SML, Menlo Park, 2005 – 11 – 09

Making COLAs with Pepsi and Coke

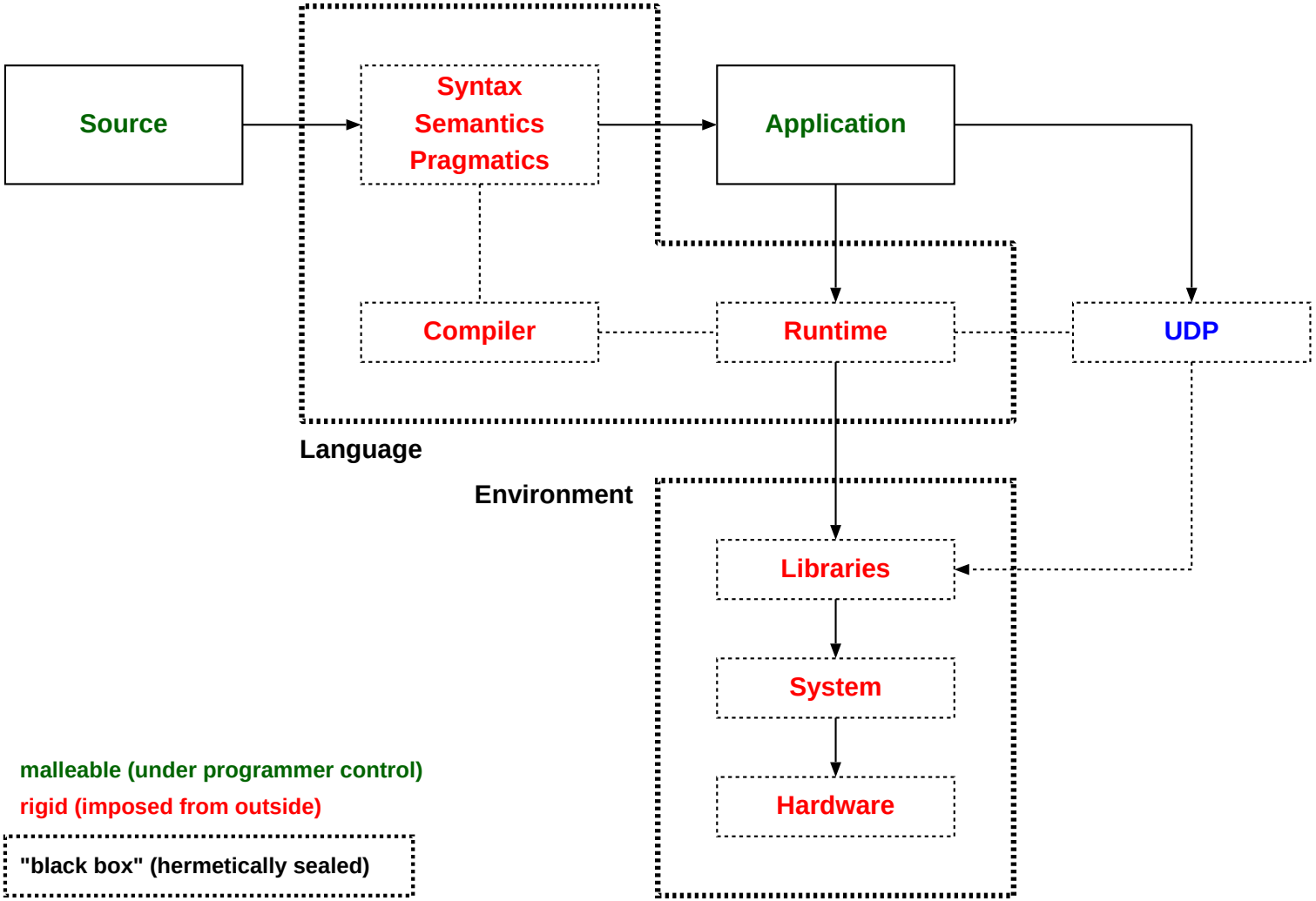
implementing open, dynamic programming systems

Ian Piumarta

HPL

piumarta@gmail.com

conventional programming



‘typical’ programming languages

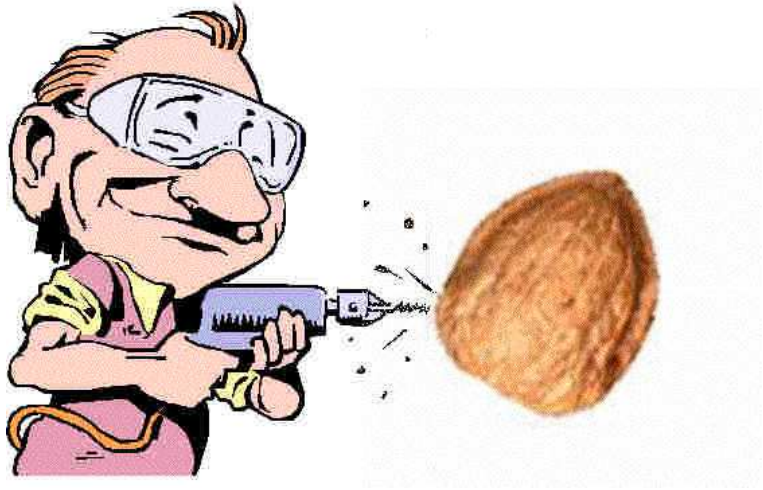
designed elsewhere, sealed hermetically,

lots of interesting stuff inaccessible

and ‘typical’ language hackers

read: *imaginative programmers* in search of *better paradigms* for *creative expression*
(or maybe just some way to get necessary things done)

armed with power tools:
`emacs`, `cc`, `as`, `ld`,
`kill`, `exec`, `gdb`
(rinse and repeat)



do you have:

- an offline compiler?
- a virtual machine or interpreter?
- a precompiled runtime library?

$\not\Rightarrow$

then you suffer in some degree from:

- early binding assumptions & rigidity
- closed implementation & future
- artificial barriers & complexities

goals

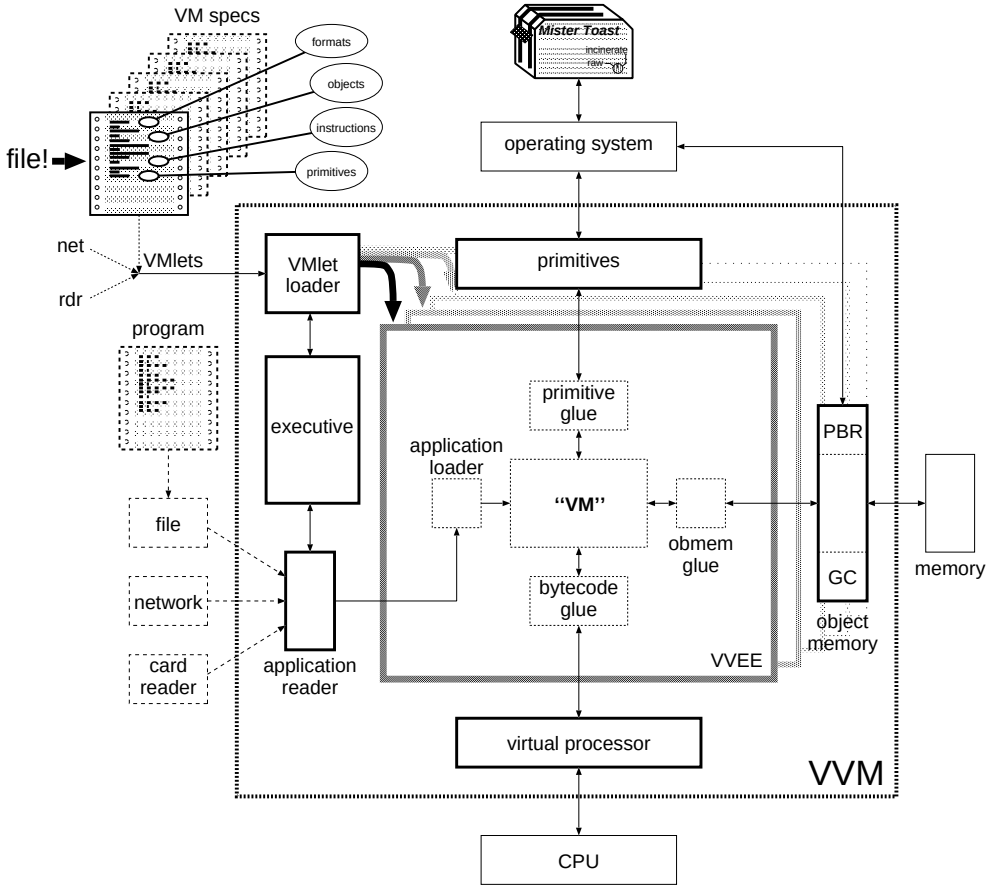
understand/modify any part of the system

