S3, HPI Potsdam, 2008 – 05 – 15

Late-bound object lambda architectures

Ian Piumarta

Viewpoints Research Institute

ian@vpri.org

the big question

for a typical personal computer user, **how hard *should* it be** to connect

- keyboards, various pointing devices, and other sensors
- a nice, big bit-mapped screen
- high-quality sound

to

- a small variety of tasks, mostly simulations of old media (paper, film, recordings)
- a few twists: electronic transferal, hyperlinking, searches, immersive games

computing systems are far too complex

*On the contrary, most of our **systems are much more complicated** than can be considered healthy, and are too **messy and chaotic** to be used in comfort and confidence.*

[...]

*You see, while we all know that unmastered complexity is at the root of the misery, we do not know **what degree of simplicity can be obtained**, nor to what extent the intrinsic complexity of the whole design has to show up in the interfaces. We simply do not know yet the limits of disentanglement. We do not know yet **whether intrinsic intricacy can be distinguished from accidental intricacy**.*

— Edsger W. Dijkstra, CACM 44(3), 2001

powers of meaning

powers of meaning:

- how *much* can you say with how *little*?

two extremes:

all of US case law	3 cubic miles of paper
all electromagnetic phenomena	4 tiny equations

mathematical powers of meaning

Maxwell's equations:

electric charges produce electric fields

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

magnetic monopoles do not exist

$$\nabla \cdot \mathbf{B} = 0$$

changing magnetic fields produce electric fields

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

changing electric currents and fields produce magnetic fields

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t}$$

some common systems and their (approximate) complexity

US case law	3 cubic miles of code
Debian	283 million lines of code
Mac OS X	86 million lines of code
Windows Vista	50 million lines of code
FreeBSD	9 million lines of code
Open Office	10 million lines of code
GTK+ v2	650 thousand lines of code
electromagnetism	4 lines of code

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

— William Gates

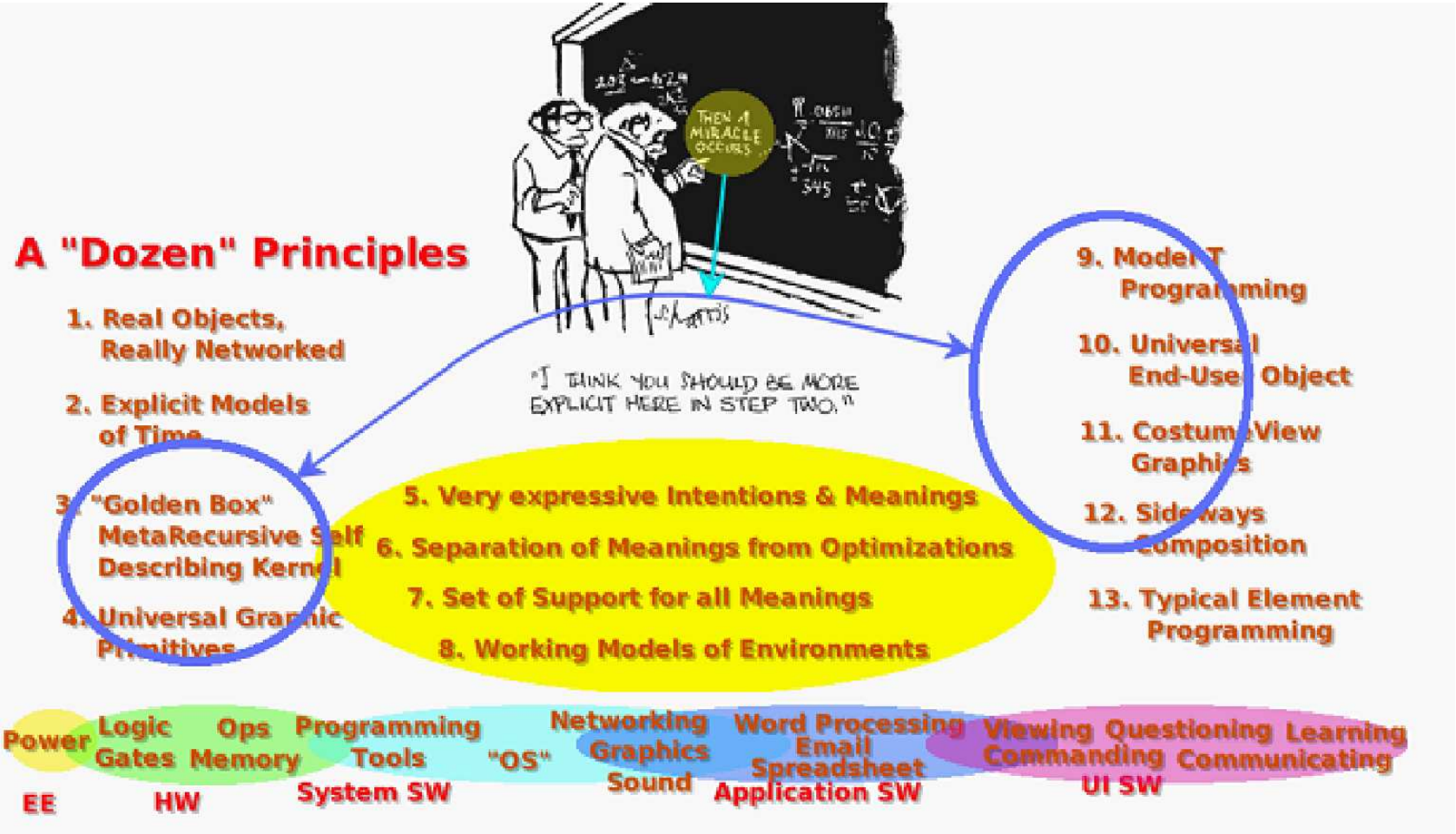
if your system were a text book...

lines	*physical artefact*
2,000	journal article
20,000	400-page book
200,000	encyclopedia
20,000,000	1000-book libary
200,000,000	10,000-book library

we think 20,000 lines is reasonable for all of a modest personal computing system

size ultimately determines accessibility and malleability

and then a miracle occurs



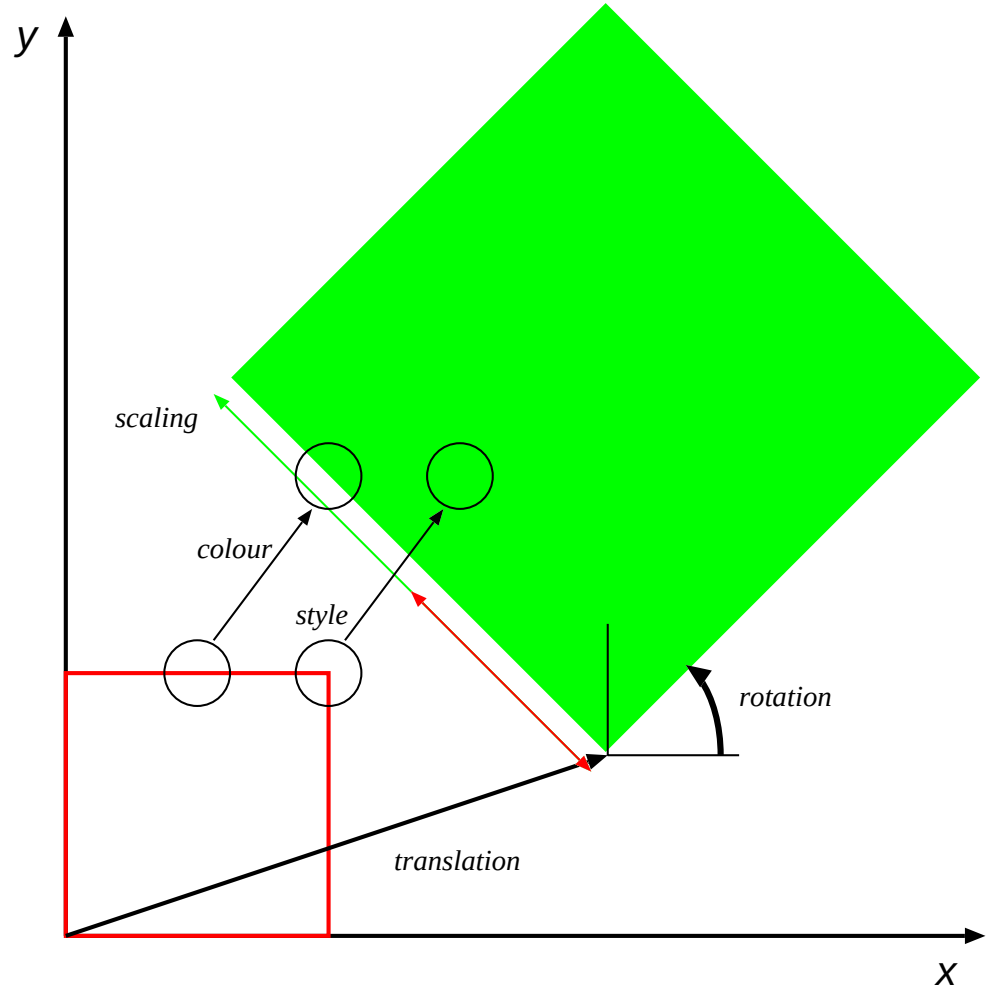
shape + visual transformation = rendering

