

IBM, Armonk, 2004-09-16

Late Binding and Dynamic Implementation



Ian Piumarta, HP Labs

ian.piumarta@hp.com

<http://www-sor.inria.fr/projects/vvm>

VMs from the language designer's perspective

a VM provides:

- idealised abstract instruction set
 - abstract representation of program code closely matched to source language semantics
 - compact representation (bytecodes, . . .)
 - architecture-neutral “engine” hides underlying CPU/OS
- idealised memory abstraction
 - execution engine “sees” application objects, *not* bits/bytes/words/. . .
 - object semantics match source language semantics
 - transparent services: GC, weak pointers, finalisation, . . .
- idealised execution environment
 - virtual “OS” services insulate app from OS
 - memory, files, threads, windows, . . .
 - platform-neutral apps: “write once, run anywhere”^a

^astill pretty much a holy grail, although several dialects of Smalltalk come very close

background: problems with virtual machines

VMs are a pretty neat idea, but what's the problem with them today?

- rigidity

it's *difficult* to specialise them for a new application or domain

- adding new abstractions is complex, and in the domain of a few experts
- reusability is *depressingly* low
- specialisation is complex and inherently static
 - much work repeated for each new application
- portability is almost non-existent

and yet there are precedents that clearly indicate the need for specialisation. . .

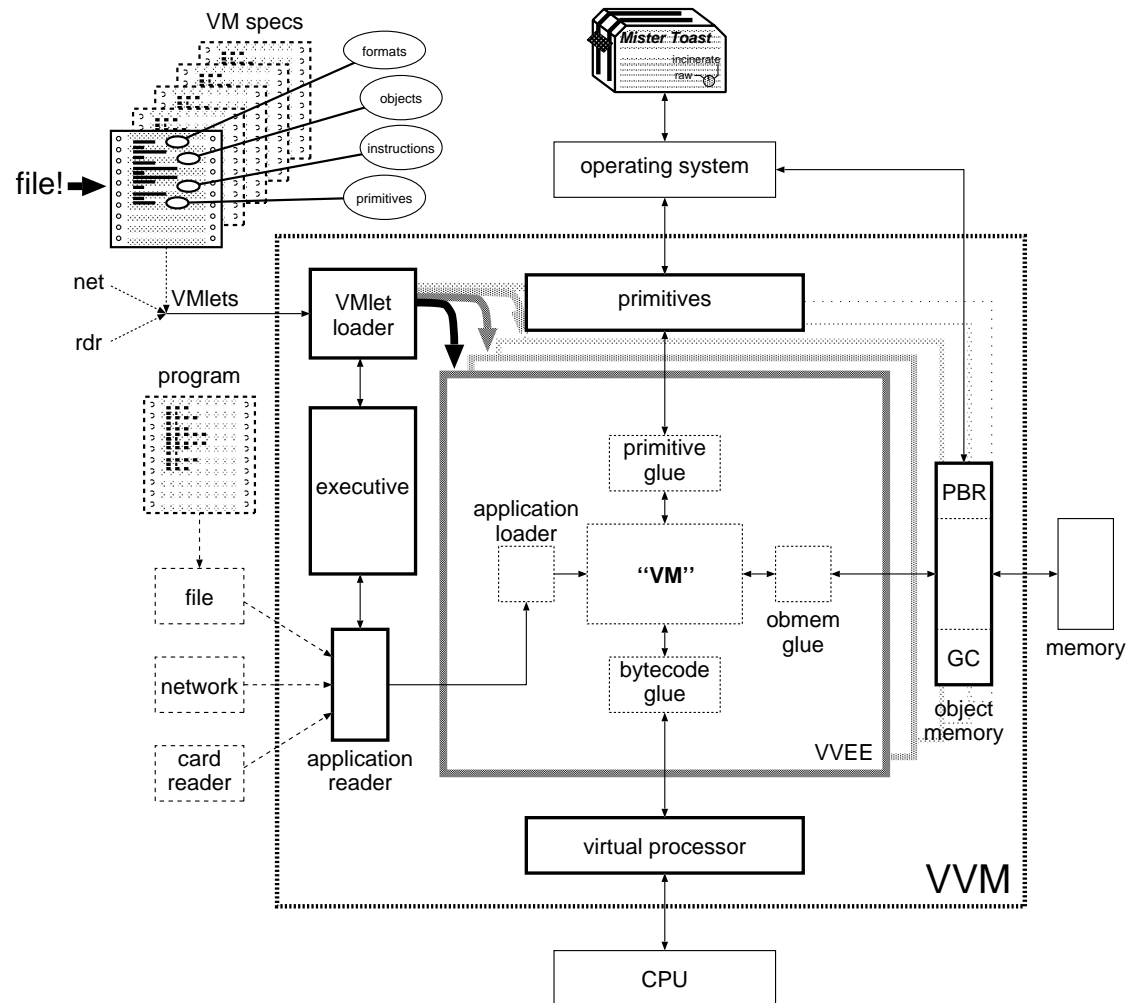
- OS “Kits”, [Personal,Embedded]Java[Card], OA scripting, . . .

. . . *and* evolution . . .