- organisation encourages modification

user-centred at all ‘levels’

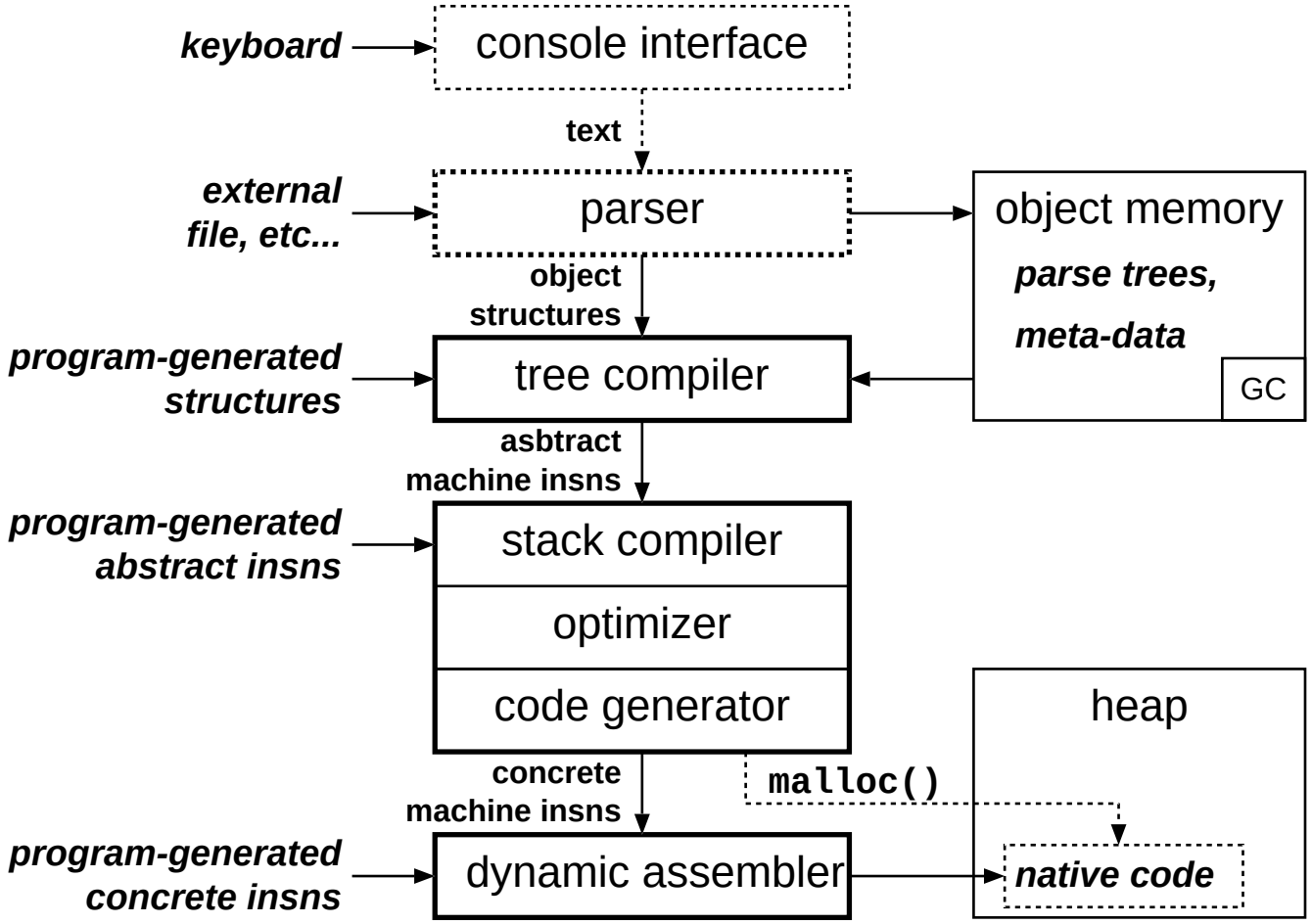
- openness
  - components are accessible and modifiable
- simplicity
  - single representation and paradigm
  - self-hosting, from input to codegen & primitives
- evolutionary programming
  - pervasive late-binding

