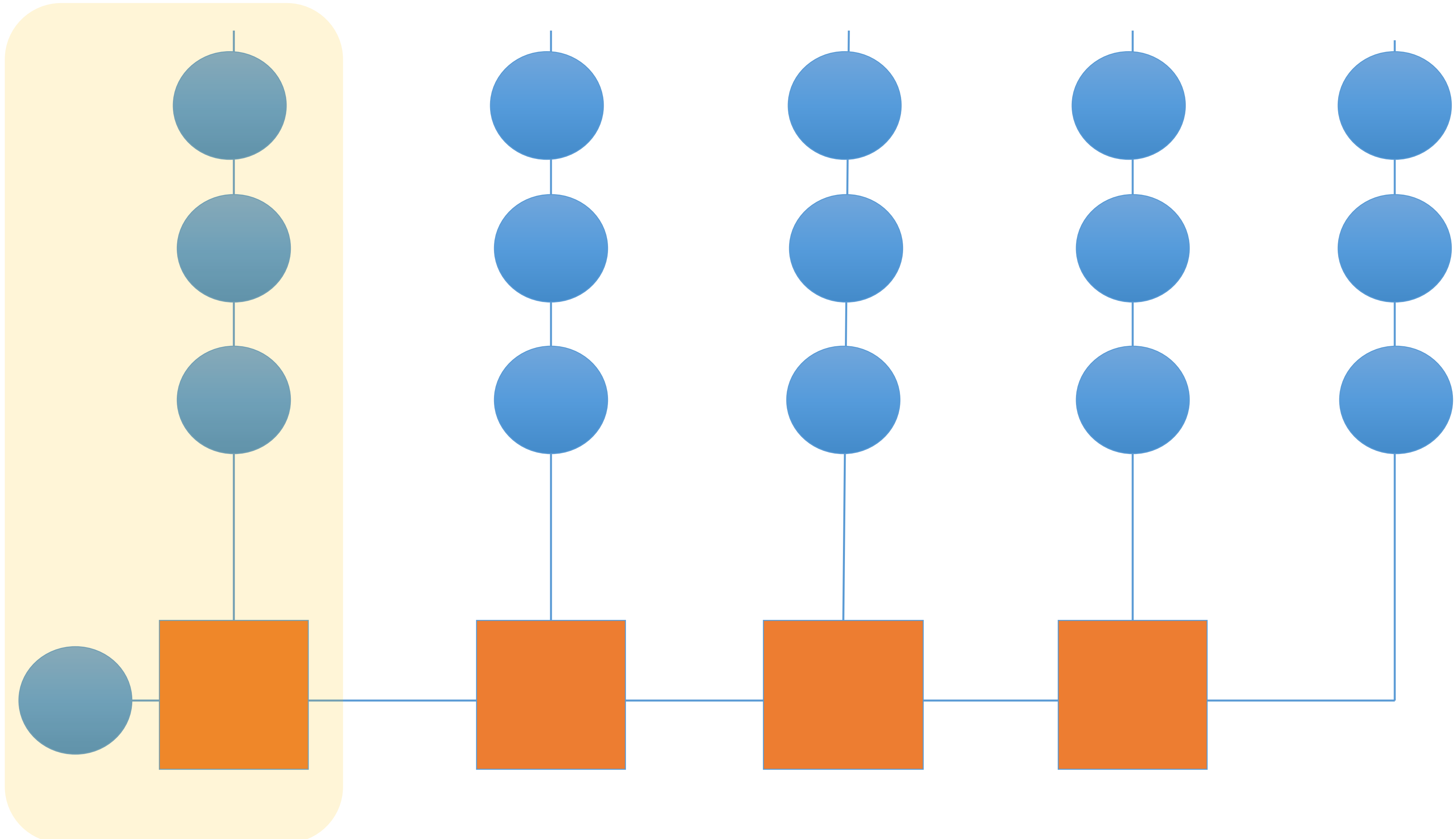


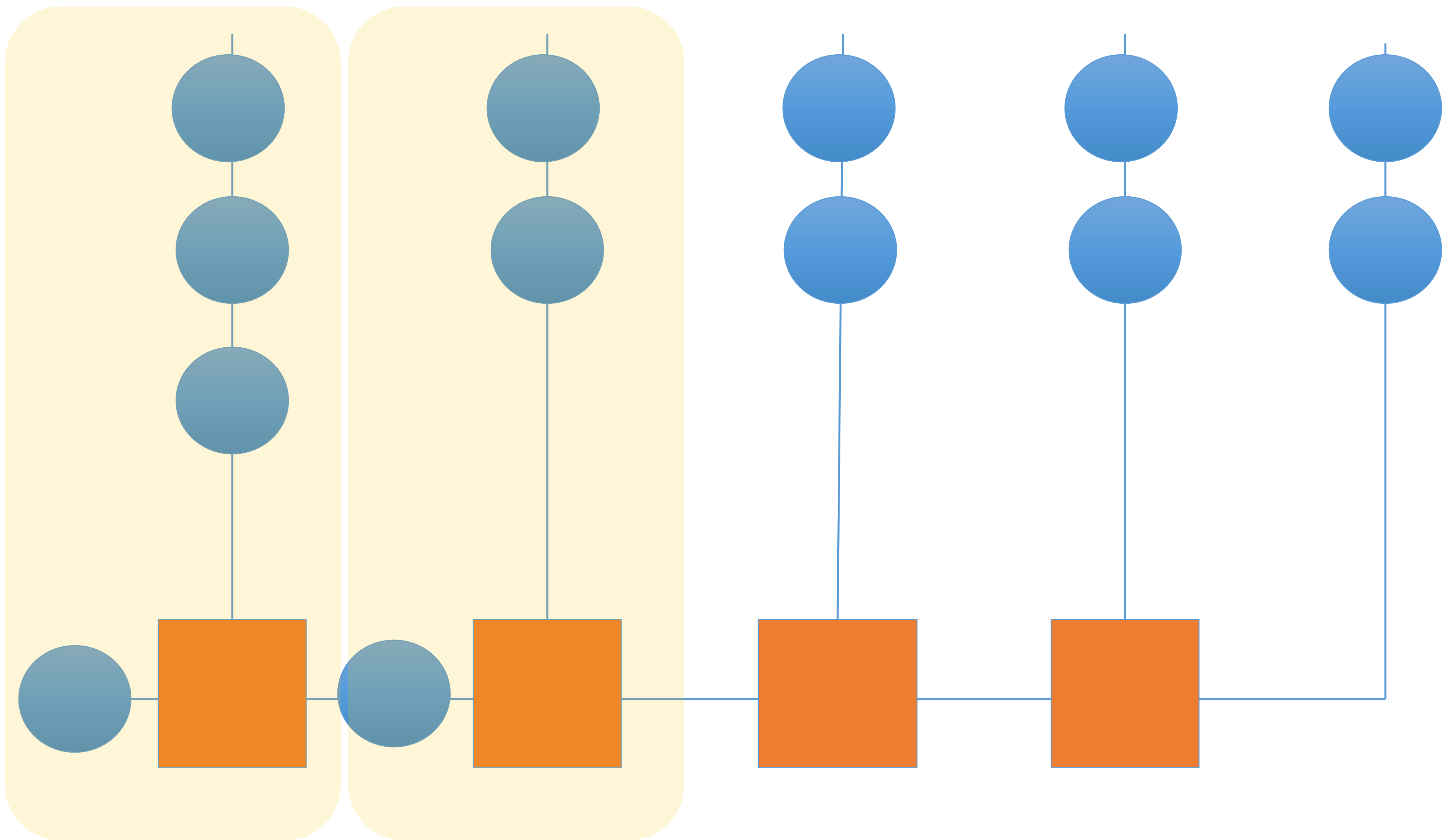


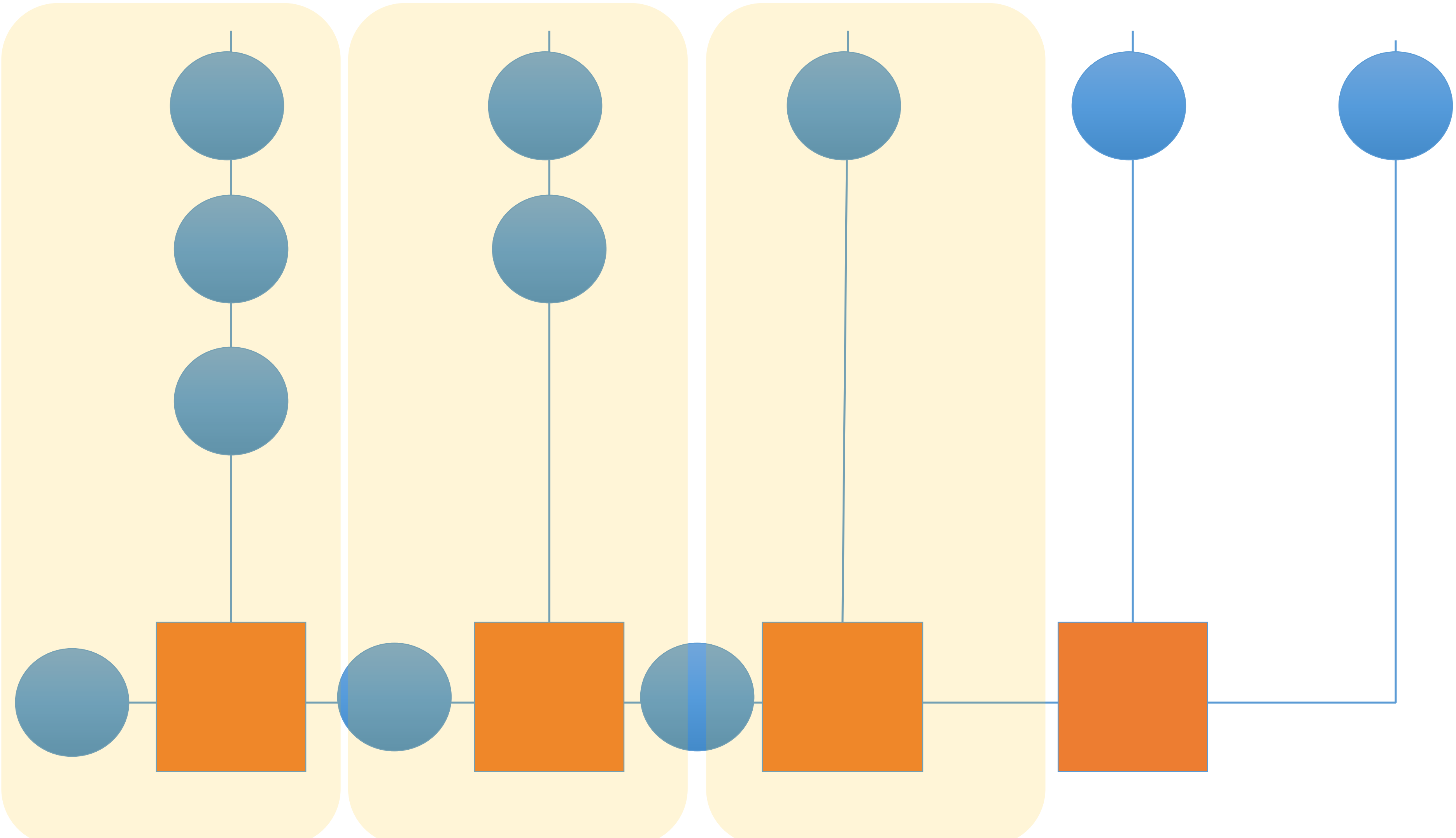