- GUI, protocols, algorithms (e.g., set-top boxes), . . .

virtual virtual machines

circa 1999



contentious statements

without justification:

- it is much easier to implement highly-dynamic languages/systems on top of a highly-dynamic language/system (than it is, e.g., on top of an entirely static language/system)

in other words: reuse (with/without modification) is easier than (re)implementation

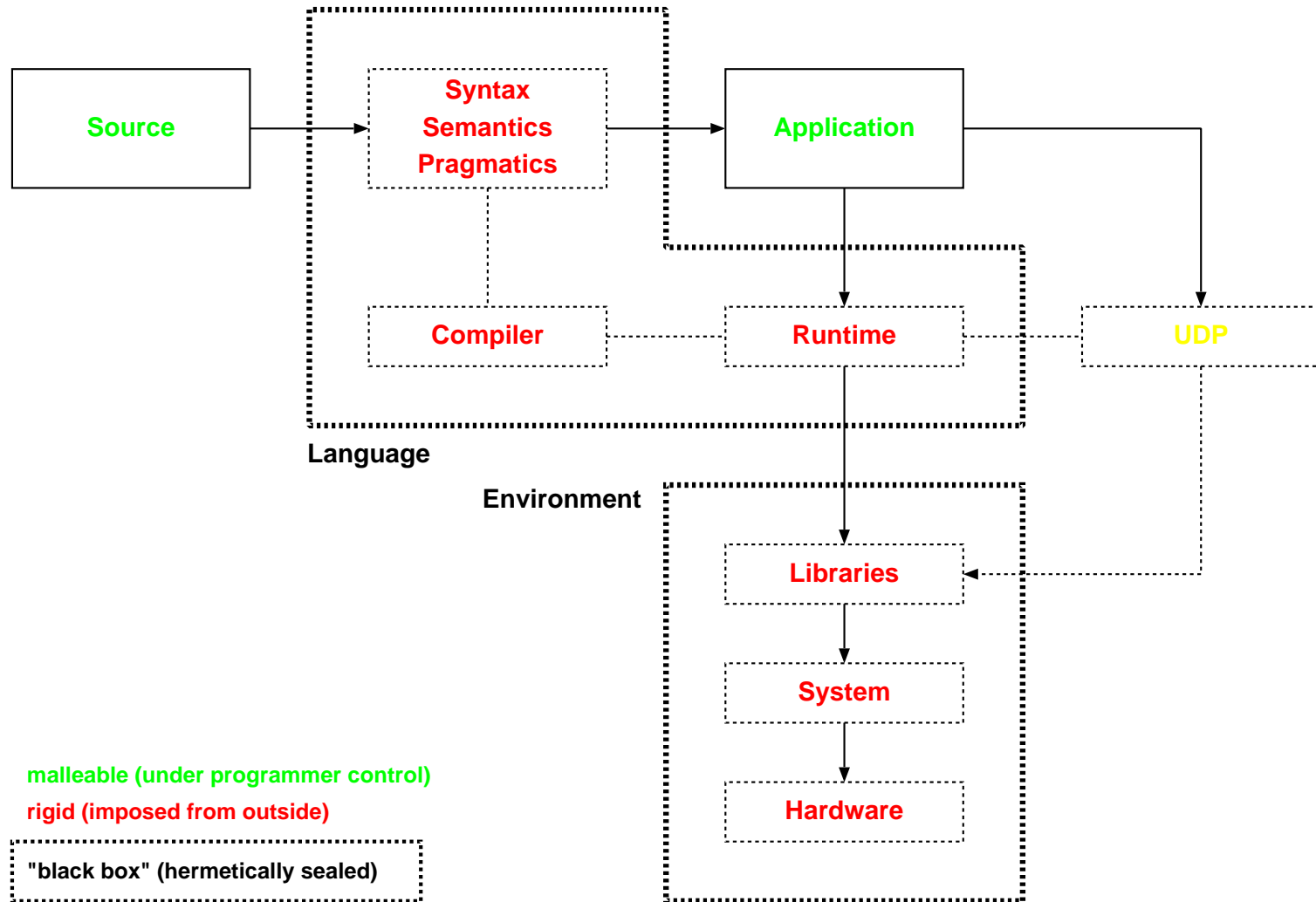
corollary:

- the more dynamic the base language/system, the easier it is to build a given dynamic language/system feature within

influences:

- BBC Basic: built-in 6502 asm
- ANDF

conventional programming languages



conventional programming languages

two black boxes:

- language
- environment

rigidity

- syntax (DSLs?)
- semantics (atomic test-and-set?)
- pragmatics (primitives?)
- libraries (protocol evolution / in-field servicing?)
- system (drivers, scheduling, resource management, . . . ?)

some observations about conventional languages

without much justification (for lack of time)

- *disasterously* early-bound
- insufficient meta-data
- no reification of implementation
- artificially-closed access to internals

(characteristics which are *inherited* by their application “offspring”)

⇒

- late-bind *everything* (including language implementation)
- make *everything* 1st-class (including input [‘programs’] and output [native code])
- leave the entire compilation chain open by default (but make it trivial to close it off entirely, selectively, or according to arbitrary verification policies as needed)

for some suitable definition of “*everything*”

(traditionally: the *simplest possible one*)

the simplest possible definition of “everything”

lowest common (implementation) denominator:

- platform’s native code
- C ABI
- connection to `libc` / `posix` / whatever ...

assume nothing about anything else

expose this “everything” to the application (or the app’s runtime)