coordinate

- translation
- rotation
- scaling

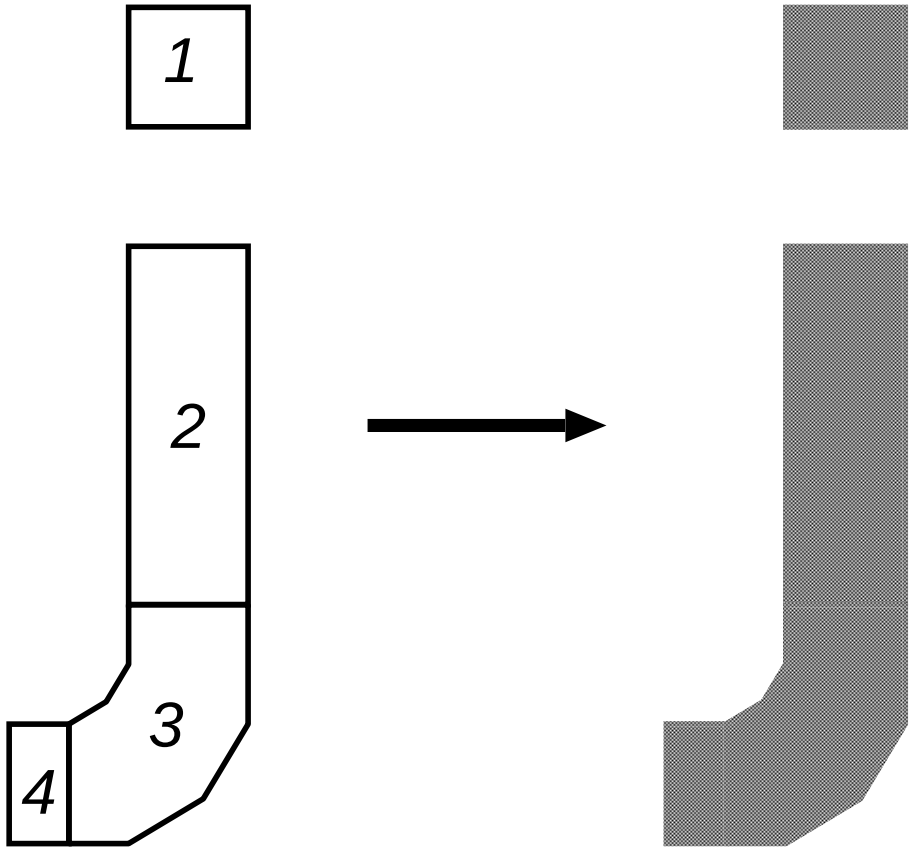
and stylistic

- colour
- drawing style
  - solid fill
  - stroked outline
- . . .



is this fast enough for an interactive user interface?

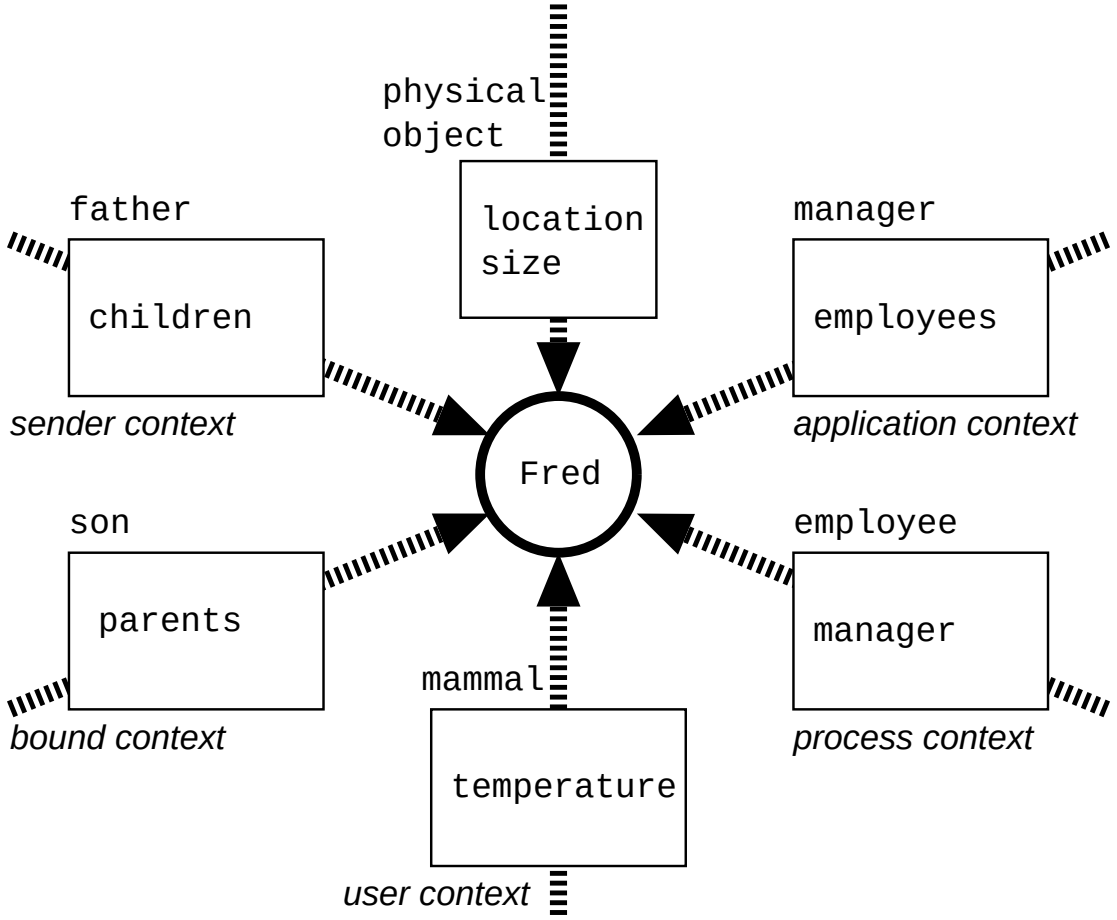
text is just (a lot of) polygons (shapes)



world

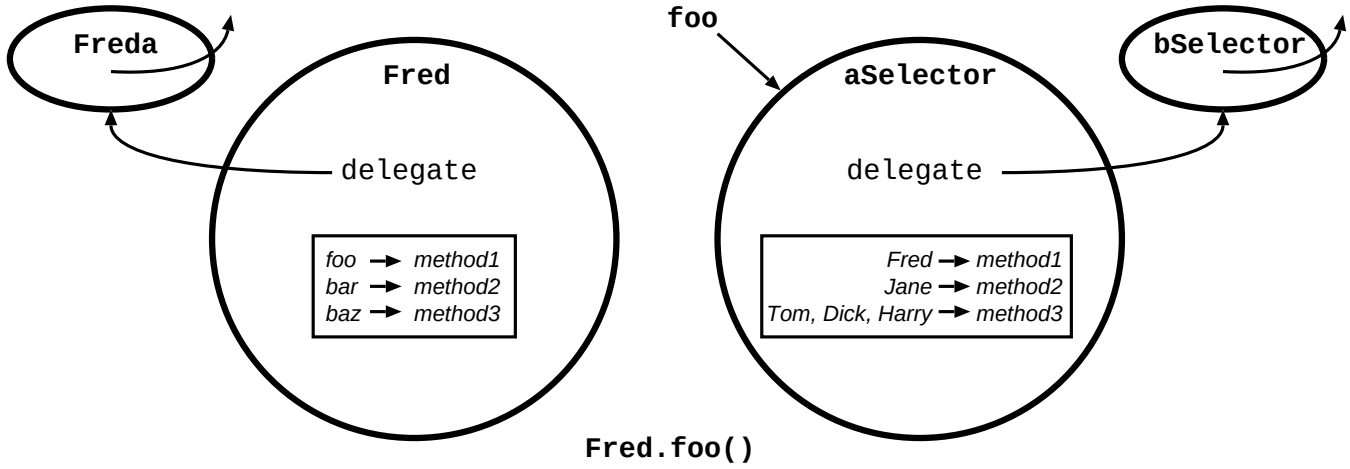
multiple subjective perspectives

why are we satisfied with a single role for each object?



unbreaking essential symmetries

why choose to store behaviours by name in types?



- multiple dispatch
- selector inheritance
- namespaces, capabilities, ...

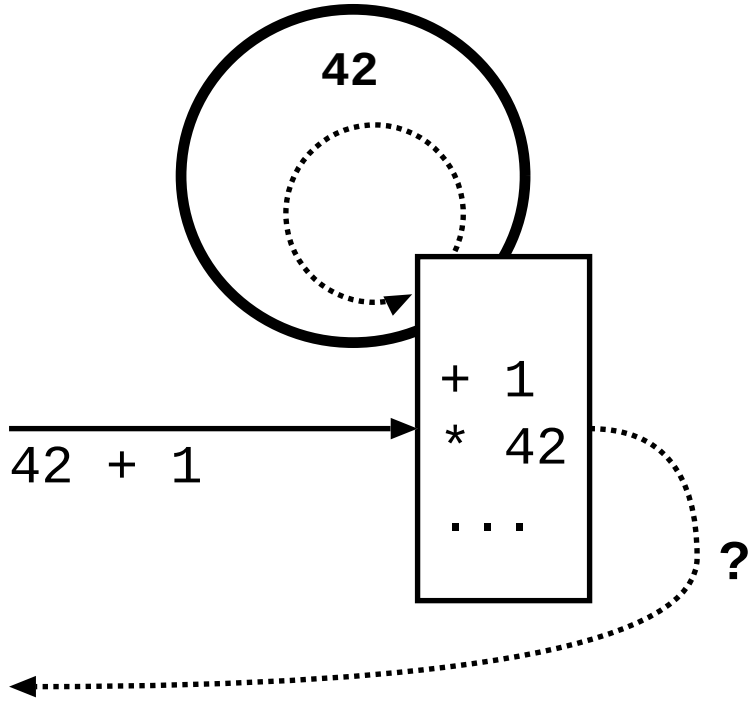
and others...

computation as fields acting on particles

- react to changes in environment, *not* to explicit commands
- publish-subscribe mechanisms
- Fisher's monitors, TEX, ...

Internet all the way down

- scales incredibly well outside the box
  - why not inside the box too?
- messages are requests
- messages can be lost
- requests can be denied/ignored
- responses are asynchronous



pattern-directed transformations

- top-down, bottom-up grammars
- inference

how to waste 1,000,000 lines at the bottom

on the left we need to build something ‘like’ the system before we can begin to build ‘the’ system

> *GTK, which stands for the Gimp ToolKit, is a library for creating graphical user interfaces. It is designed to be **small and efficient**, but still flexible enough to allow the programmer freedom in the interfaces created.*
>
> *GTK+ 2.0 more than **doubles the size** of GTK+ 1.2, moving from 230,000 lines of code to **620,000 lines of code**.*
>
> — GTK+ 2.0 release notes

a curious attitude:

- why is *adding* lines of code an achievement, rather than *removing* them?

typical of many software environments

GType, GObject, GModule, ... lots of other GStuff ...