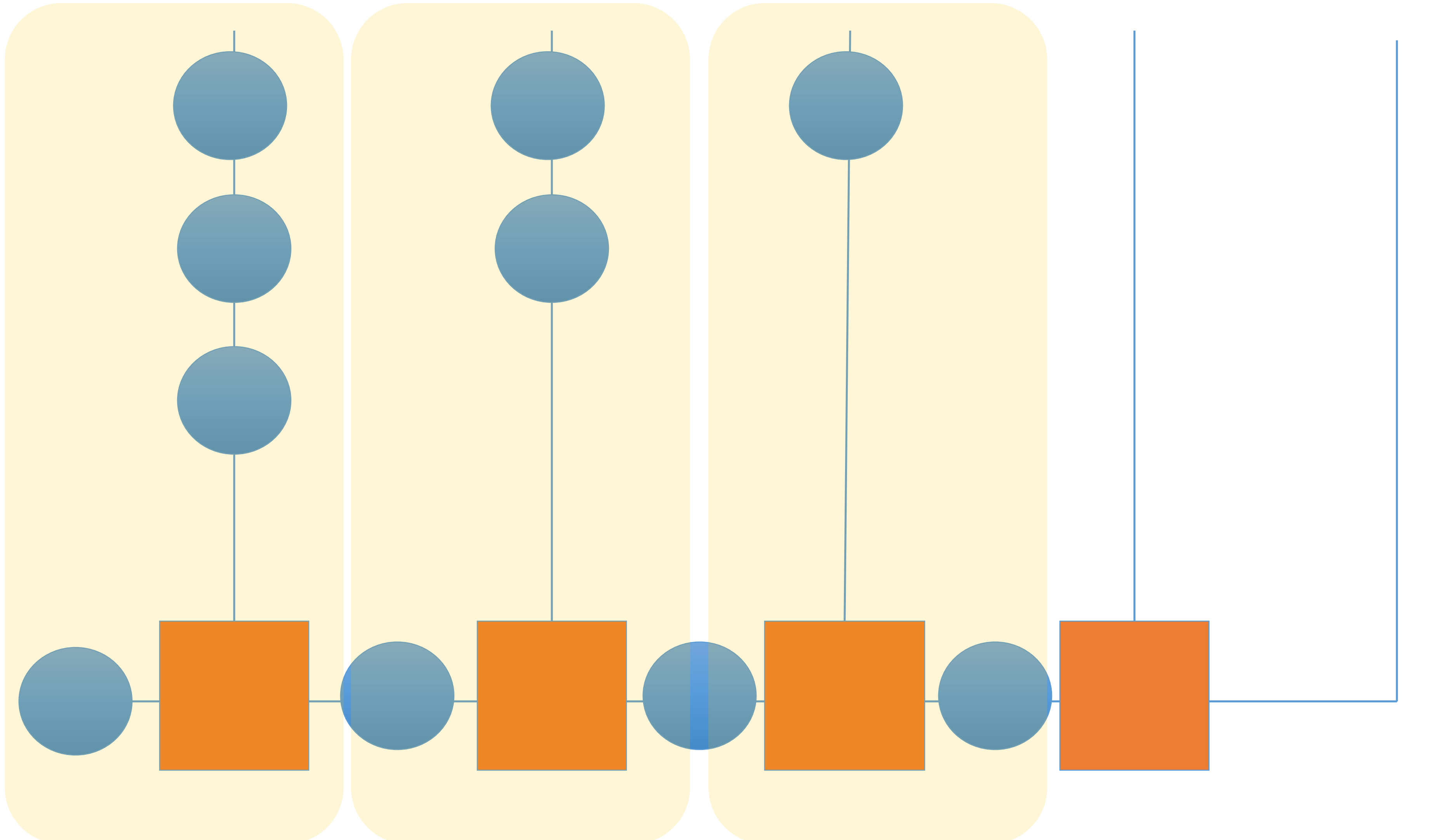












Users!
Plessey video motion estimation

"Using muFP, the array processing element was described in just one line of code and the complete array required four lines of muFP description. muFP enabled the effects of adding or moving data latches within the array to be assessed quickly."

Bhandal et al, An array processor for video picture motion estimation, Systolic Array Processors, 1990, Prentice Hall