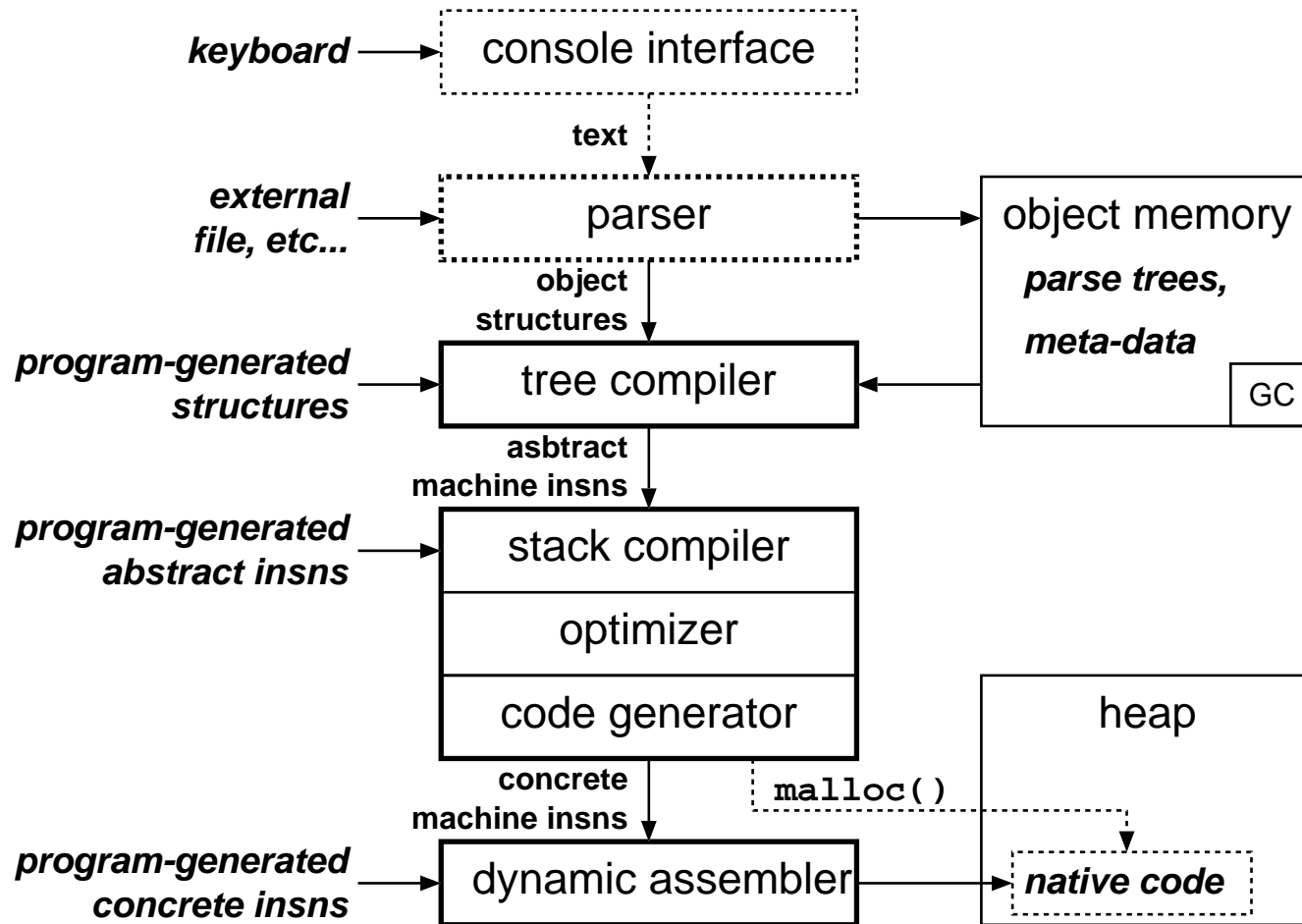
- in the simplest, least-constrained form possible