- these have *nothing* to do with the actual problem being solved
- between 0 and a few hundred LOC would have been reasonable

layered *above* a highly-deficient language

- the solutions cannot be right because the single most fundamental assumption (the programming language) is wrong for the task
- rigidity of C syntax
- rigidity of C runtime
- new paradigms are added at the wrong level of abstraction
  - *over* the language and runtime rather than *in* the language and runtime

tens of thousands of lines of code wasted in adding what should have been *provided for free* by the language and environment, or at least made *very cheap to add* when needed

widespread repurcussions

vertical catastrophe

- the rigid substrate imposes vast, unnecessary complexity throughout the system

horizontal catastrophe

- sheer volume imposes an unrealistic learning curve

effectively *closing the system* to

- understanding
- adaptation
- innovation

by almost all its users

predicting the future

insufficient power of meaning:

- how little they can say with how much code!

big problem: no panacea exists!

- each problem, solution, system is unique
- *impossible* to predict what features to provide or make easy to add

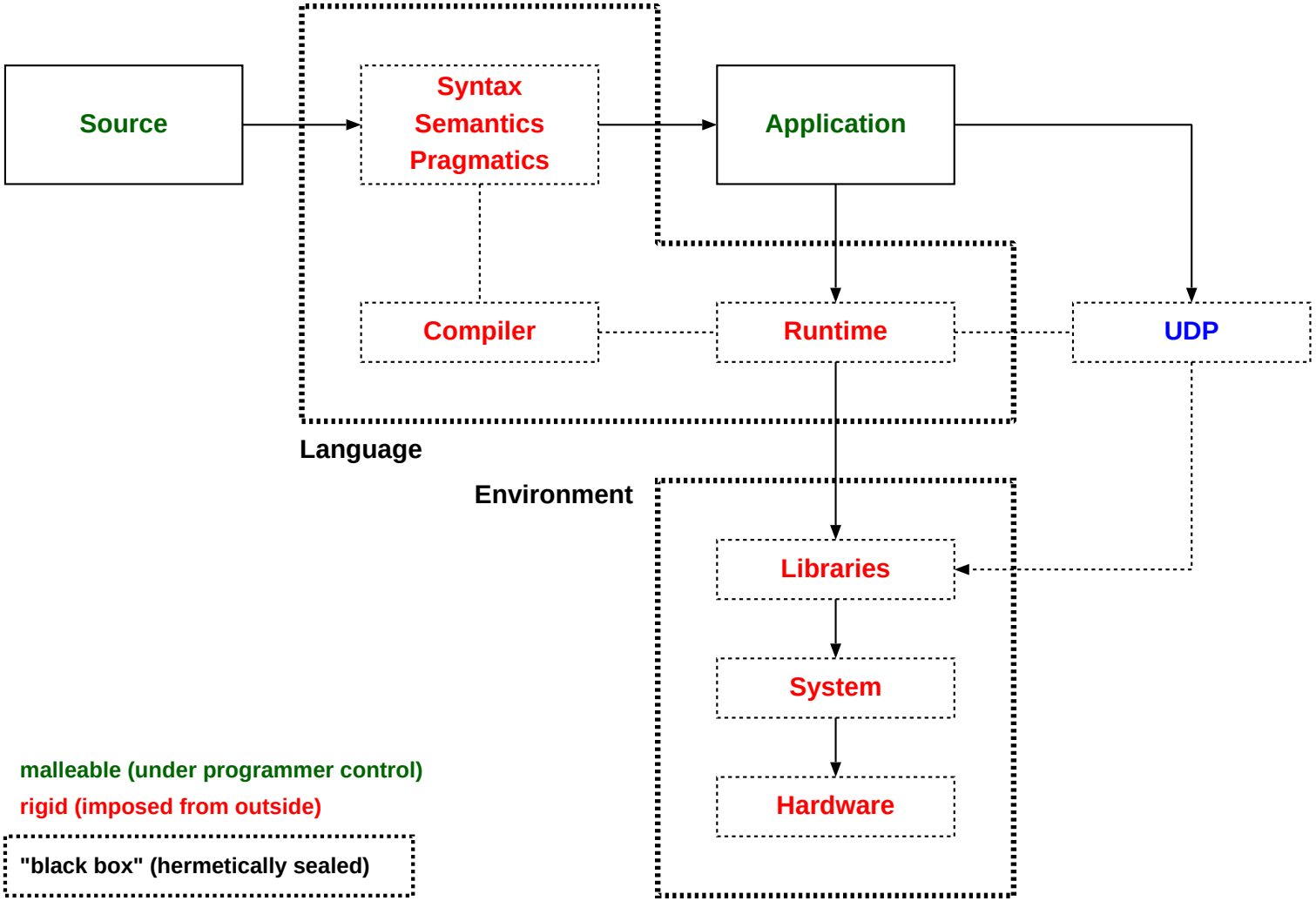
follow Einstein:

> *It is the grand object of all theory to make these irreducible elements as simple and as few in number as possible, without having to renounce the adequate representation of any empirical content whatever.*
>
> — Albert Einstein, *Mein Weltbild*, 1934

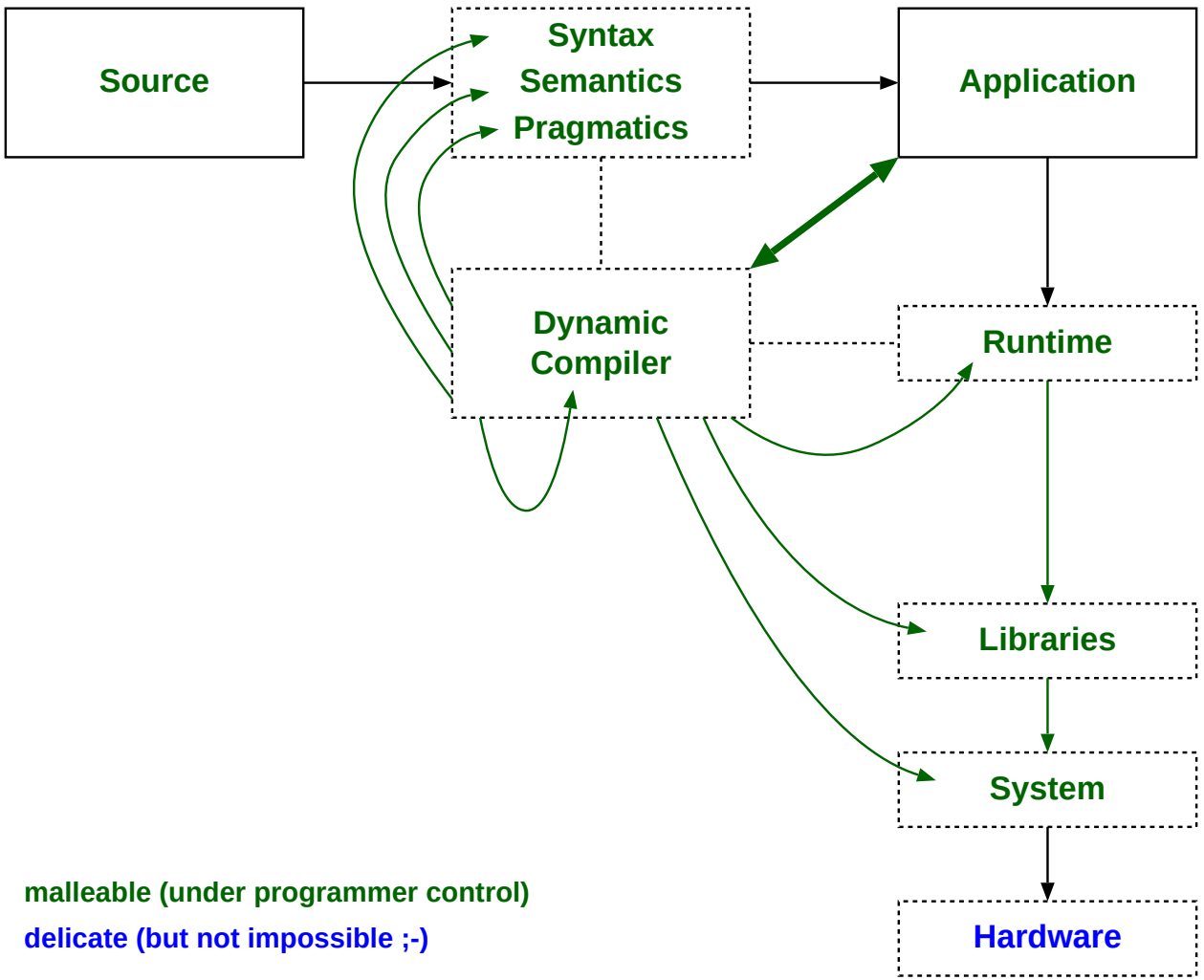
rather than trying to predict and provide everything the future might need