work with Plessey done by G. Jones and W. Luk

Lava

muFP + Functional Geometry

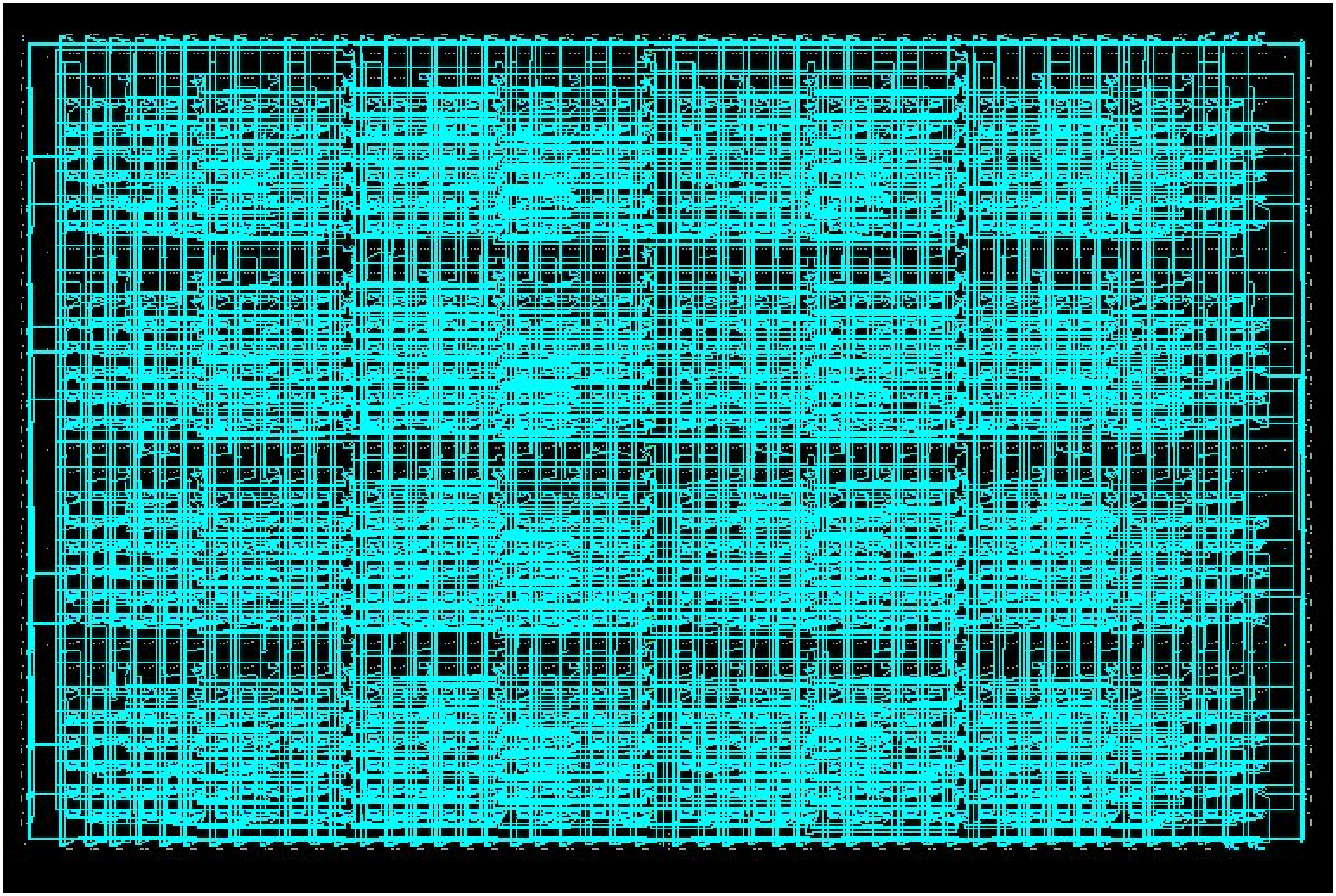
Capture *semantics* of a circuit + relative *placement*

Programmer control of geometry!