previous prototypes

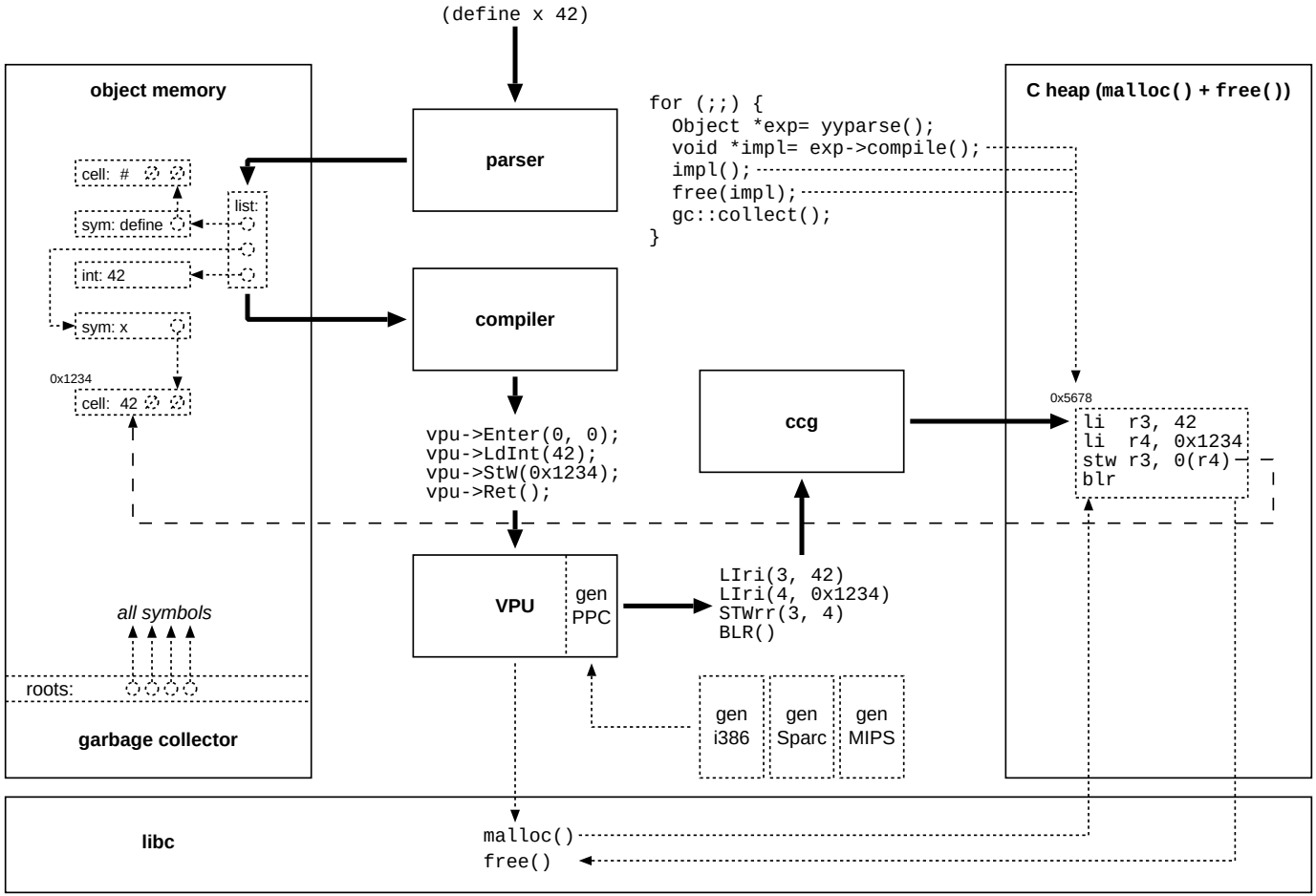


in particular: *virtual processor*, *primitives*, *object memory*

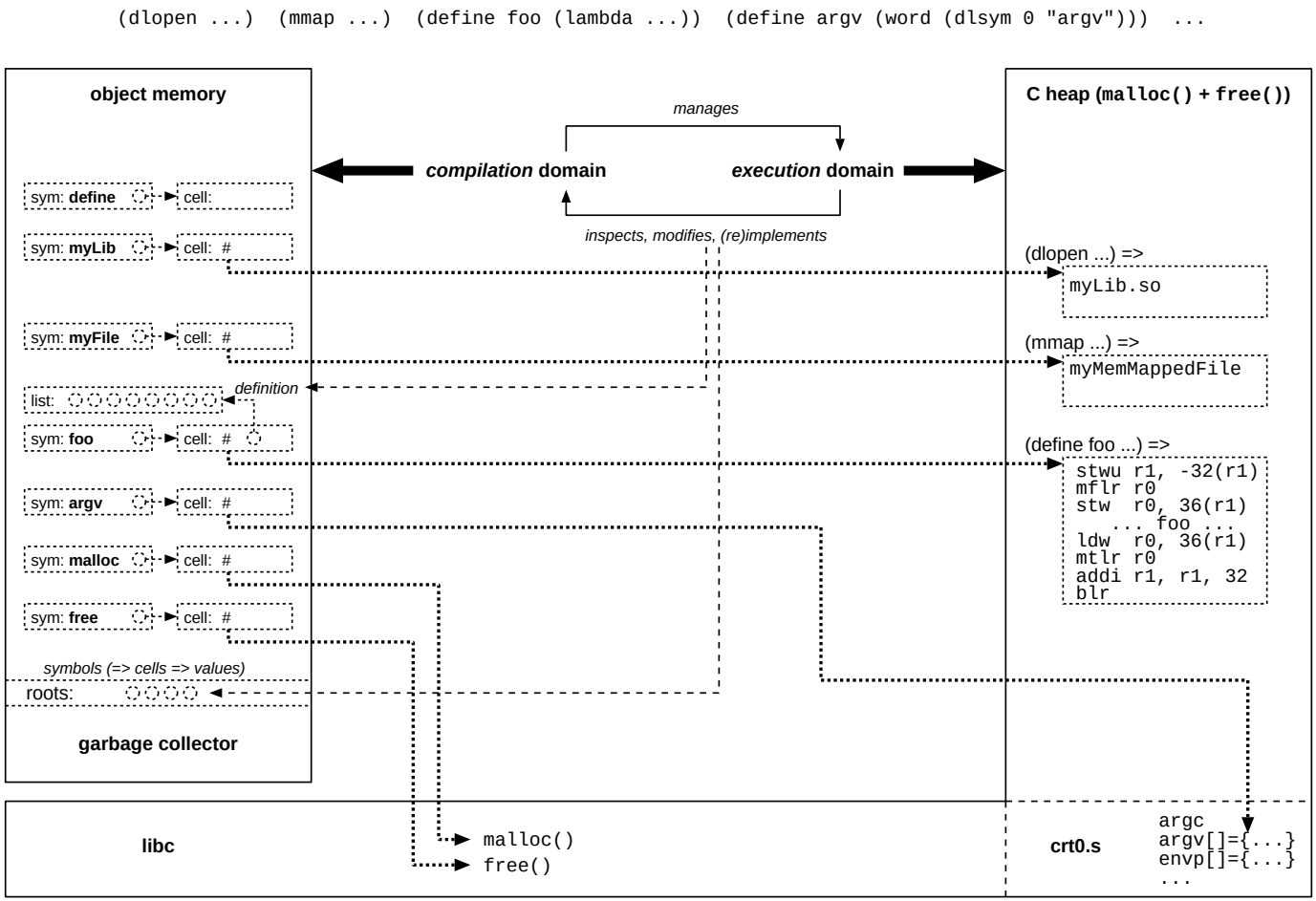
runtime architecture (management version)



runtime architecture (engineering version)

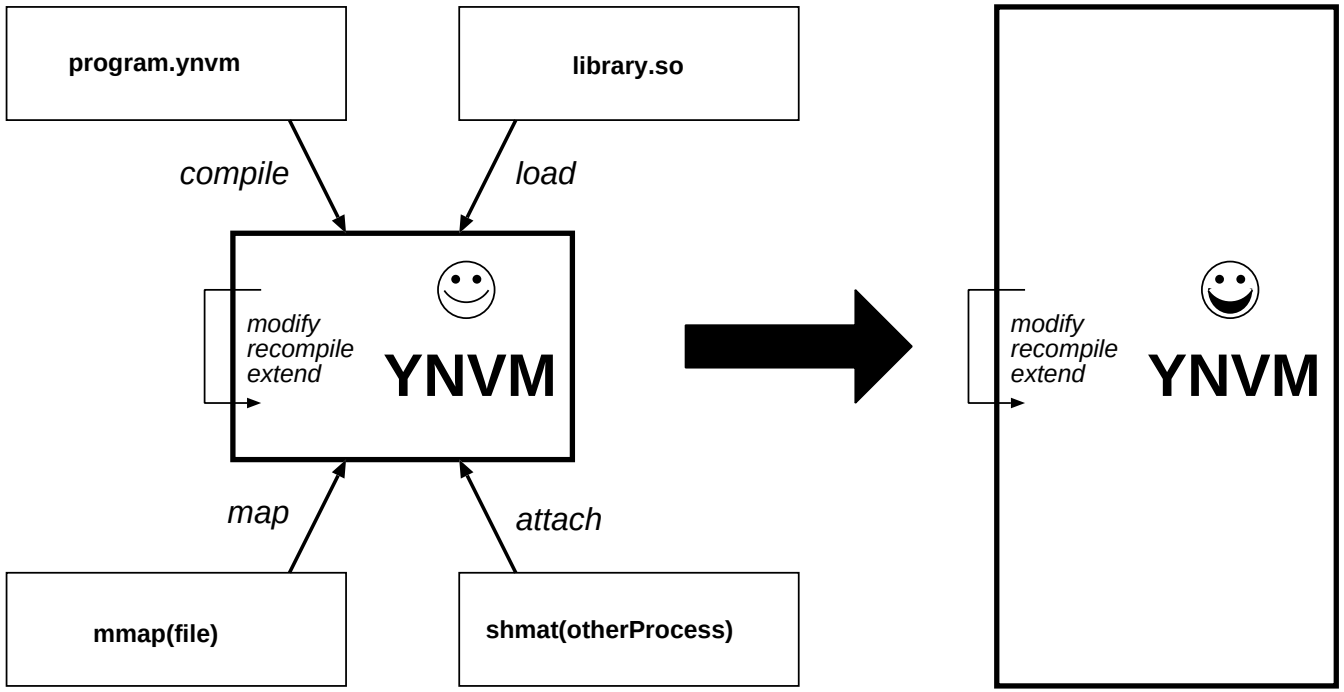


runtime architecture (sales version)