- provide the fewest, minimal abstractions that guarantee ...
- ... *any* new feature or paradigm can be added easily in the future
- extreme late binding: everything (to the metal) can be changed dynamically

conventional programming



unconventional programming languages



combined object-lambda architecture

proto-behaviour, with which to represent *function*

- low-level lisp-like s-expressions

proto-structure, with which to represent *form*

- objects manipulated through messaging

complementary

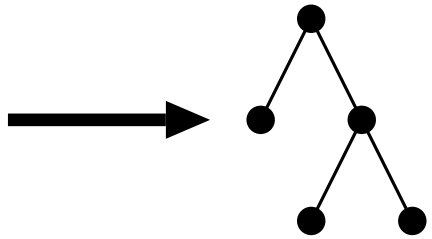
- form cannot be animated without function
- function cannot be represented without form

objects are form

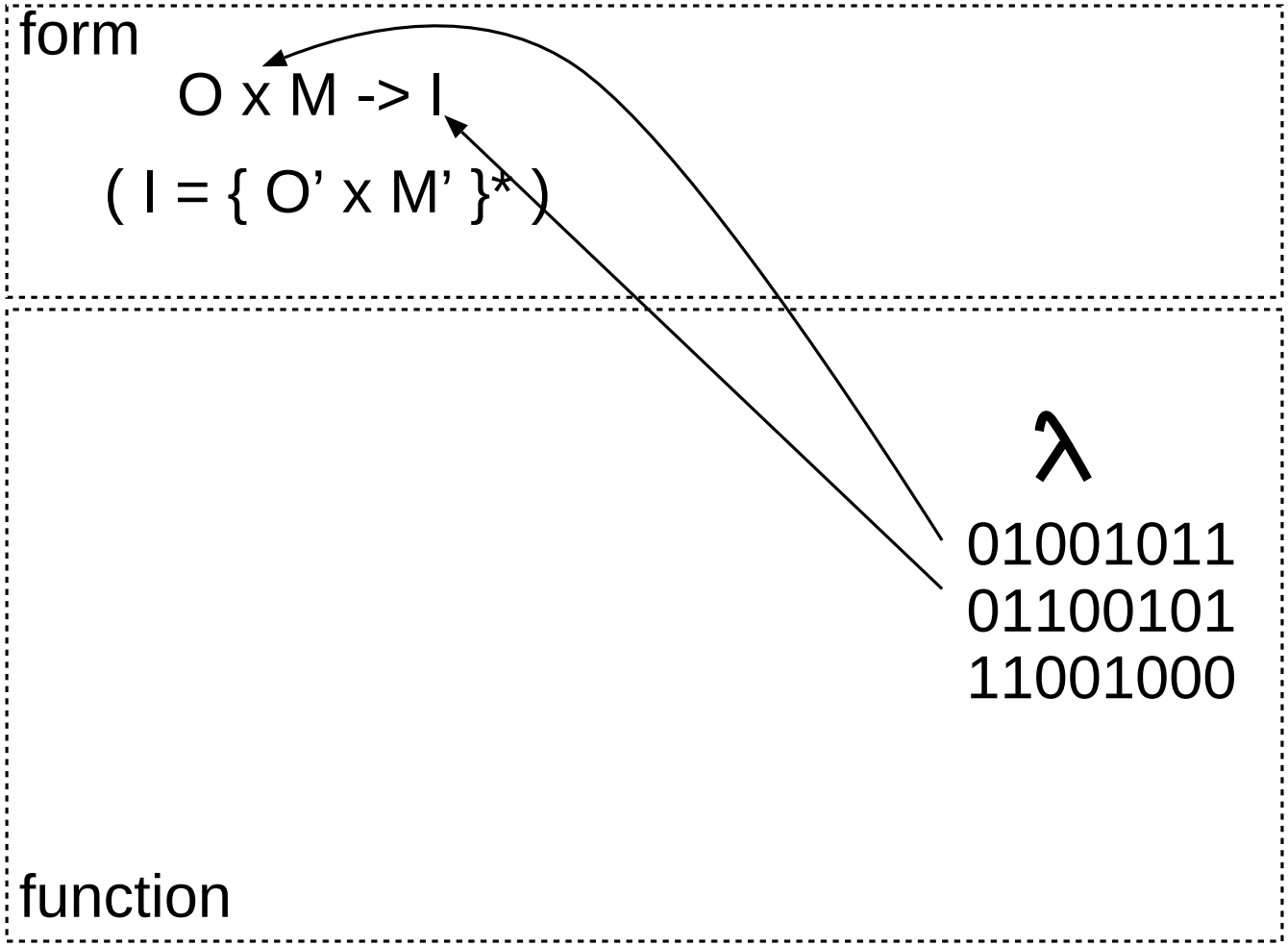
form

$$O \times M \rightarrow I$$

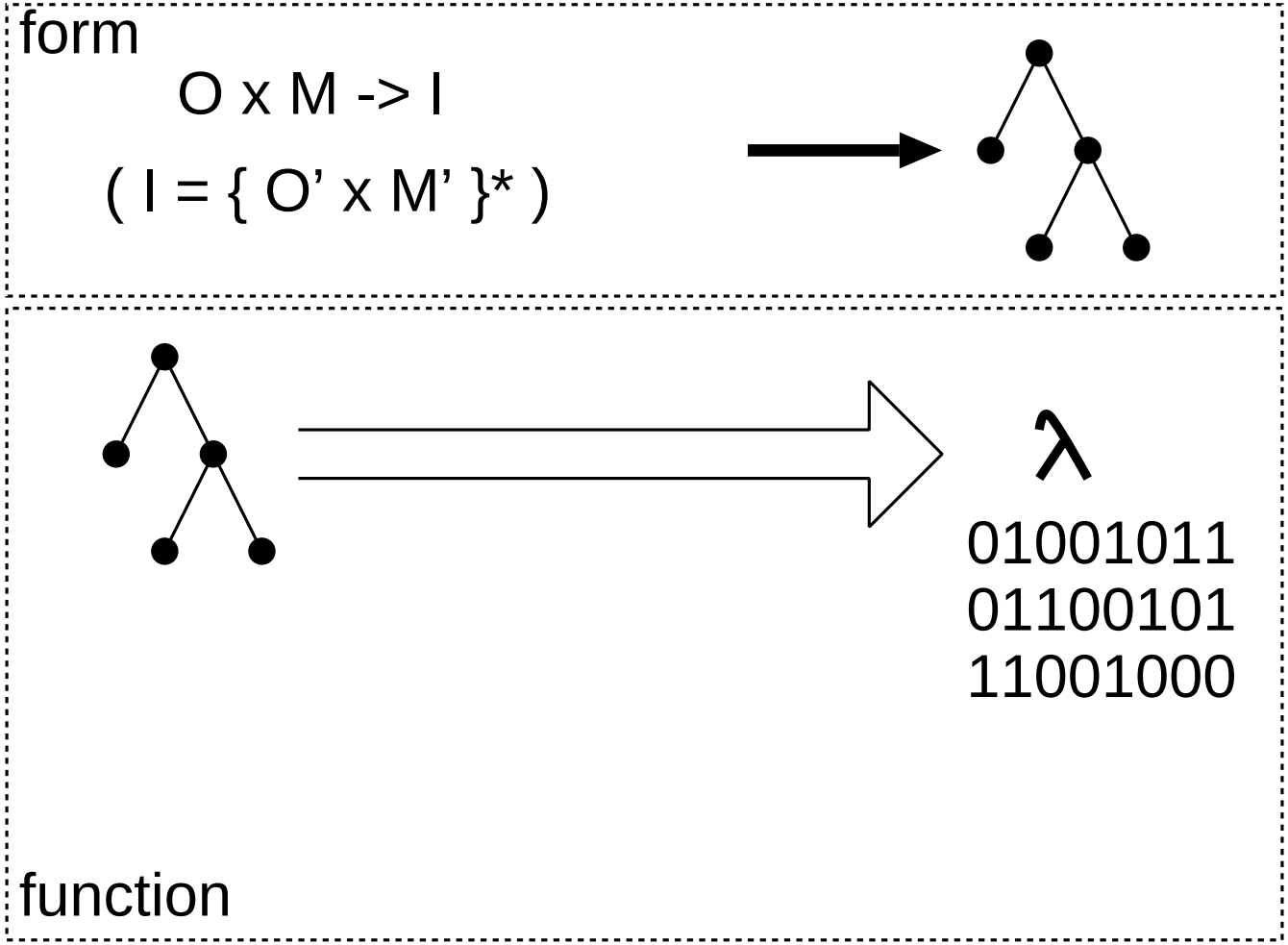
$$(I = \{ O' \times M' \}^*)$$



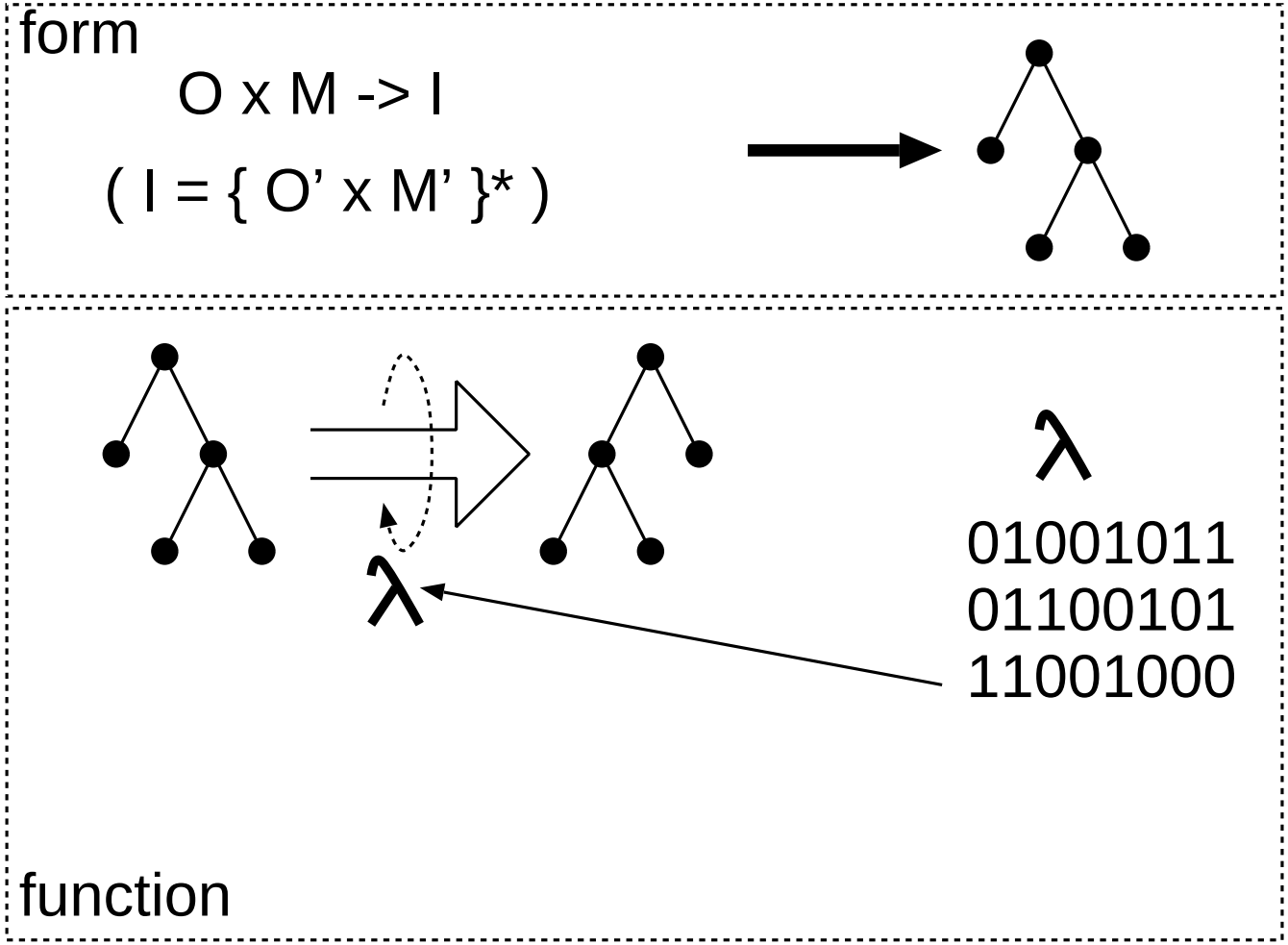
form needs function



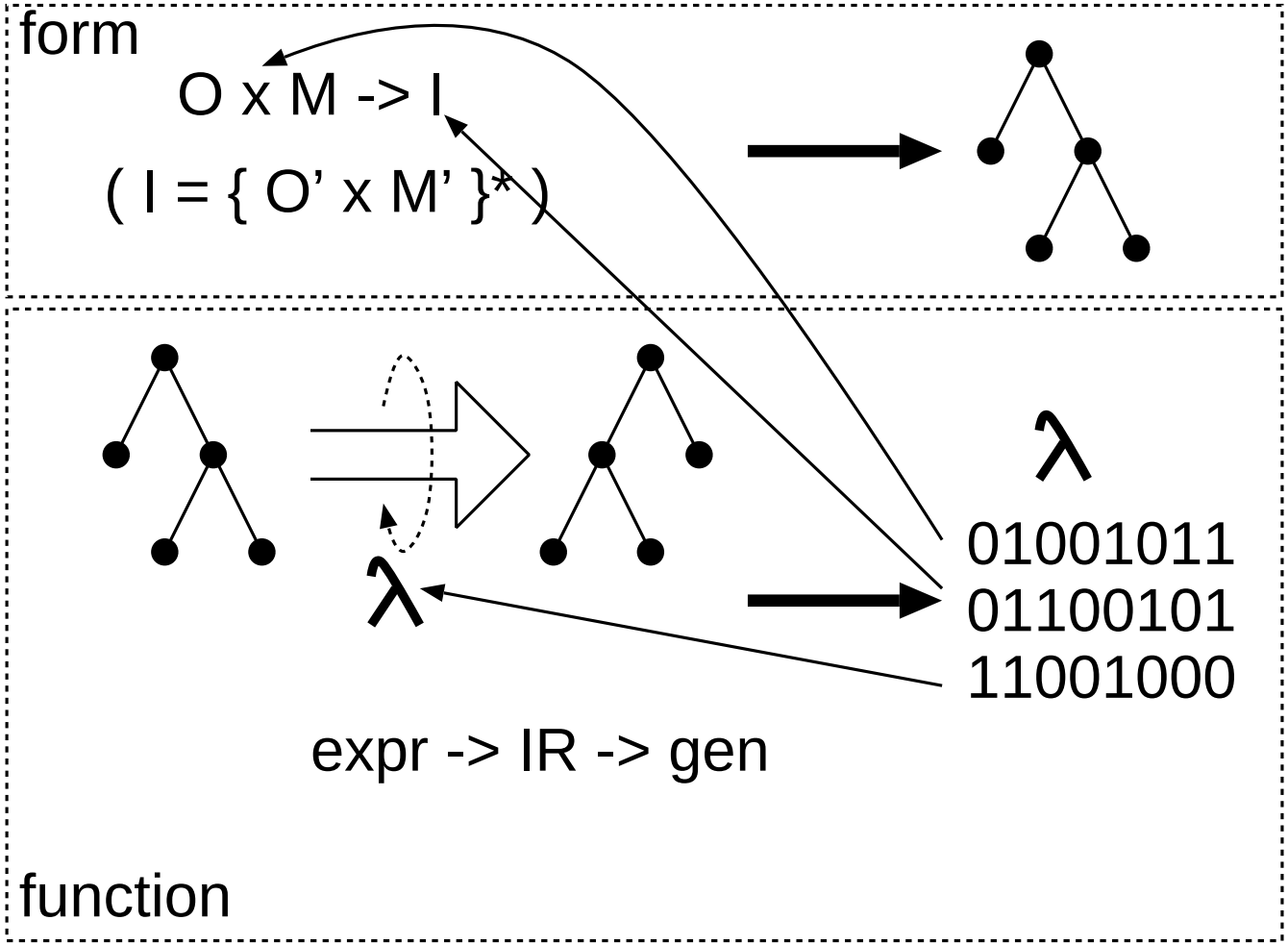
form describes function



functions transform form into function

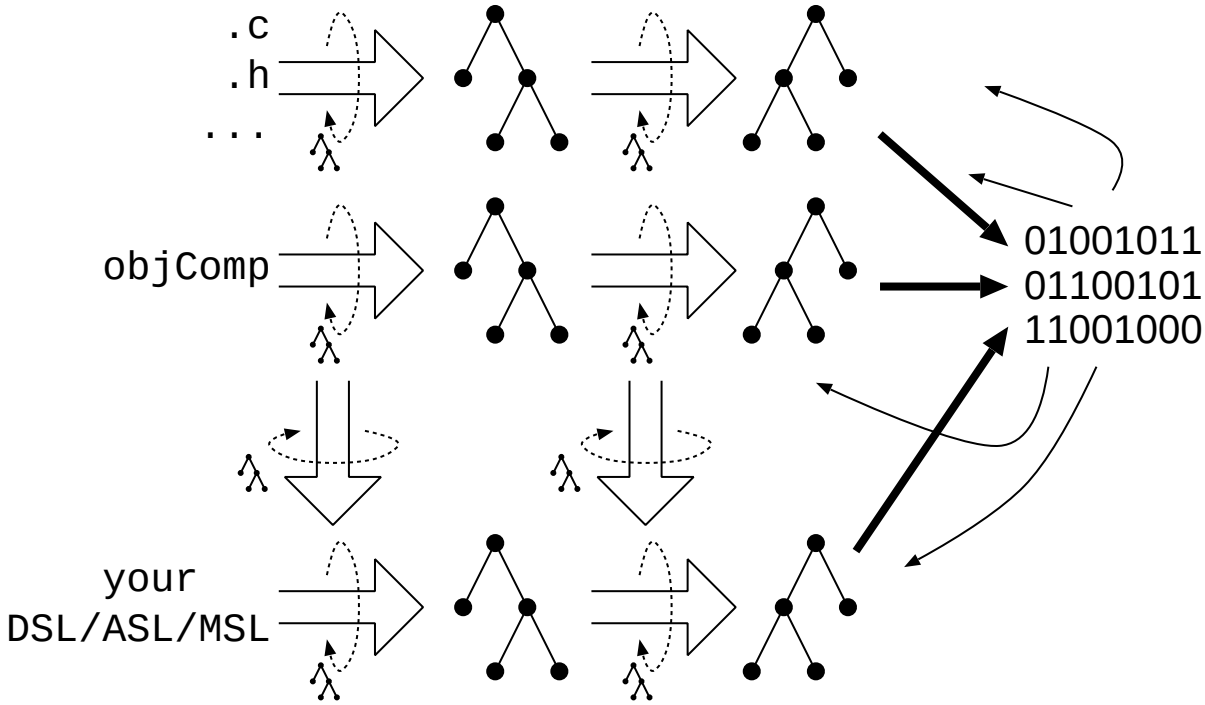


form describes function implements form



everything is self-describing structure

downwards, sideways



& upwards

- lexical, syntactic, semantic, IR analysis as pattern-directed transformation

pattern-directed transformation

```
grammar    =    - definition+