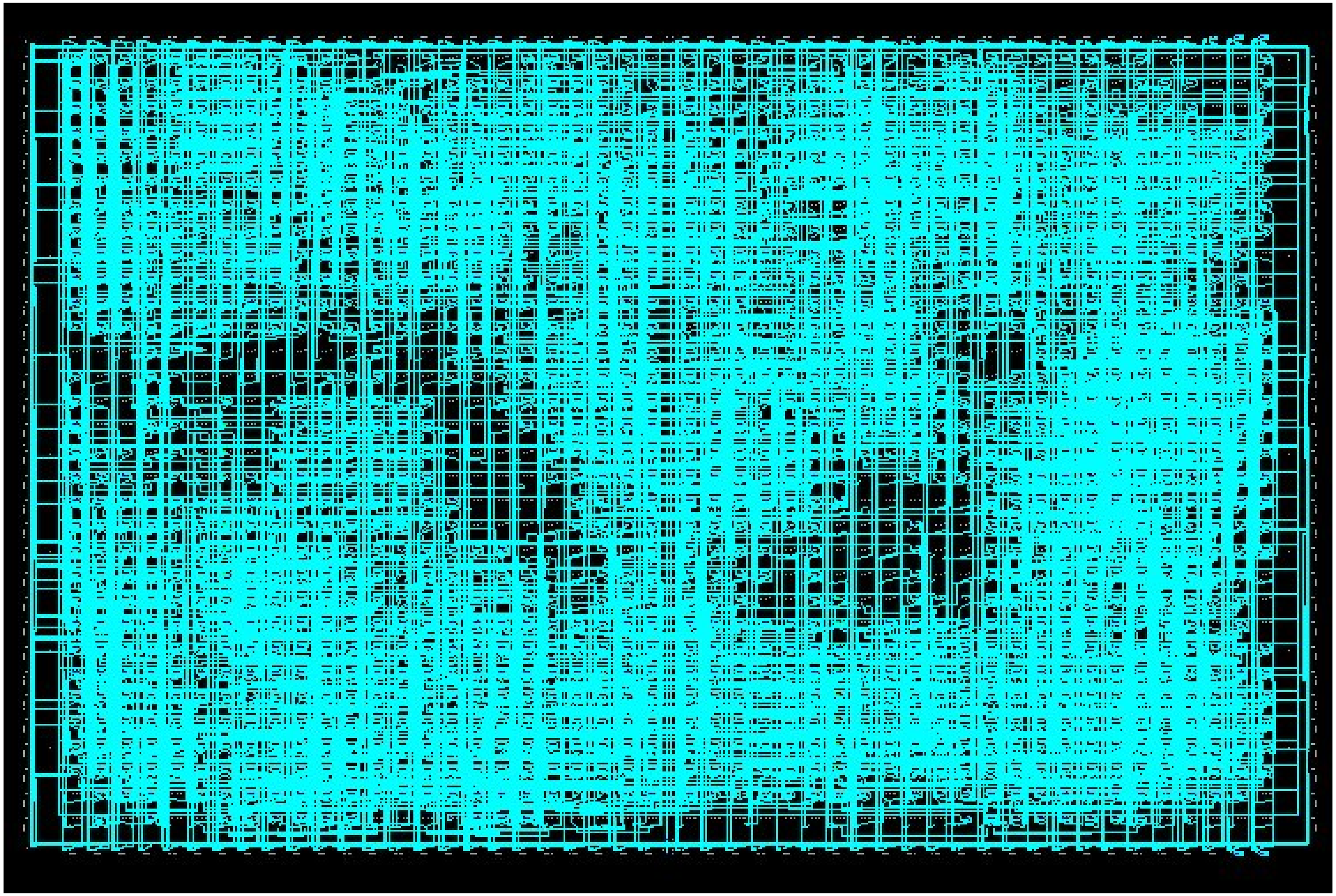
➔ FPGA layouts on Xilinx chips

Satnam Singh
at Xilinx

Four adder trees from Lava



From VHDL, without layout info



Intel

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0

Intel

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 0

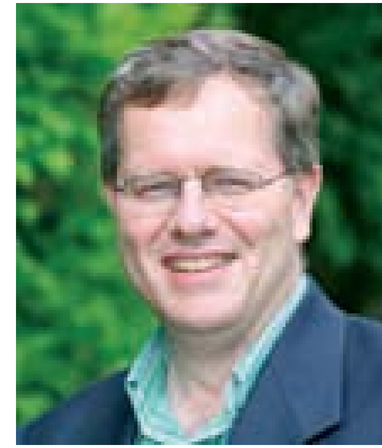
Flawed Pentium

4195835.0 - 3145727.0*(4195835.0/3145727.0) = 256

$475 million


intel inside
pentium
PROCESSOR
Bad companies are destroyed by crises;
good companies survive them;
great companies are improved by them.
— Andy Grove, December 1994