then go have (lots of) fun building maximally-dynamic systems on top of it

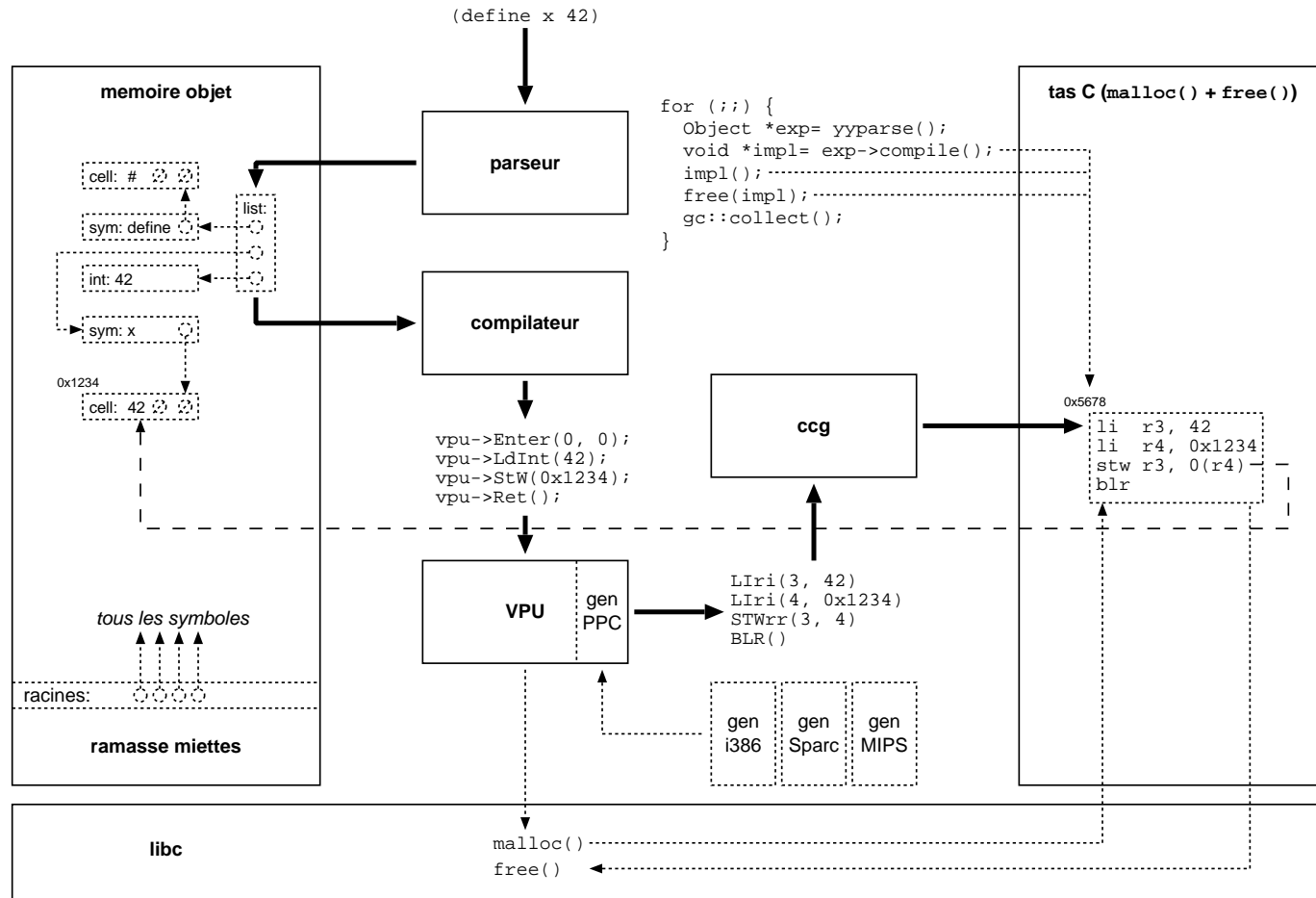
a universal dynamic compiler: YNVM

- dynamic compilation (to native code, C ABI)
- structured, language- / architecture- independent input
 - object structures (\sim “parse” trees) describing *C semantics*, built from a small number of object types (symbols, literals and lists)
- simple, persistent meta-data (program structure, environment)
 - using the same objects as the structured input
 - \Rightarrow *programs = data = metaⁿ data = programs*
- customisable semantics
 - application-defined transformations on input data structures
 - applied at runtime (during compilation) & can be *redefined* on-the-fly
- access to all levels of compilation & code generation
 - the compiler itself is *reified*, *1st-class data* in the application’s address space (don’t like the compiler? *redefine it* on-the-fly. . .)
- single point of access to the host system: `dlsym` (or equivalent)
 - hence (by transitivity) arbitrary & unlimited access to all OS services

architecture



architecture



symbols

contain three kinds of value:

- primitive
- object
- syntactic

hierarchical *namespaces*

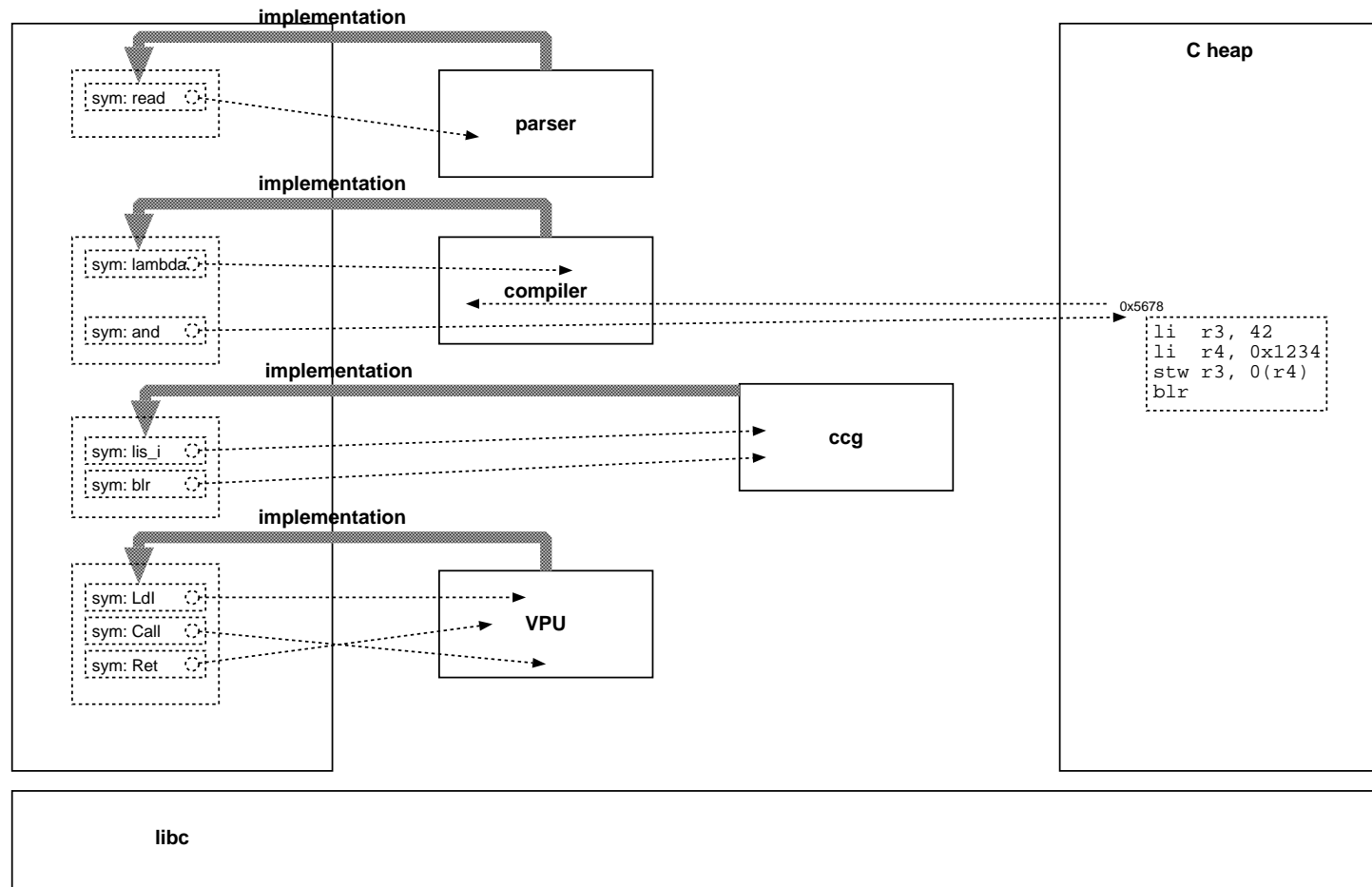
- any symbol's *object value* can be a list of symbols (another namespace)
⇒ hierarchy

named semantics

- *syntactic value* = closure invoked during compilation

⇒ context governs meaning

implementation is reified in its own metadata



...