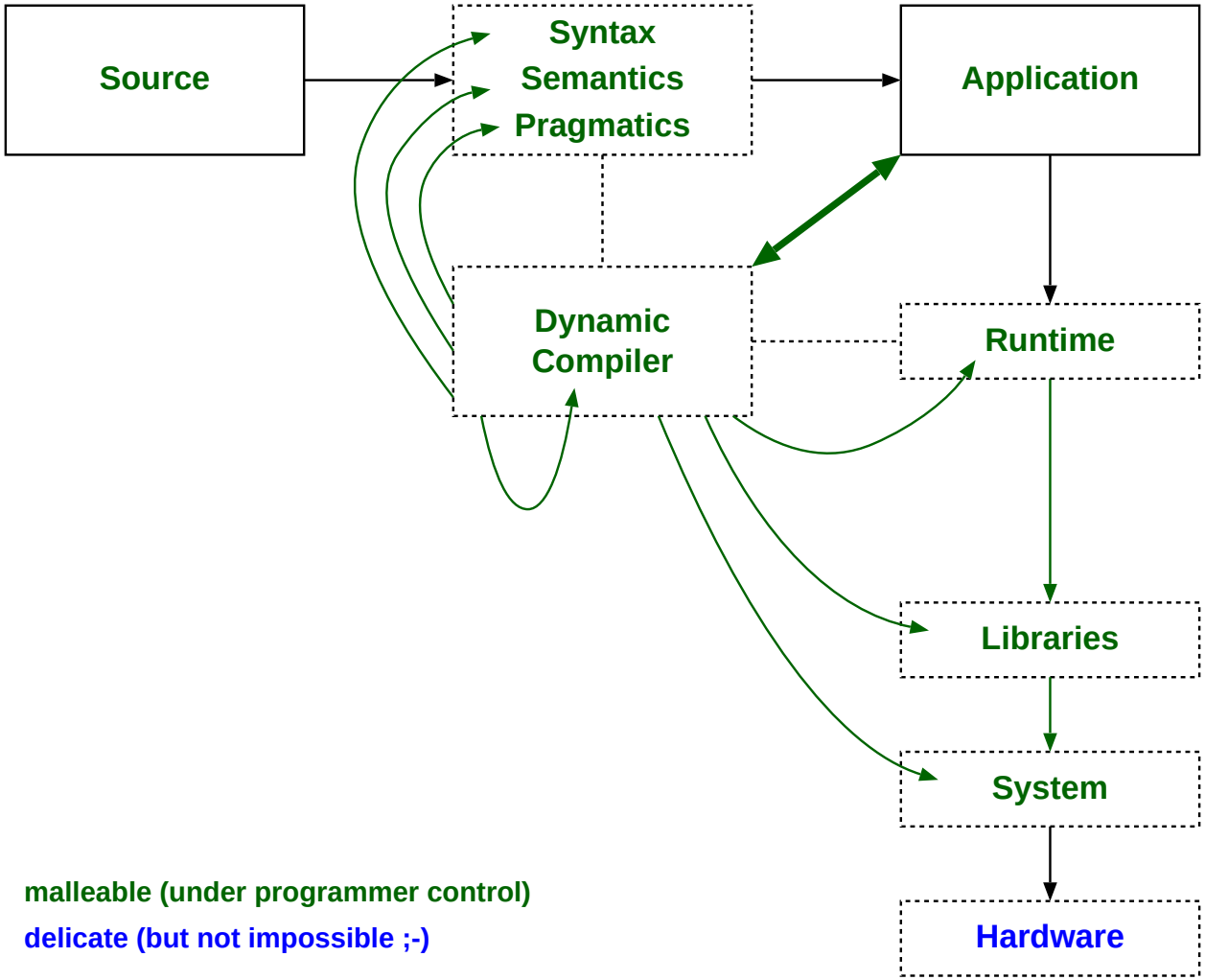


...

runtime architecture (marketing version)



unconventional programming



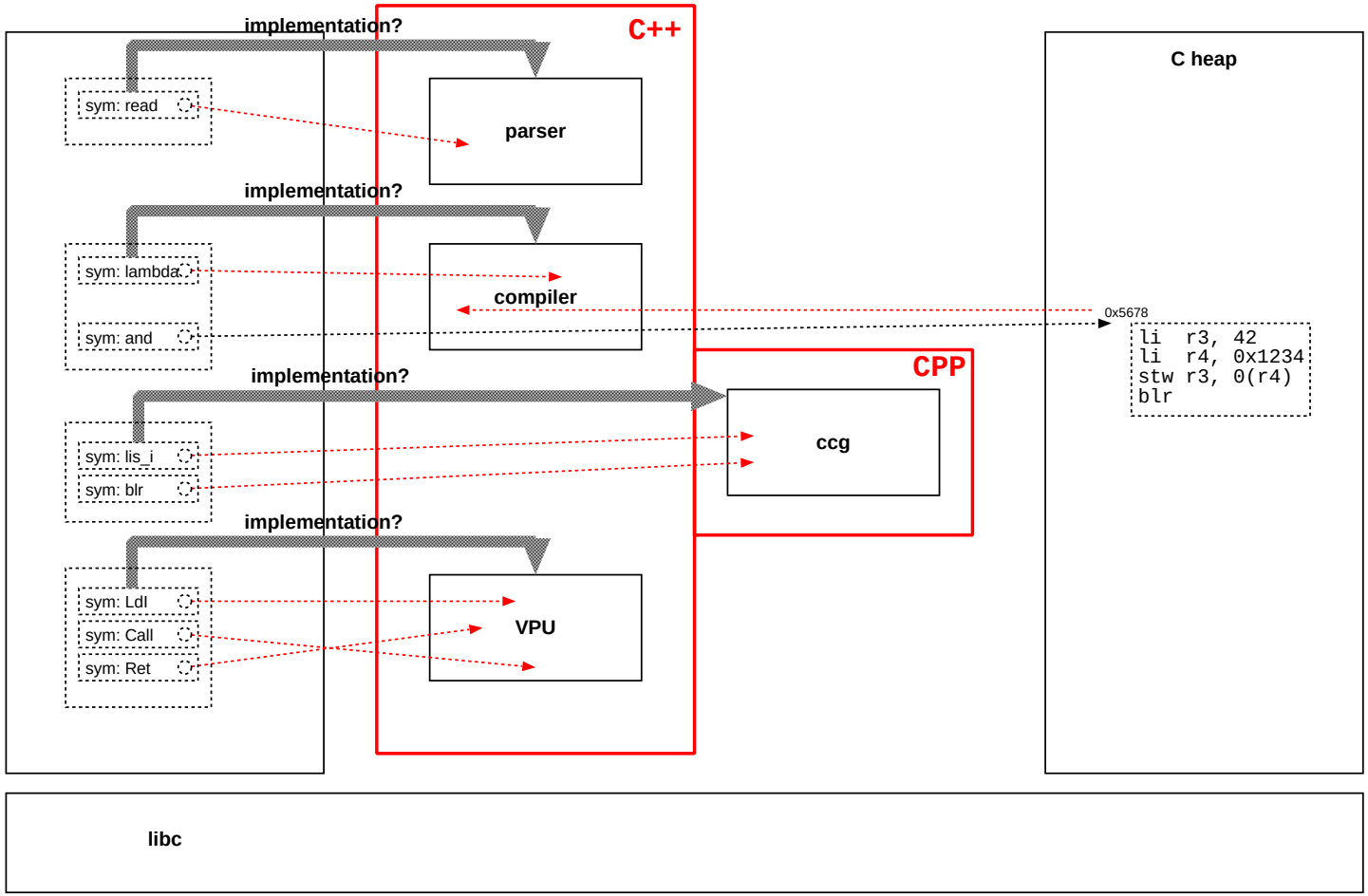
malleable (under programmer control)

delicate (but not impossible ;-)

‘atypical’ programming language

$\Rightarrow$

runtime architecture (R&D version)



...