definition =    identifier:i EQUAL expression:e SEMICOLON?    -> [e named: i]
expression =    sequence:s ( BAR sequence:t -> [s or: t]:s )* -> s
sequence   =    prefix:p ( prefix:q -> [p and: q]:p )*        -> p

prefix     =    AND suffix:s -> [s peek]
           |    NOT suffix:s -> [s not]
           |        suffix

suffix     =    primary:p ( QUESTION -> [p zeroOne]:p
                          | STAR     -> [p zeroMore]:p
                          | PLUS     -> [p oneMore]:p
                          )?         -> p

primary    =    identifier:i !EQUAL     -> [i parseName]
           |    COLON identifier        -> [i parseStore]
           |    OPEN expression CLOSE
           |    literal
           |    class
           |    DOT                     -> [PeAny]
           |    ARROW cola-expression:e -> [e parseResult]

identifier =    ( [-a-zA-Z_][-a-zA-Z_0-9]* )$:i -> [i asSymbol]

literal    =    ( ( ['] ( !['] char )* $ ['] )
                | ( ["] ( !["] char )* $ ["] ) - ):t -> [t parse]

class      =    '[' ( !']' range )* $ ']' - :t -> [t asCharacterClass parse]

range      =    char '-' char | char

char       =    '\\' [abefnrtv'"\[\]\\]
           |    '\\' [0-3][0-7][0-7]
           |    '\\' [0-7][0-7]?
           |    !'\\' .
```