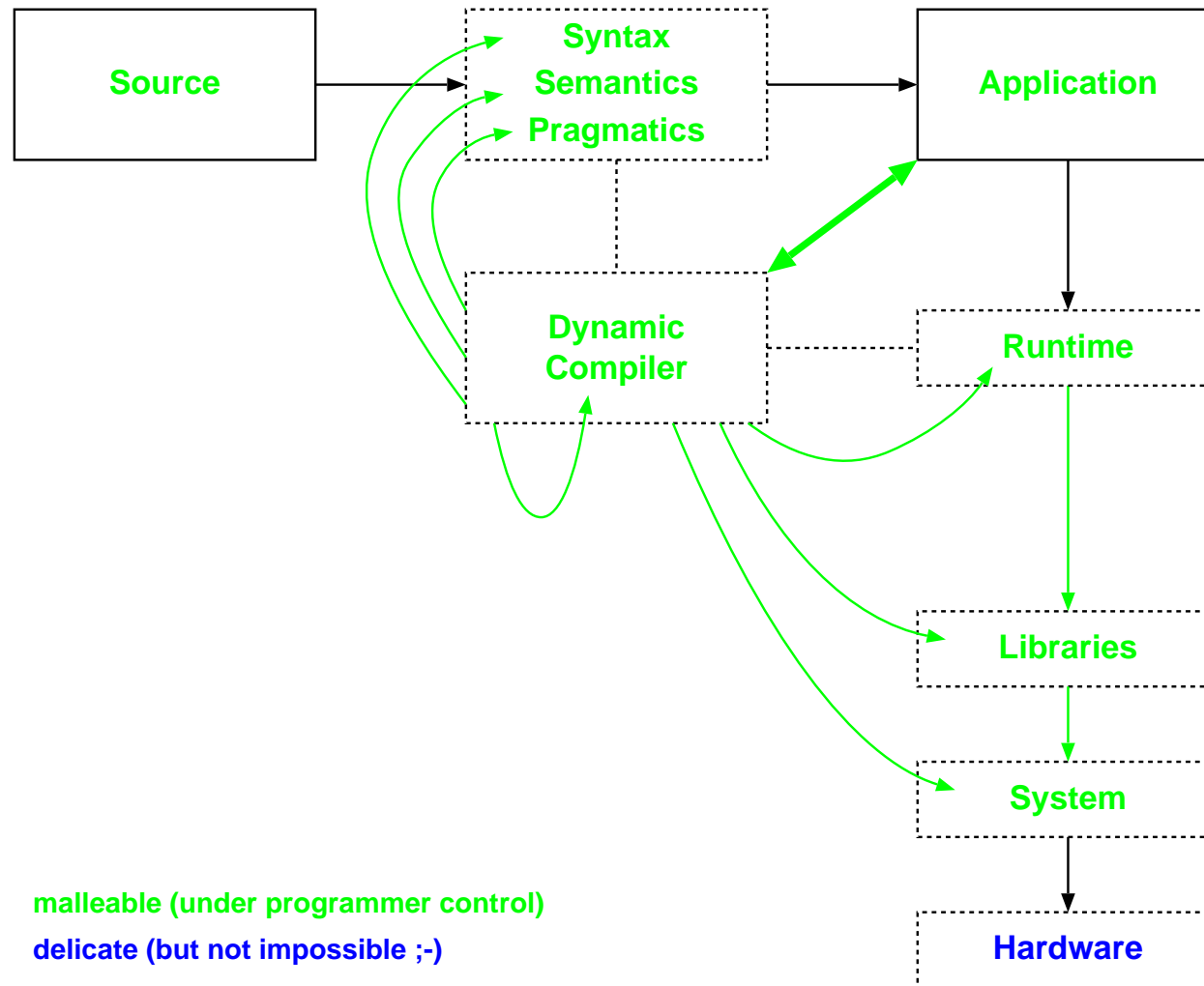
infinite extensibility

runtime implements dynamic compiler over its own implementation domain

every intrinsic semantic operation is (pre)defined as an (user-defined) extension

- no “privileged” meaning
- anything can be modified/removed/added on-the-fly
- unlimited malleability, under program control

unconventional programming languages

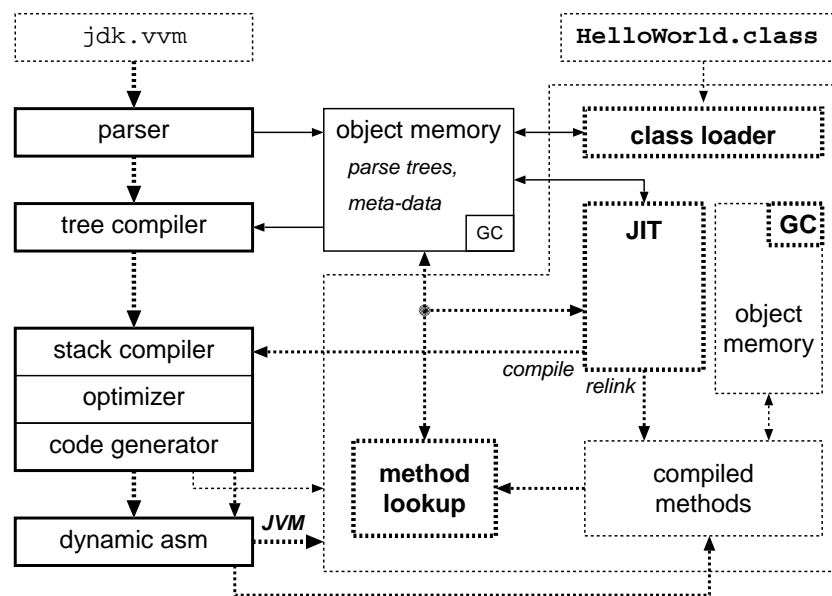


malleable (under programmer control)
delicate (but not impossible ;-)