C++

random observations

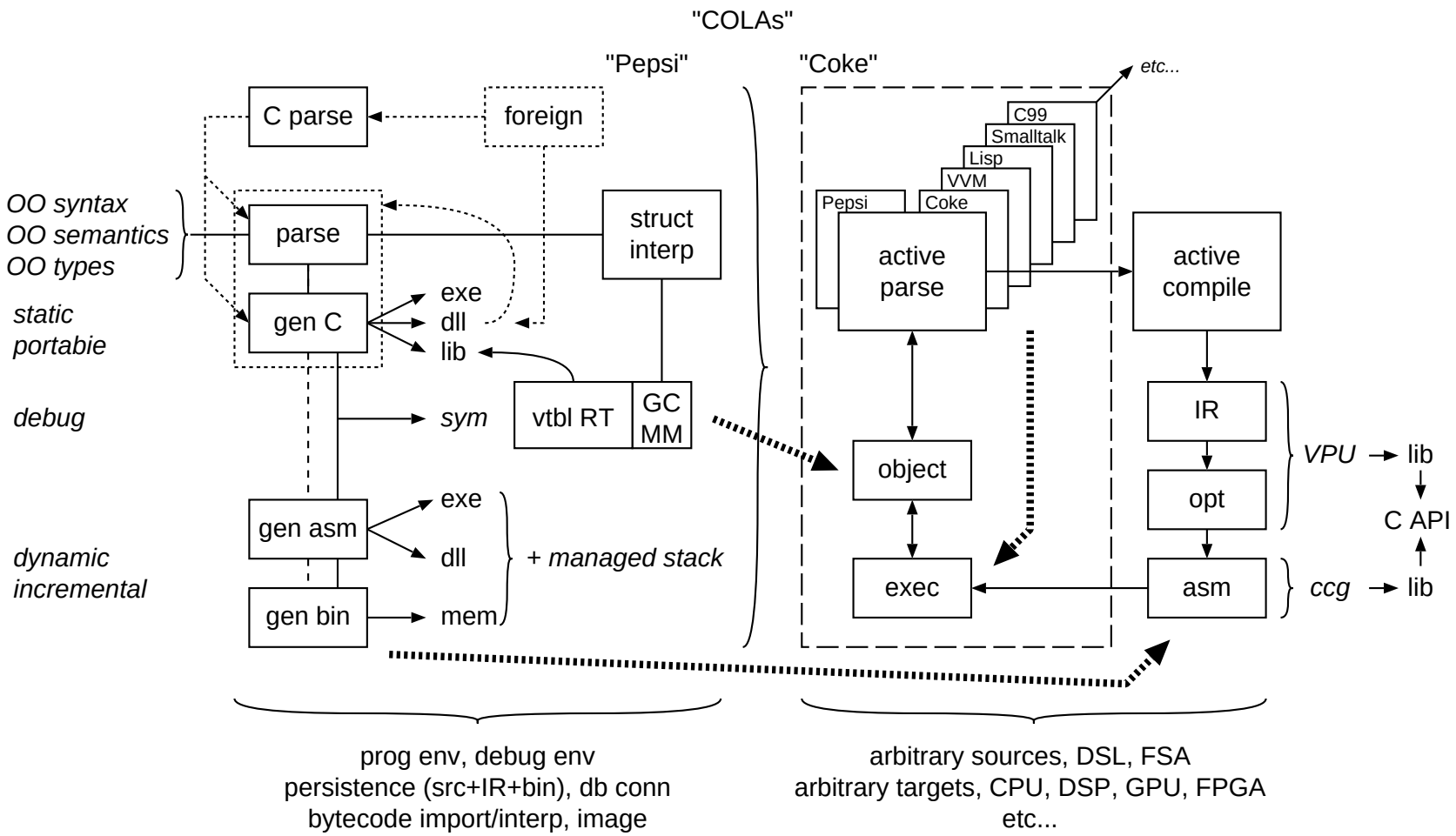
- no runtime ‘meta’ data: reflection difficult, intercession almost impossible
- static compilation only: dynamic modification of behaviour proscribed
- vtable layouts calculated statically: no modification at runtime
- ABI arbitrary and concealed: difficult to target reliably
- intrinsic types are not objects: no reuse in dynamic contexts
- intrinsic operators are not messages: no specialisation of primitive behaviour
- virtual functions useless: *a priori* knowledge (declarations) required
- complications loading shared objects into other languages

a better kernel language

prototypes + messages + delegation (shared behaviours + state)

- pure objects and messages (nothing else)
- clone families: somewhere between lightweight classes and instance-specific behaviour
- implementation (language) model = (precisely) implemented (language) model
- everything can be modified, dynamically
- free, unrestricted mix of static and dynamic language
- on-the-fly modification of kernel behaviour

COLA

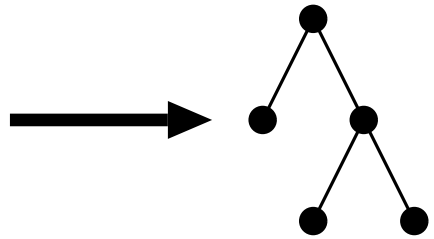


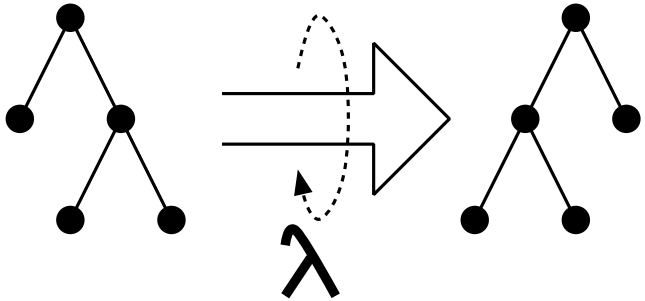
structure creates meaning

pepsi

O x M -> I

(I = { O’ x M’ }*)