pattern-directed transformation

```
EQUAL       = '=' -
COLON       = ':' -
SEMICOLON   = ';' -
BAR         = '|' -
AND         = '&' -
NOT         = '!' -
QUESTION    = '?' -
STAR        = '*' -
PLUS        = '+' -
OPEN        = '(' -
CLOSE       = ')' -
DOT         = '.' -

-           = (space | comment)*
space       = ' ' | '\t' | end-of-line
comment     = '#' (!end-of-line .)*
end-of-line = '\r\n' | '\n' | '\r'
```

expressive intentions and meanings

parsing expressions

```
. 'string' [charset] #literal #(structure) @type
e? e* e+
&e !e
e1 e2
e1 | e2
```

formed into grammars

```
{
  grammarName :=
    rule1   = <p-expr>
    rule2   = <p-expr>
    <p-expr>
}
```

that create parsers

- final p-expr (if present) is start rule
- if not quoted, grammar evaluates to a parser that becomes active *immediately*

idiom:

```
{ ruleName } text to be parsed ...
```

or:

```
{ grammarName-ruleName } text to be parsed ...
```

expressive intentions and meanings

```
'{
  structure :=
    error       = ->[parser error: ['"syntax error near: " , [parser contents]]]
    eol         = '\r''\n'* | '\n''\r'*
    space       = [ \t]
    comment     = [-+] (!eol .)* eol
    ws          = (space | comment | eol)*
    _           = space*
    letter      = [a-zA-Z]
    digit       = [0-9]
    identifier  = < letter (letter | digit)* > _    -> [[parser text] asSymbol]
    number      = < digit+ > _                      -> [Integer fromString: [parser text] base: '10]
    columns     = '|'                               -> (structure-begin parser)
                   ( _  number->0                   -> [bitmap at: column put: (set bitpos [parser @ '0])]
                       (number->0)* '|'             -> (let () (set bitpos [parser @ '0])
                                                               (set column [[parser readPosition] - anchor]))
                    )+ eol ws                       -> [bitmap at: column put: (set width [bitpos + '1])]
    row         = ( number->0                       -> (set row [parser @ '0])
                    ) ? '|'                         -> (let () (set anchor [parser readPosition])
                                                               (set column '0))
                  _ ( identifier->0 '|'             -> (structure-field parser)
                      _ )+ eol ws                   -> (set row [row + width])
    name        = identifier->0 (!eol .)* eol       -> (structure-end [parser @ '0])
    diagram     = ws columns row+ name | error
}
```

expressive intentions and meanings

```
                                        { structure-diagram }
+-------------+----------+----------+-------------------+-------------------------------------------------+
| 00 01 02 03 | 04 05 06 | 07 08 09 | 10 11 12 13 14 15 | 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 |
+-------------+----------+----------+-------------------+-------------------------------------------------+
|                      sourcePort                       |                 destinationPort                 |
+-------------------------------------------------------+-------------------------------------------------+
|                                                 sequenceNumber                                            |
+-----------------------------------------------------------------------------------------------------------+
|                                              acknowledgementNumber                                        |
+-------------+----------+----------+-------------------+-------------------------------------------------+
|    offset   | reserved |   ecn    |    controlBits    |                    window                       |
+-------------+----------+----------+-------------------+-------------------------------------------------+
|                      checksum                         |                 urgentPointer                   |
+-------------------------------------------------------+-------------------------------------------------+
                      tcp -- Transmission Control Protocol packet header [RFC 793]
```

```
['{ svc     = &->(svc? [self peek])
    syn     = &->(syn? [self peek])     ->(out ack-syn)
    req     = &->(req? [self peek])     ->(out ack-psh-fin)
    ack     = &->(ack? [self peek])     ->(out ack)
            ;
    ( svc (syn | req | ack | .) | .     ->(out ack-rst)
    ) *
  } match: [NetworkPseudoInterface tunnel: '"/dev/tun0" from: '"10.0.0.1" to: '"10.0.0.2"]]
```

similarly via reduction and bottom-up rewrite all the way to the metal

bottom-up rewriting

```
return 3 + 4;
```

top-down transformation to

```
(RETI4
  (ADDI4
    (CNSTI4 3)
    (CNSTI4 4)))
```

bottom-up transformation driven by

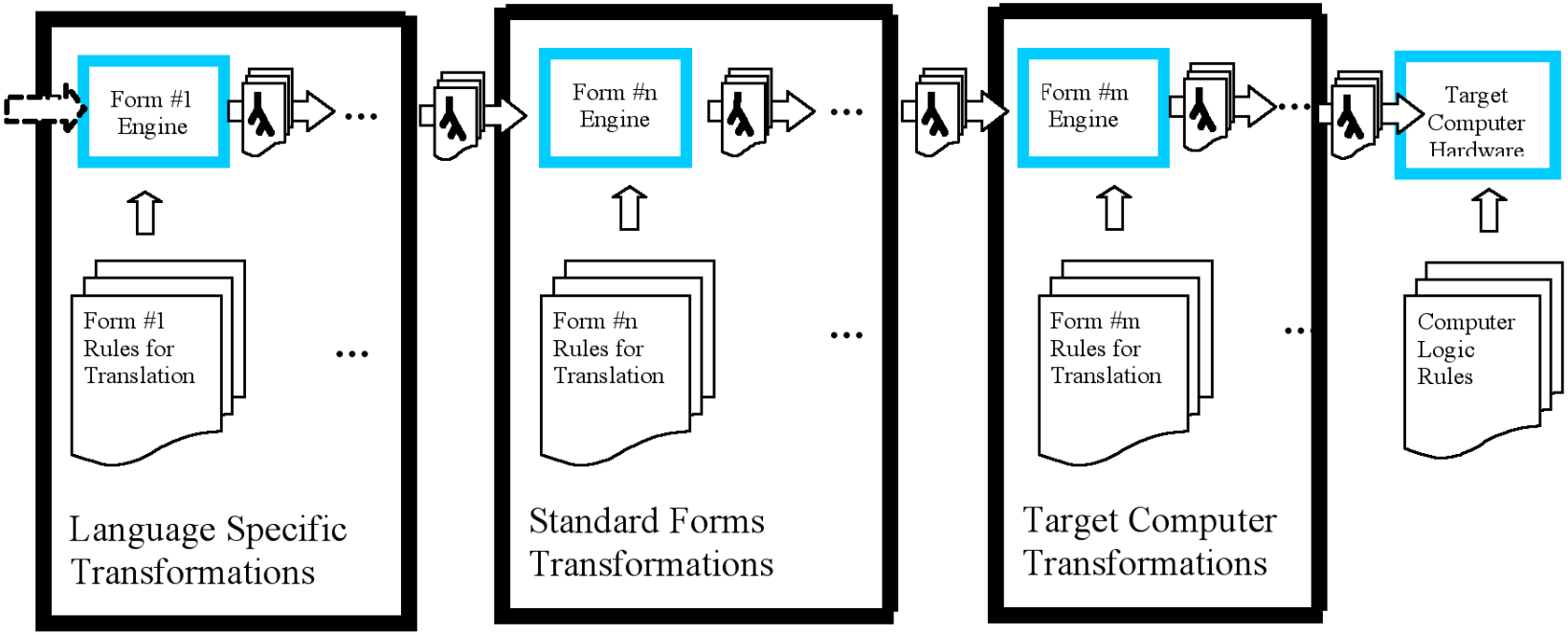
```
VOID = #(#reti4    REG:r      ):o -> ppc mr 3, r; ppc blr;

REG  = #(#cnsti2   .:i        ):o -> ppc addi o output, 0, i;
     | #(#addi4    REG:r REG:s):o -> ppc add  o output, r, s;
     | #(#addi4    I16:i REG:r):o -> ppc addi o output, r, i;
     | #(#addi4    REG:r I16:i):o -> ppc addi o output, r, i;
```