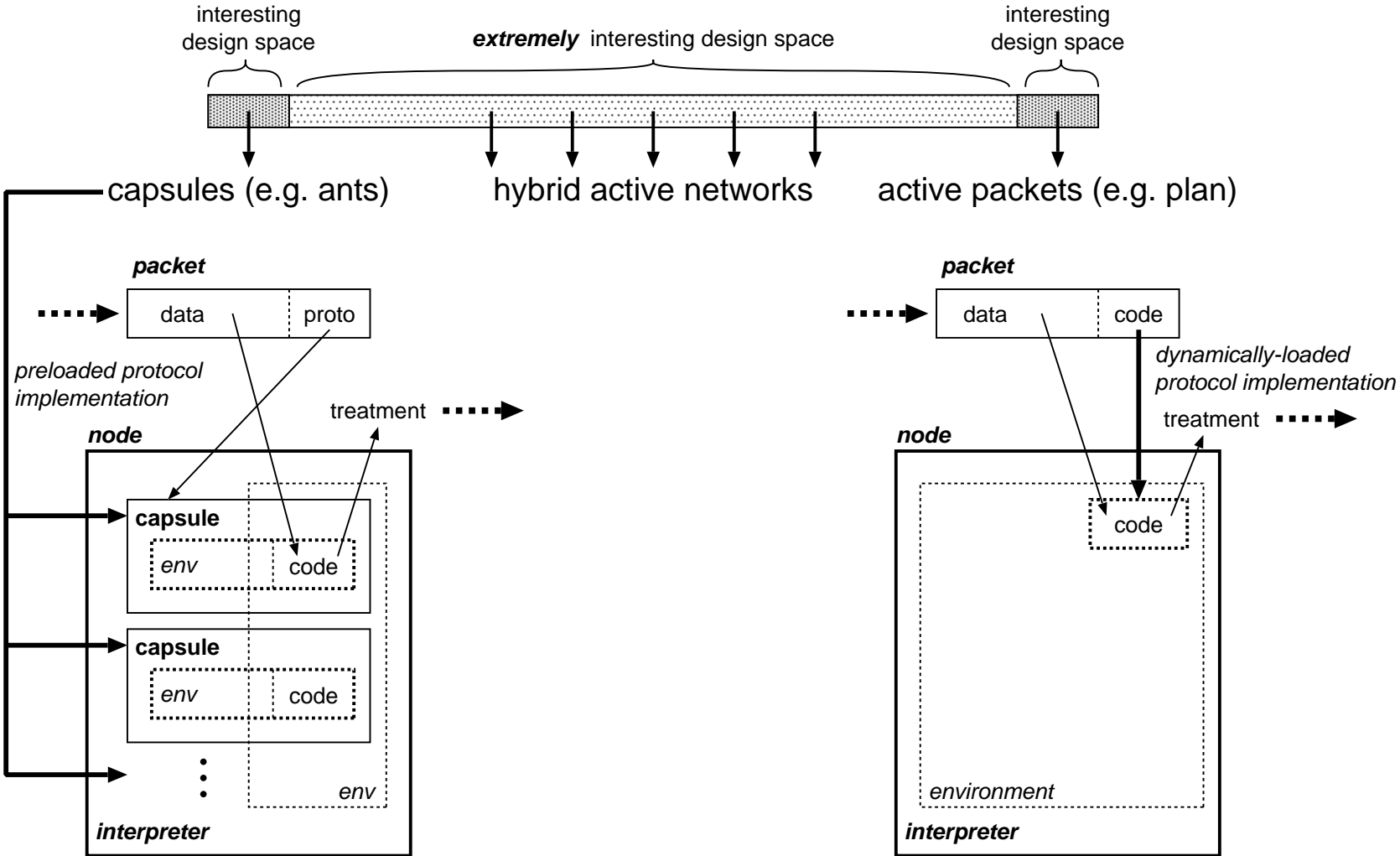
virtual machines

Java, for example. . .

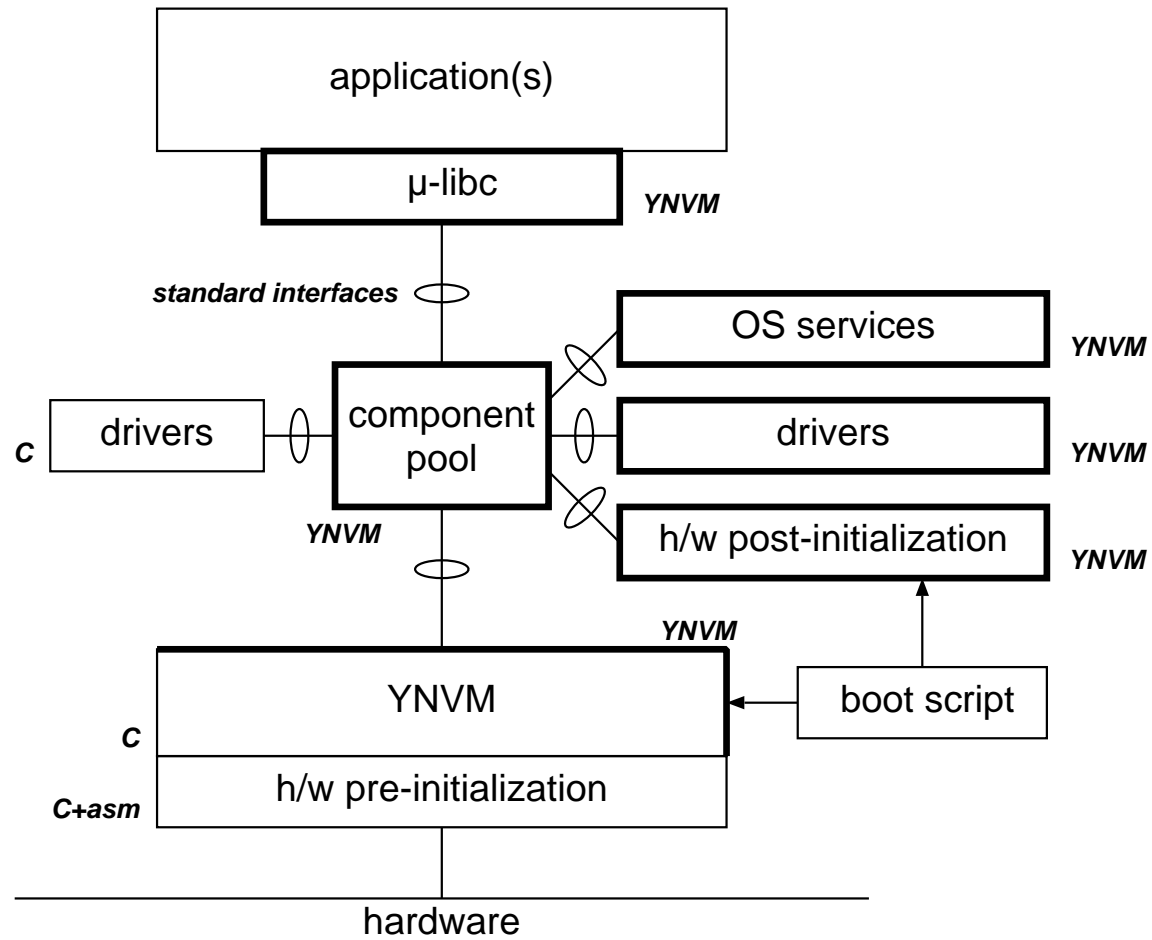
- modify the compilation chain on-the-fly (3 KLOC)
 - YNVM replaces a vertical section of a “traditional” JIT compiler
 - then introspects and *intercedes* on the entire JIT+JVM+etc. . .
- new object formats, real runtime MOP, dynamically-defined native methods, etc. . .