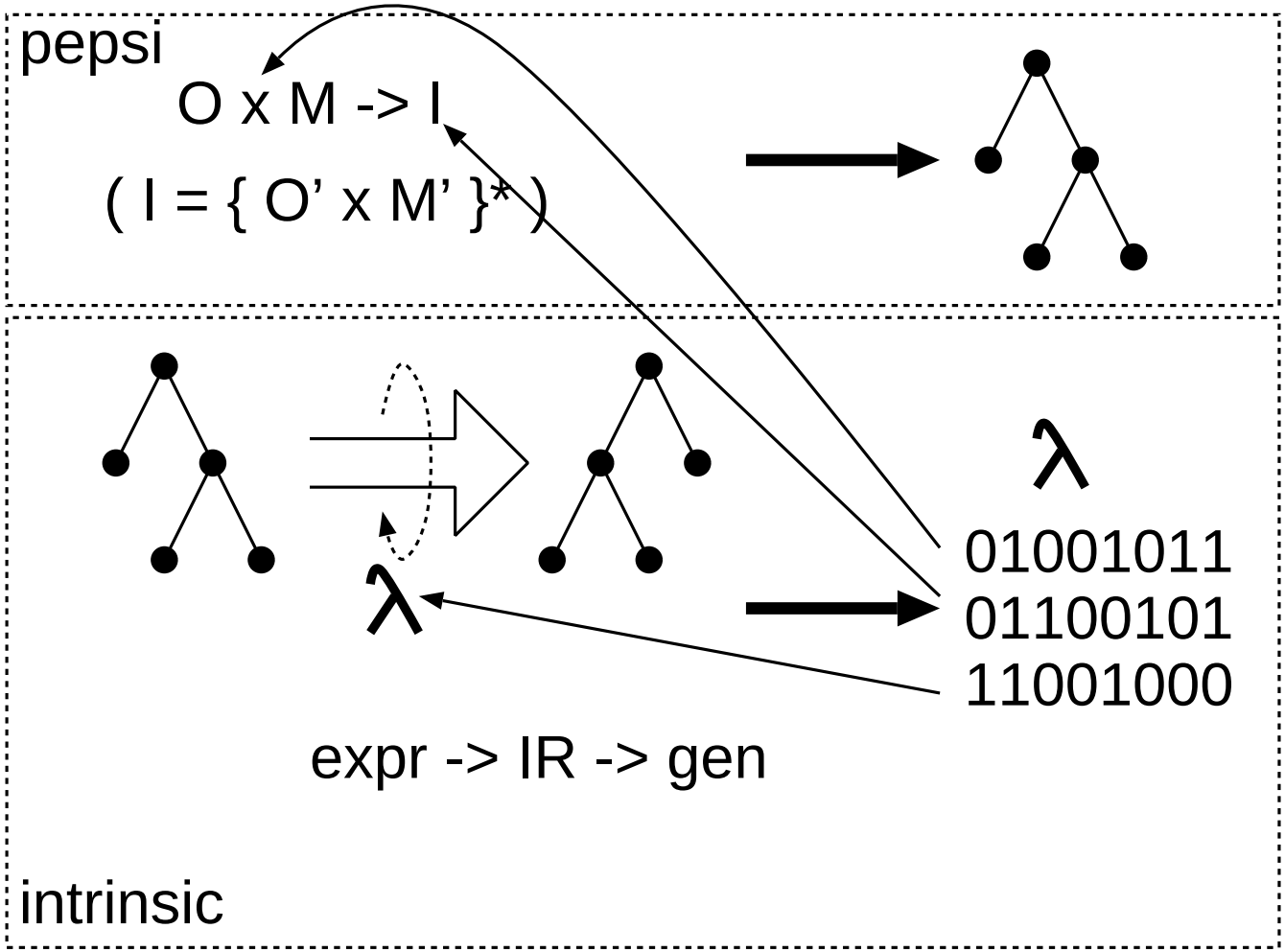
λ

01001011
01100101
11001000

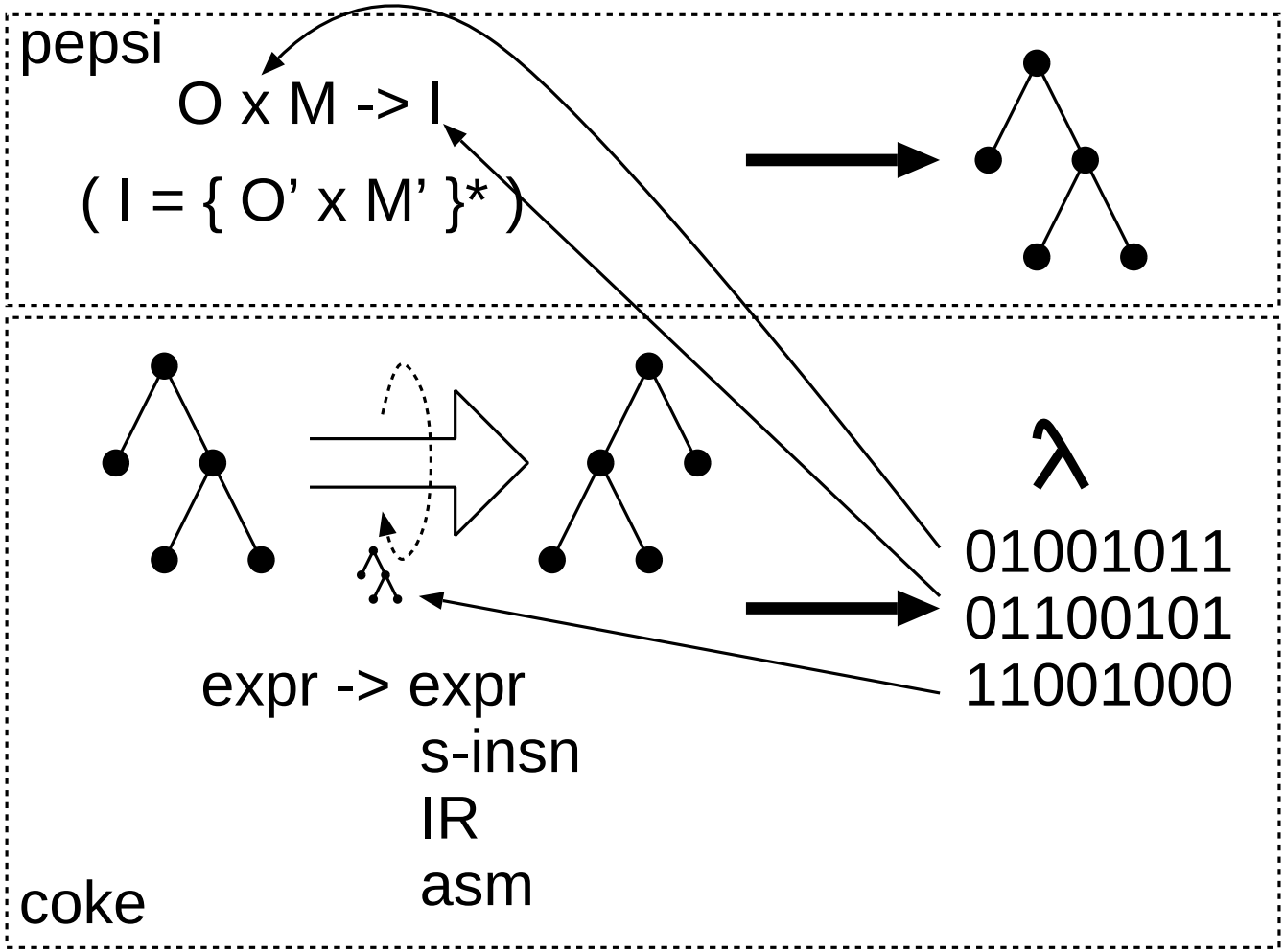
expr -> IR -> gen

intrinsic

meaning implements structure

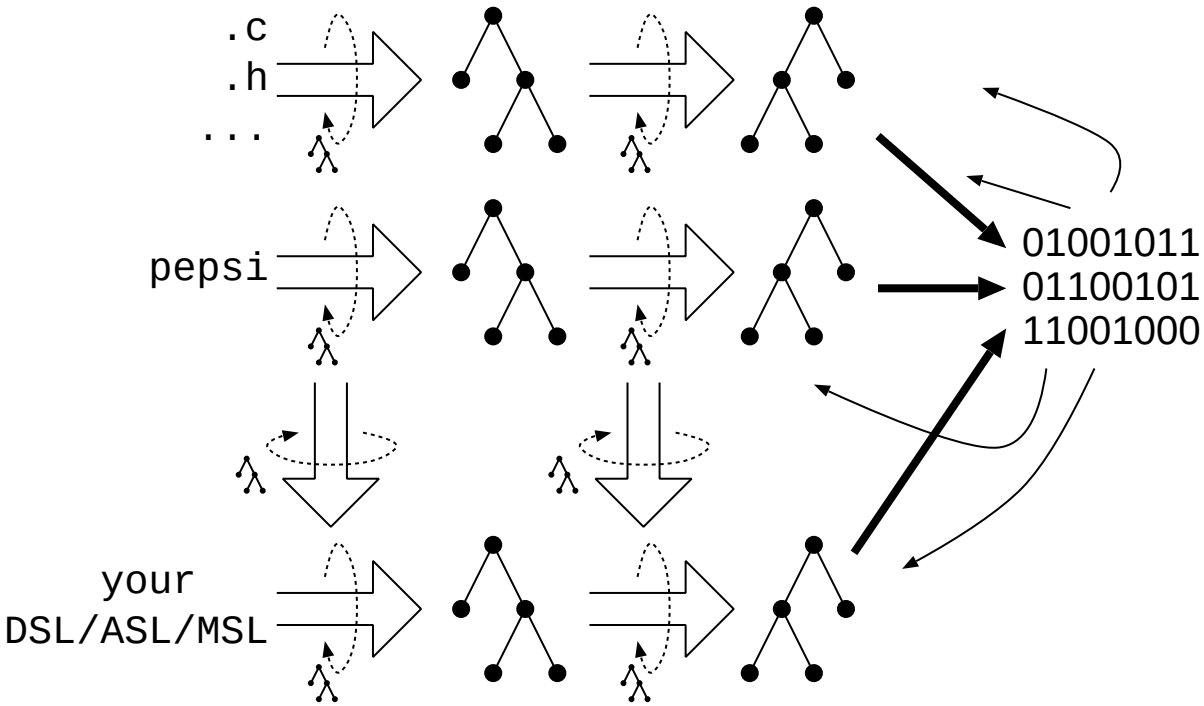


structure implements structure