Intel

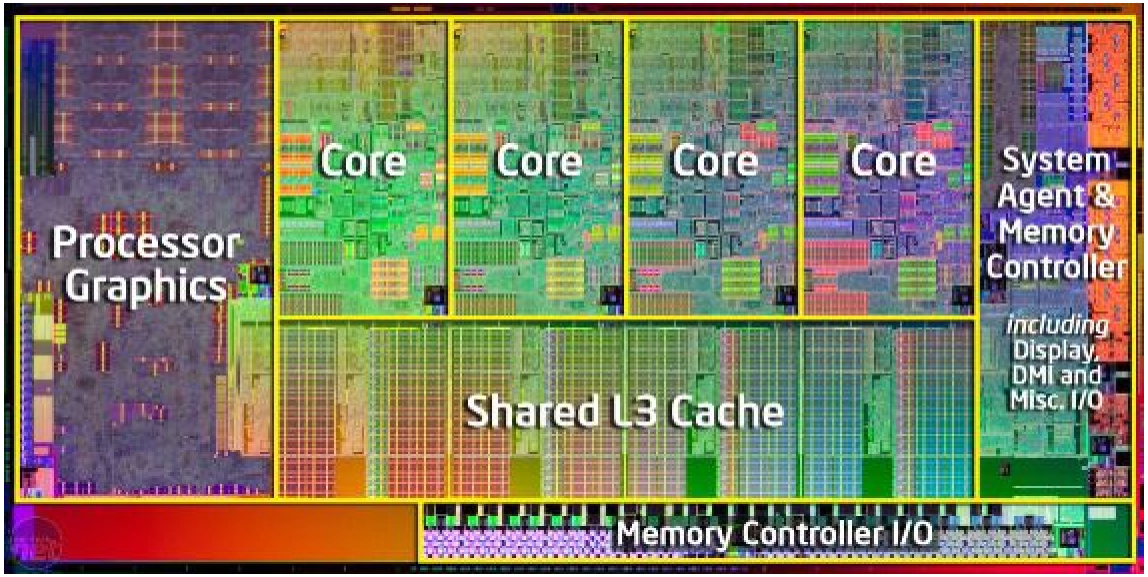


fl —lazy functional language

built-in decision procedures
HW symbolic simulator

Forte System 1000s users

Thanks to Carl Seger (Intel)


Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including Display, DMI and Misc. I/O
Shared L3 Cache
Memory Controller I/O

Google

CARDANO

Scala

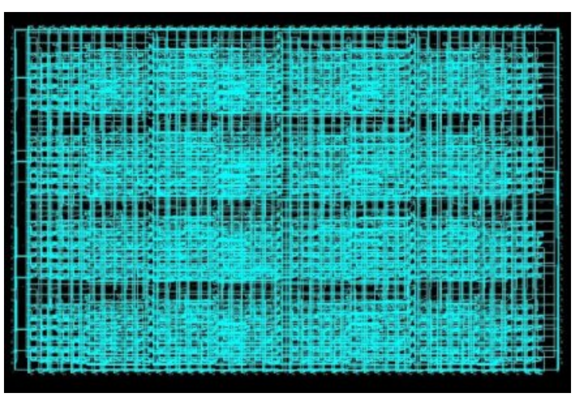
ERLANG

ERICSSON

FP courses

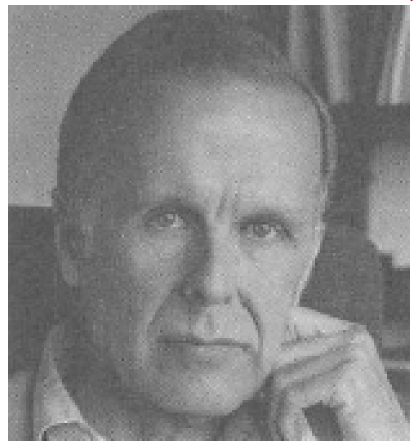
- TDA451 Functional Programming
  - Elective on IT line
- DAT280 Parallel functional programming
  - 3rd year+
  - Haskell, Erlang, emphasis on performance, guest lecturers from industry
- TDA342 Advanced functional programming
  - 3rd year+, MPALG
  - Key ideas behind functional libraries such as Akka

```
two  f x = f (f x)
one  f x = f x
zero f x = x
```



Whole values

Combining forms



Simple laws

Functions as representations