active networks



embedded systems



example: inline cache detail

```
(define (syntax ->)
  (lambda (form)
    ; (-> object (selector args))
    (syntax.match form (? ,recv ( ,[object.symbol? sel] ,@args ))
      (let ([posic (:posic.malloc)])
        `(let ([_recv ,recv])
           ((if (= (:class.class-of _recv)
                  (:posic.prev-class ,(object.integer posic)))
                (:posic.dest-meth ,(object.integer posic))
                (relink ,(object.integer posic)
                        _recv
                        ,(object.integer sel))))
            _recv
            ,@args))))))
```

≈ 1.98 × static C function call

(various demos offline for anyone interested)

the catch(es)

people are terrified of too much flexibility

need for stable intermediate forms (“layers” of abstraction)

- in deployment
- in “vertical” system / language design
 - code gen, intrinsic forms, etc. . .
 - where are the borders of the VM?
 - APIs accessible only to certain groups

c.f., Herbert Simon, *parable of the watchmakers*

- complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not; the resulting complex systems in the former case will be hierarchic.

the catch(es)

C++ intrinsics

- `vtbl` is not 1st class
- modification of object (memory) behaviour is ugly
- modification of stack compiler is ugly

⇒

- pure C (macro assembler) implementation
- proto + delegate (50 LOC in bootstrap)
- everything 1st class
- 100% malleable down to the metal

the catch(es)

type safety

- dynamically-typed objects, messages
- statically-typed native code (platform headers)

self-implementing bootstrap

- dynamic – static compilation “slider”

reoptimisation for performance (llvm, ...)

dynamic specialisation (FDO) vs. static specialisation (DSL, ...)

isolates for domain protection

metrics?

buzzwords

- configurability, extensibility, evolution;
- in-field update (dynamic upgrade of running engine code)
- administration, response to change, reduced total cost of ownership
- interoperability: implementation languages with fine-grained communication

language designers/implementors: *freedom to innovate*

- expressivity; application, language, runtime;
- distribution, runtime adaptability (dynamic proxies, interfaces, ...)

target-specific dynamic compilation

dynamic optimisations: self-modifying IR code

performance: dynamic vs. static compilation

security: `erights.org`

metrics?

compatibility and startup costs

- C ABI: performance loss?
- mixed dynamic/static code? (cf., gcj trick)
- dynamic – static slider

extending the runtime environment:

- increasing granularity (higher level abstractions → language)
- vs. decreasing granularity (lightweight transactions → language/runtime prim w/ associated syntax/semantics)

how much of system is written in managed code?

- microkernels
- lkms

metrics?

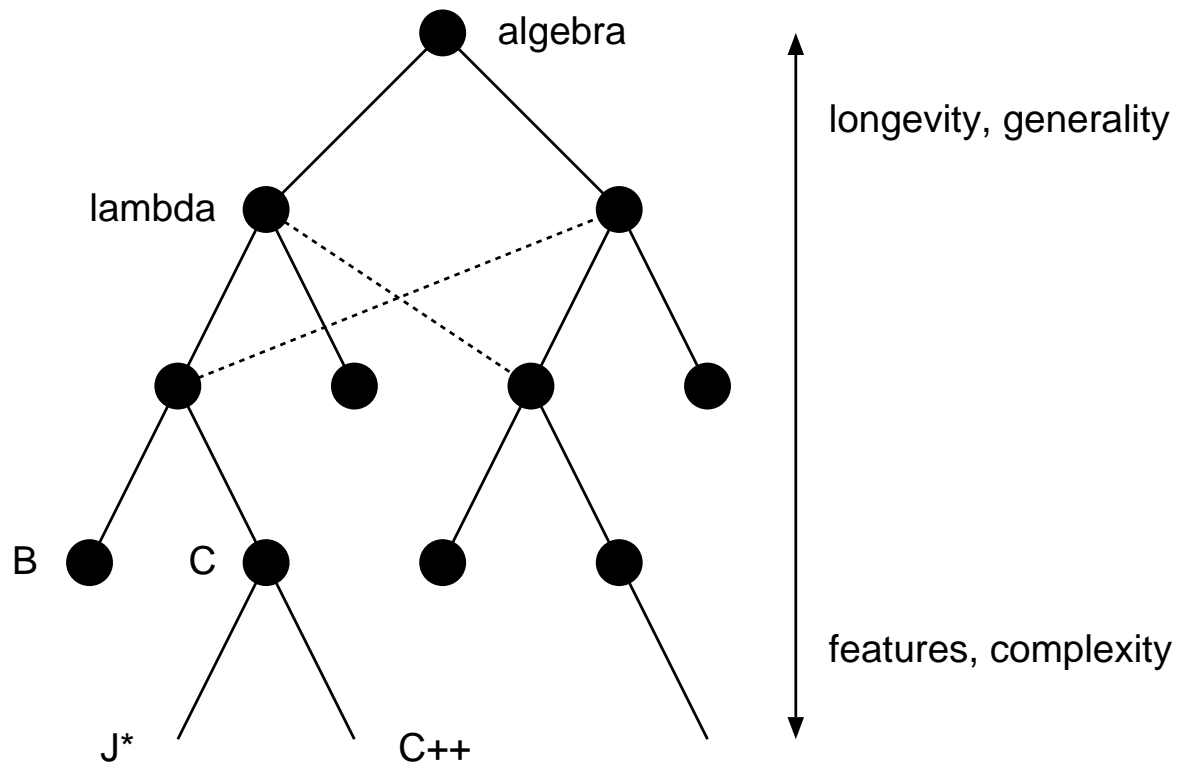
new language with features that don't fit?