everything is self-describing structure

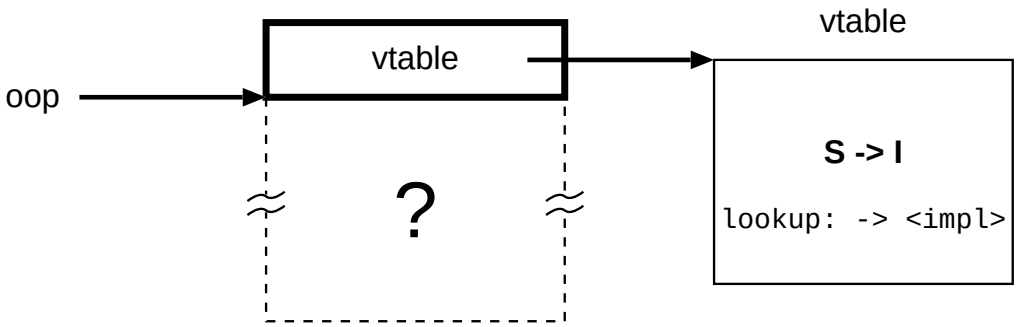
downwards, sideways



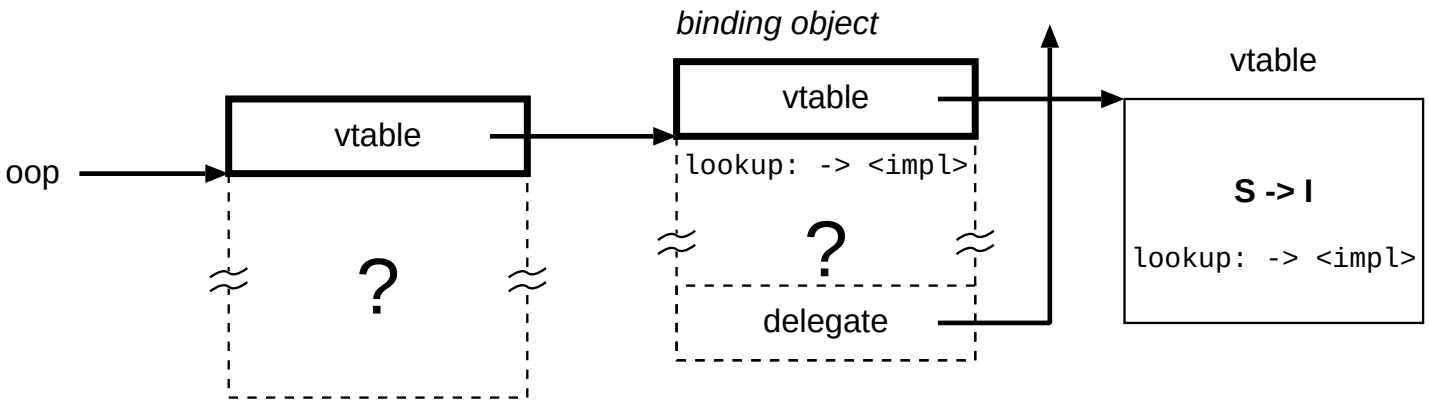
& upwards

- pattern matching
- nested, restartable regexps on structure
  - free text is structure, just like the rest