yielding

```
addi  3, 0, 3
addi  3, 3, 4
blr
```

cola pipeline



transformations at each stage

- first-class, dynamic
- optimise expressivity for each domain or activity

‘mathematical’ frameworks: relationships

$$p \Rightarrow q$$

‘mathematical’ frameworks: relationships about relationships

$$p \Rightarrow q$$

$$(p \Rightarrow q) \not\Rightarrow (q \Rightarrow p)^{\text{a}}$$

algebras of meaning

- systems that can reason about their own representation and implementation

[a] for a particular definition of $\Rightarrow$

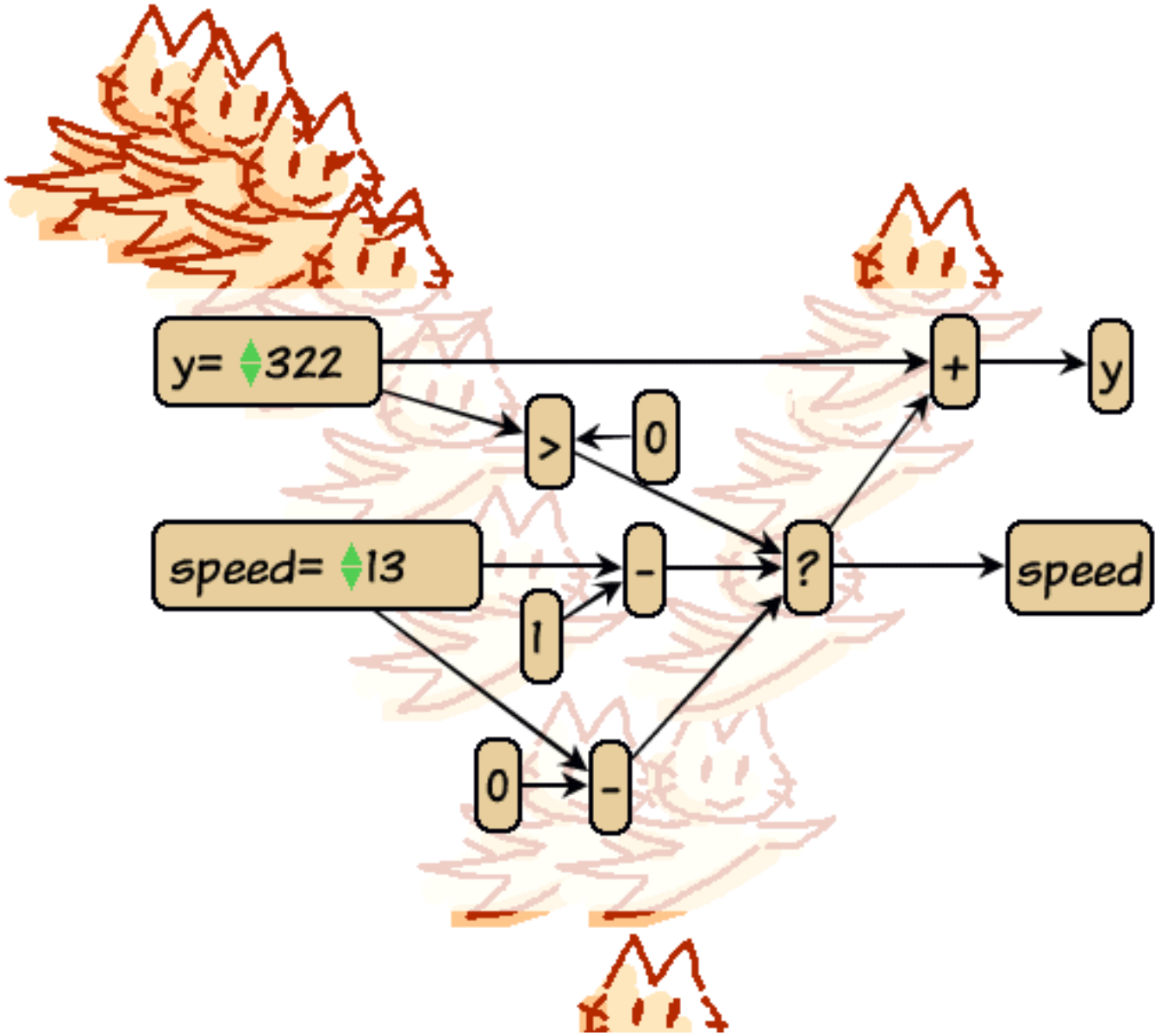
malleability facilitates experimentation

a few hundred lines is sufficient to build ‘the essence’ of new and existing languages and paradigms

- Javascript
- Prolog
- CodeWorks
- etc...

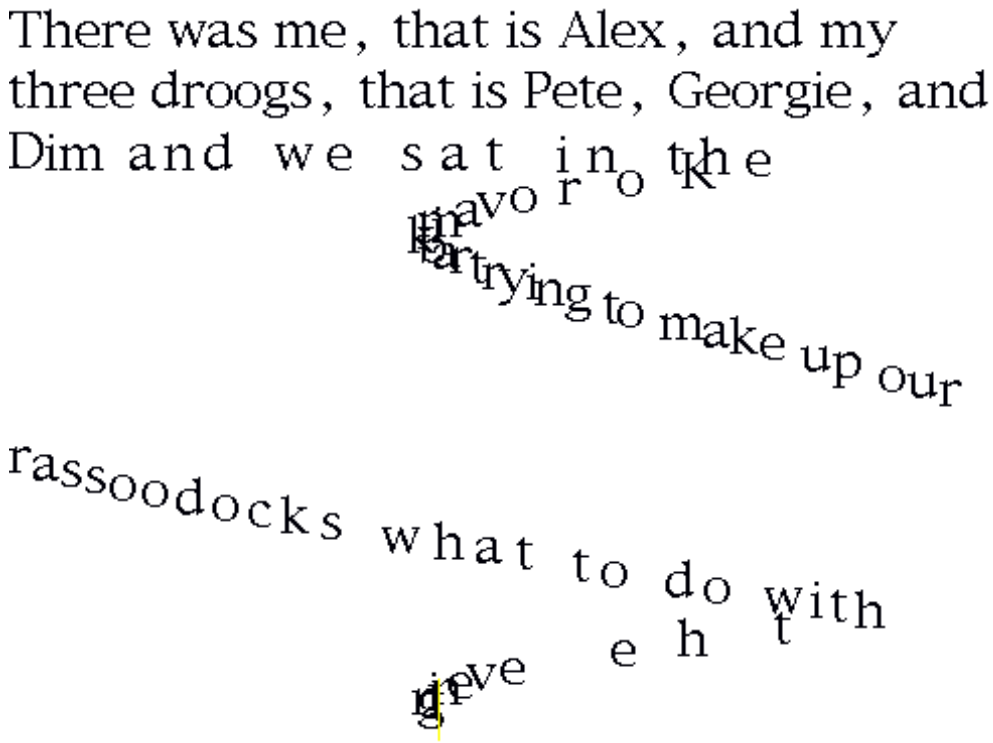
plus...

dataflow



particles and fields

self-organising systems with strictly local knowledge and control



to see farther than others, *stand on the shoulders of giants*

"Maxwell's" equations?!

electric charges produce electric fields

Gauss's law

$$\nabla \cdot \mathbf{E} = \frac{\rho}{\epsilon_0}$$

magnetic monopoles do not exist

Gauss's law for magnetism

$$\nabla \cdot \mathbf{B} = 0$$

changing magnetic fields produce electric fields

Faraday's law of induction

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

changing electric currents and fields produce magnetic fields

Ampère's circuital law, with Maxwell's correction

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} + \mu_0 \epsilon_0 \frac{\partial \mathbf{E}}{\partial t}$$

[special relativity, ...]

some of our giants

in the ‘engine room’...

- self-describing parsers: Val D. Schorre (Meta-II); D. I. Andrews and J. F. Rulifson (Tree Meta)
- parametric grammar rules: Larry Tesler et al (Lisp70)
- ordered choice: Bryan Ford (Parsing Expression Grammars)
- EBNF syntax: W3C (XML standard)
- code generation: Aho et al (bottom-up rewrite systems)
- IR forms: Fraser, Hanson (lcc, iBURG)
- proto-behaviour: McCarthy (recursive functions)
- proto-structure: Kay, Leiberman, etc. (recursive objects)

and many, many more in the ‘miracle’ and end-user parts

from the 1950s to the present day

adopt and amplify; avoid ‘NIH’

- sometimes progress is *behind you*

fresh perspectives from a mathematical point of view

once upon a time...

- memory was fast
- processor was slow
- processor speed limited performance

and then...

- processors got a lot faster
- memory got a bit faster
- memory bandwith limits performance

processor-intensive solutions are much more viable than they used to be

- forget ad-hoc approximations
- do ‘the math’ for real

gezira

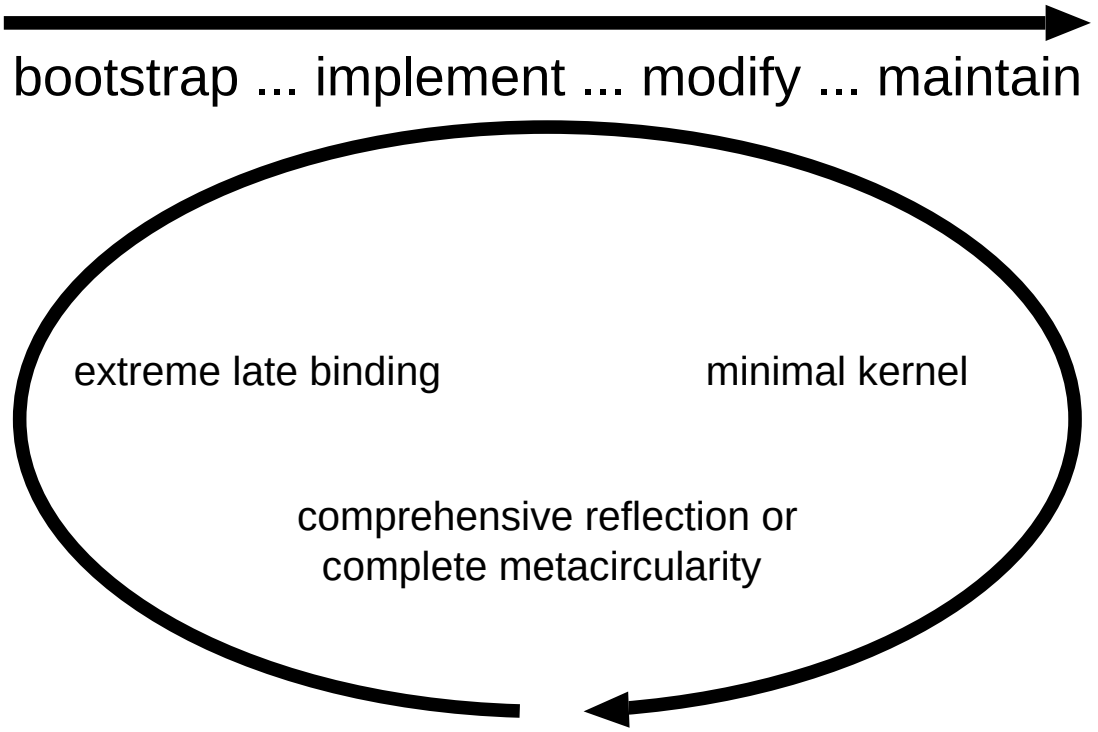
contibution of edge $\overrightarrow{AB}_i$ to pixel at (x, y) is