- impossible to predict the future
- must not legislate paradigms/models
- policies are separate from implementation/mechanisms
- can you do continuations in it?

right tool for job:

- right language for job
- right level (abstraction) in architecture for the (implementation) job

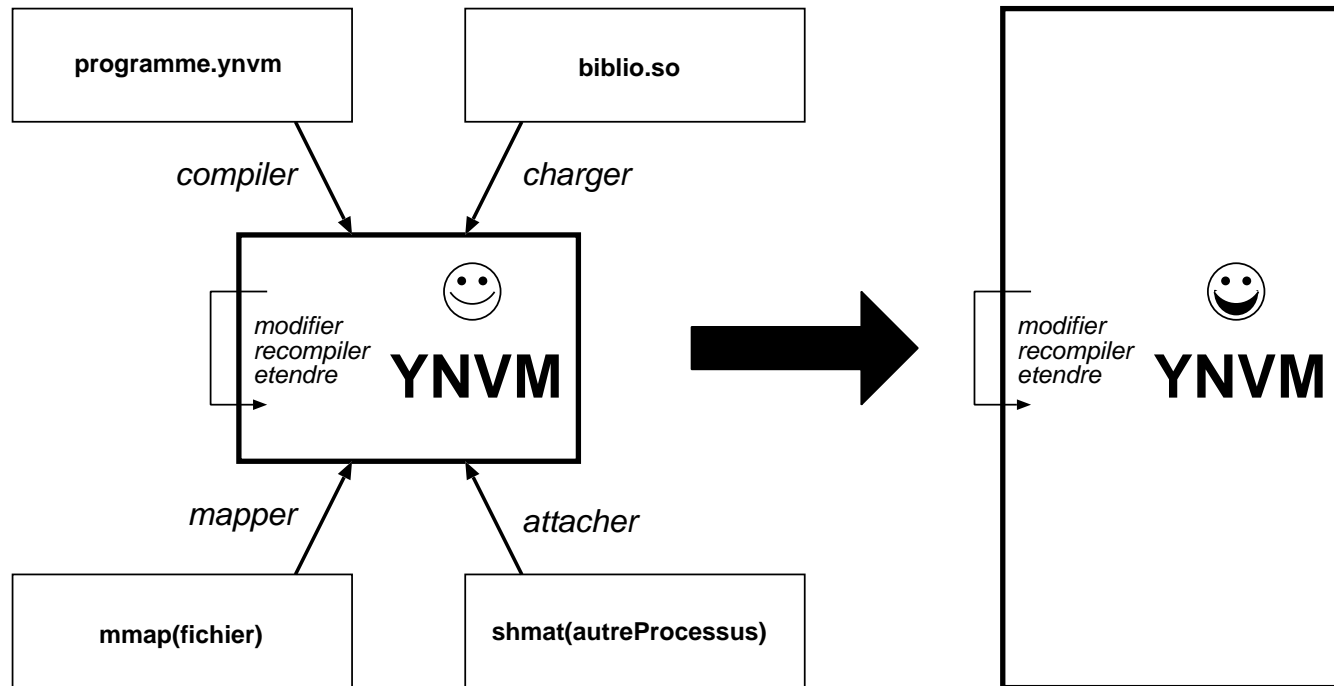
conclusion



conclusion

- choice of flexibility / complexity “compromise”
 - 5 (well-defined, in reality even more) levels of entry
- dynamic compilation overhead seen by applications
 - ≥ 3 MB native code per second (on current hardware)
- symbolic space affects semantic space
 - domain-specific compiler customisation
 - design & optimise your solution space, then use it
- agility
 - programs are (meta) data are programs are ...
 - semantics are dynamic (interpret or compile the same code)
 - the compiler’s implementation is just part of the client’s namespace
 - system state created incrementally, described dynamically, changes felt immediately
- simplicity: ~ 10 KLOC, 250 KB code, for *everything*
- performance: 0.85 – 1.15 static (optimised) C programs

conclusion-in-a-picture



recommendations

ideally everyone shares a single, malleable “implementation engine”

otherwise:

- include a C semantics (and ABI) JIT
- maximise the potential for reimplementing of static parts of system at runtime
- design out all the barriers to innovation etc. . .
- adopt and apply (rigorously) simple (but entirely sufficient) safety practices

(then put it in your (OS) kernel!)

recommendations

safety/security: `erights.org`

- principle of least authority
- capabilities as (object) references: good OO practice (nothing more, nothing less, *bulletproof*)

live inside the kind of system you build

reduce to the max: simplicity

parse trees (not bytecodes) as an archival/interchange format

open source (and freely licensed)