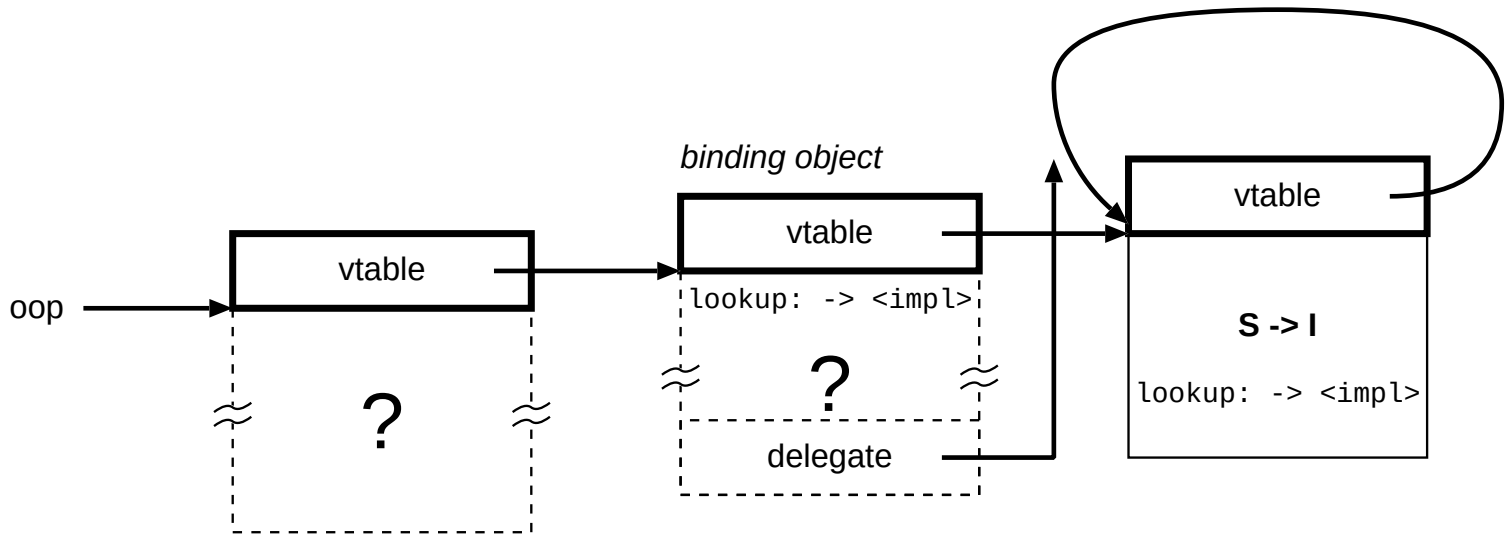
structure: intrinsic dynamic behaviour



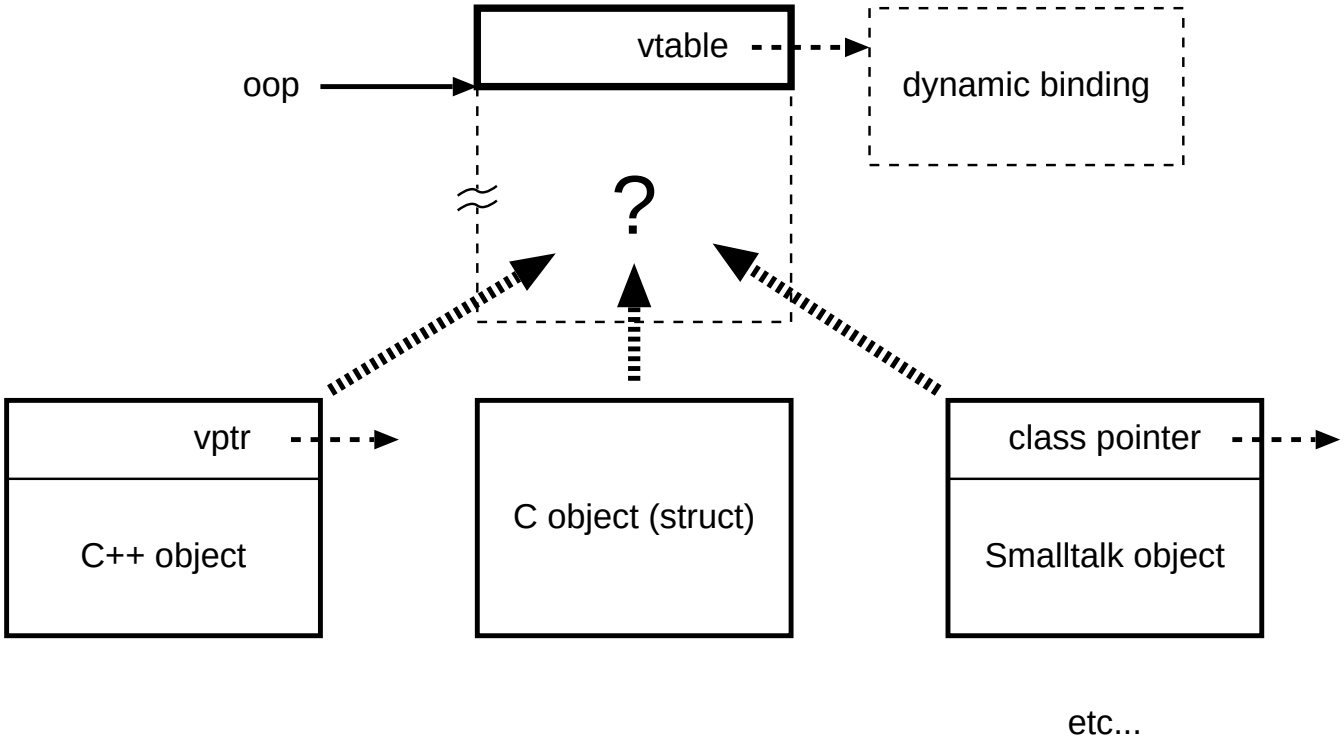
structure: dynamic dynamic behaviour



structure: everything is an object



encapsulation



- header (`.h`) parsing
- extract type and function signatures

dynamic binding

```
bind(object, message) =
    vtbl.lookup(vtbl, message)
```

method cache

```
bind(object, message) =
    vtbl ⟵ object[-1] ;
    cache[vtbl, message] ? cache[vtbl, message]
                         : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)
```

immediate types

```
bind(object, message) =
    vtbl ⟵ object == 0 ? vtbl_nil
                       : object & 1 ? vtbl_fixint
                                    : object[-1] ;
    cache[vtbl, message] ? cache[vtbl, message]
                         : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)
```

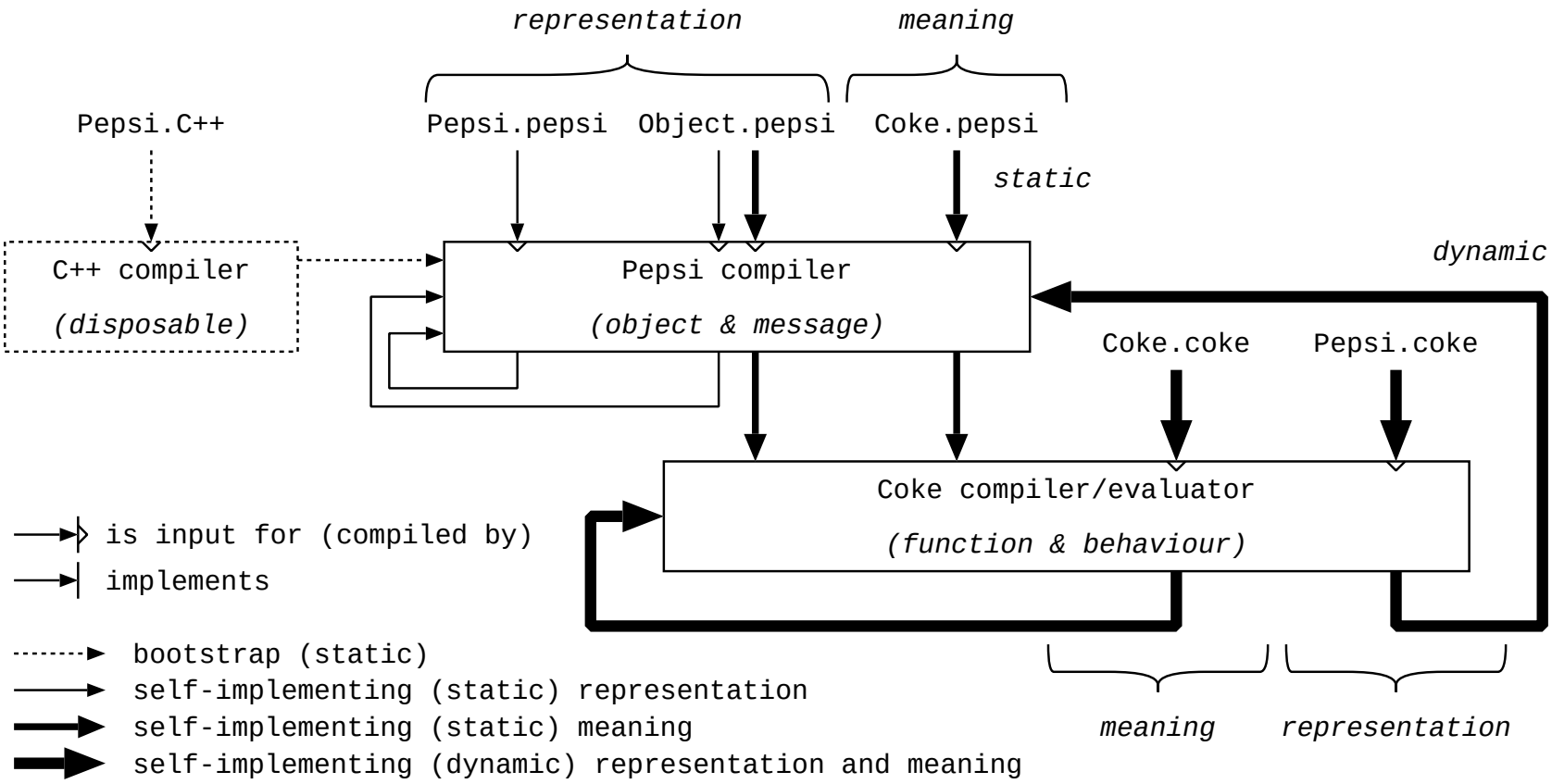
circular semantics

```
bind(object, message) =
    vtbl ⟵ object == 0 ? vtbl_nil
                       : object & 1 ? vtbl_fixint
                                    : object[-1] ;
    cache[vtbl, message] ? cache[vtbl, message]
                         : cache[vtbl, message] ⟵ vtbl.lookup(vtbl, message)

vtbl.lookup(vtbl, message) =
    (bind(vtbl, #lookup))(vtbl, message)
```

- message send implemented by sending messages
- override ⇒ new semantics

bootstrapping



conclusion

VVEE

- self-hosting virtualised virtual execution environment
- circularity complete: Coke implements Pepsi, dynamically
- everything open, reusable, dynamically modifiable
  - horizontal mutation into application domain
  - vertical integration of paradigms, abstractions
- no early-bound assumptions
- no artificial barriers

$\approx$ 3000 LOC
$\approx$ 400% Smalltalk
$\approx$ 60% C++

```
\endinput
```