$$min(|\sum coverage(\overrightarrow{AB}_i)|, 1)$$

where

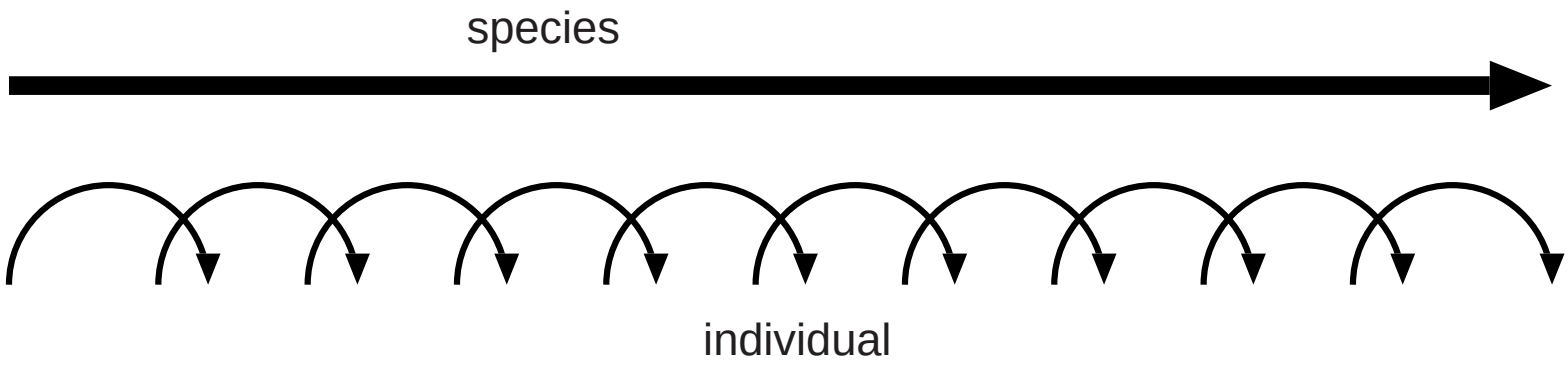
$$\begin{aligned}
\sigma(P, Q) &= (Q_y - P_y)(x + 1 - \frac{Q_x + P_x}{2}) \\
\gamma(P) &= min(x + 1, max(x, P_x)), \\
&\quad min(y + 1, max(y, P_y)) \\
\omega(P) &= \frac{1}{m}(\gamma(P)_y - P_y) + P_x, \\
&\quad m(\gamma(P)_x - P_x) + P_y \\
coverage(\overrightarrow{AB}) &= \sigma(\gamma(A), \gamma(\omega(A))) + \\
&\quad \sigma(\gamma(\omega(A)), \gamma(\omega(B))) + \\
&\quad \sigma(\gamma(\omega(B)), \gamma(B))
\end{aligned}$$

self-sustaining languages



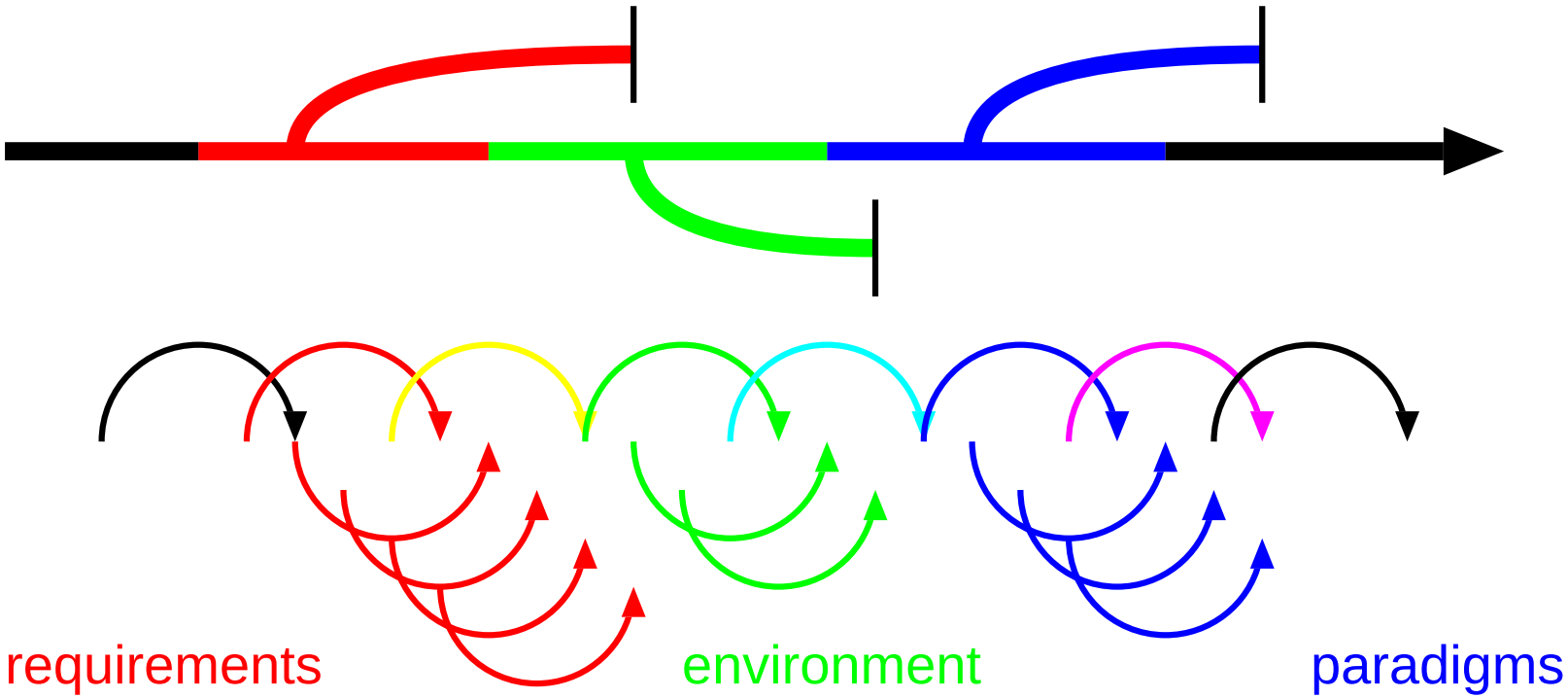
- self-bootstrapping, -implementing, -modifying and -maintaining
- small number of powerful abstractions
- engines of their own replacement
- laboratories for their own future(s)

self-sustaining systems

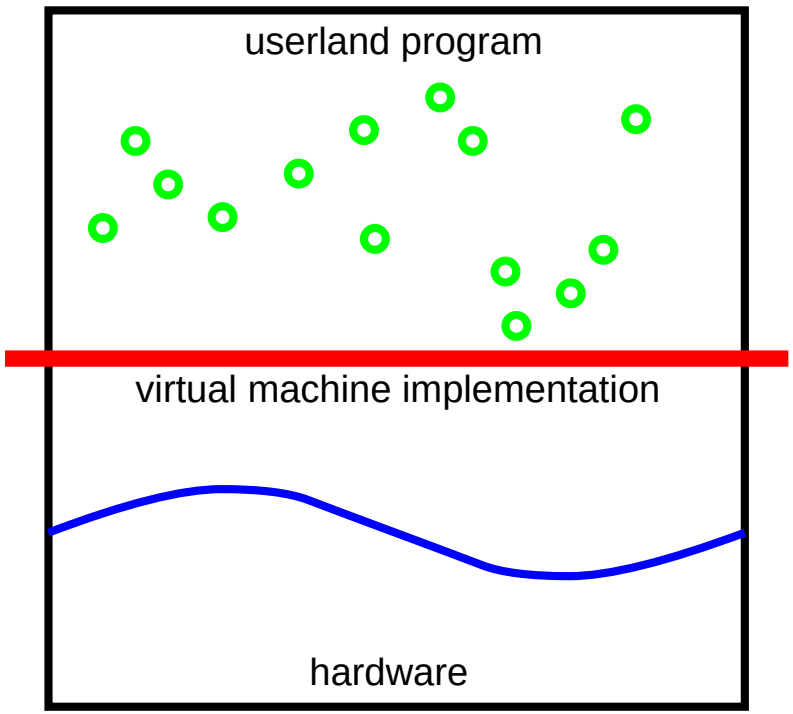


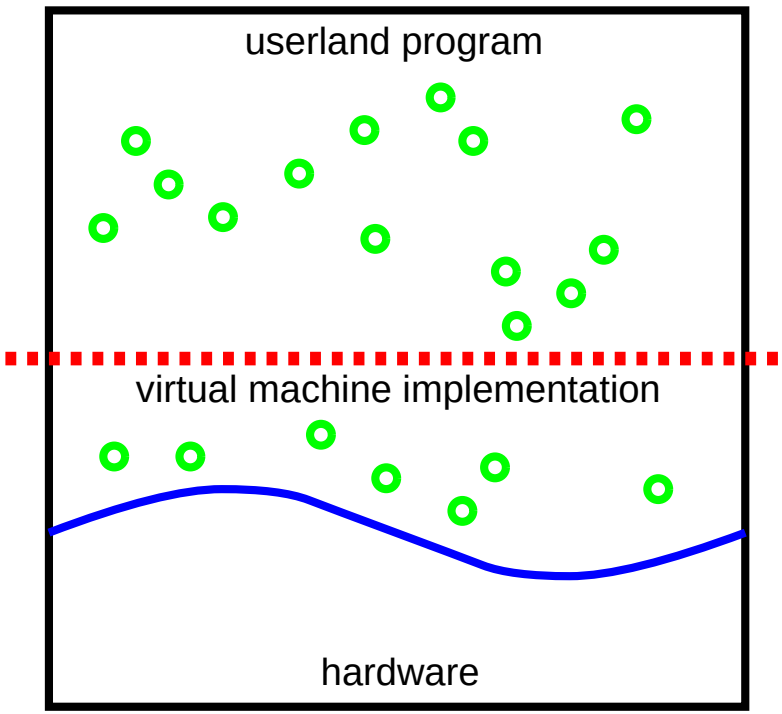
- more than the individuals in which it is expressed
- must respond to internal and external pressure by evolving
- depends on local death for global survival

self-sustaining systems



relevance to VM-based languages





information

Viewpoints Research Institute

`http://vpri.org`

STEPS project

`http://vpri.org/work/ifnct.html`

- 2007 end-of-year report

software download

`http://piumarta.com/software/cola`

- colas-whitepaper

would like to thank

NSF

intel

NOKIA
Connecting People

SAP

for generously sponsoring our work