COG

Back to the Future, Part II

faster open source VMs for
Croquet, Squeak & Newspeak

- Why?
- What?
- Where to & when?

Why Cog?

- Qwaq Forums
- SAS
- Client experience

Why Cog?

- Small part of a larger whole

cog•no•men |käg'nōmən; 'kägnəmən|

noun
an extra personal name given to an ancient Roman citizen, functioning rather like a nickname and typically passed down from father to son.
- a name; a nickname.

Why Cog?

- Success is 99% failure

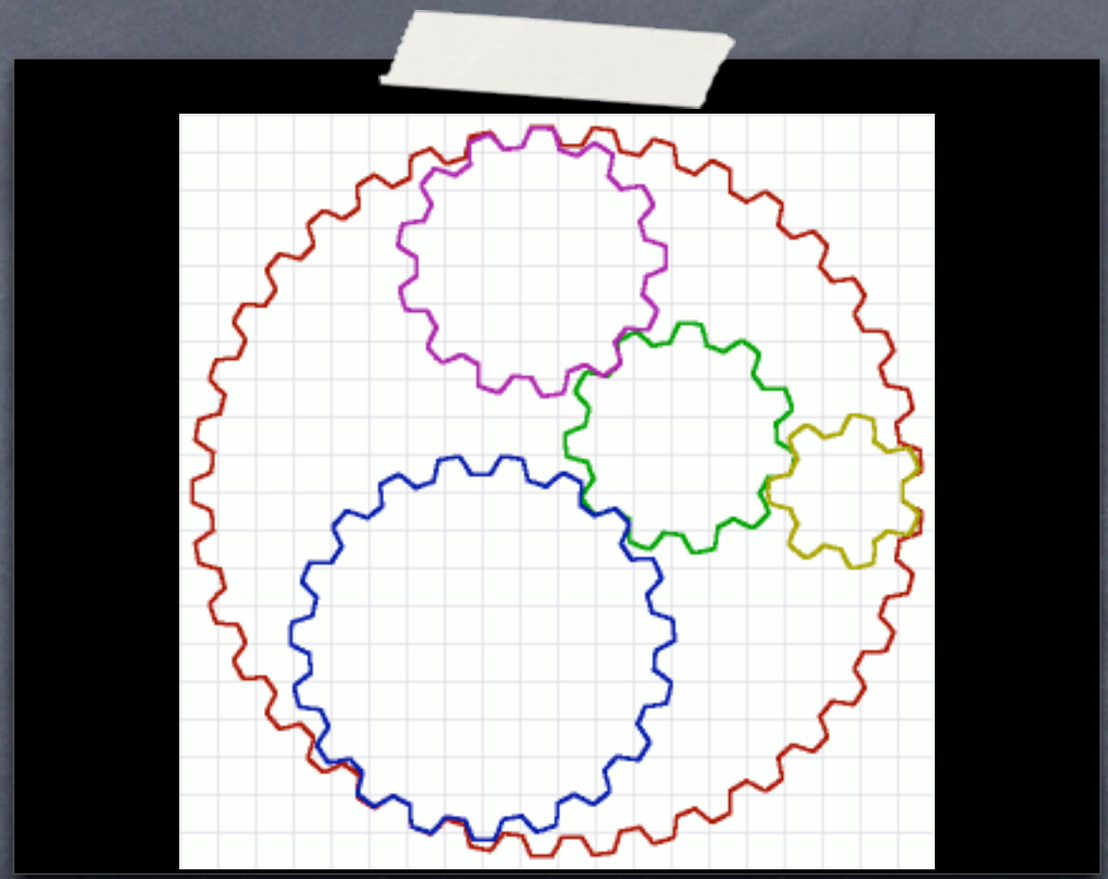
MARKETING GIRL:
When you have been in marketing as long as I have, you'll know
that before any new product can be developed,
it has to be properly researched.
I mean yes, yes we've got to find out what people want from fire,
I mean how do they relate to it, the image -

FORD:
Oh, stick it up your nose.

MARKETING GIRL:
Yes which is precisely the sort of thing we need to know, I mean
do people want fire that can be fitted nasally?

get the marketing out of the way early...

What's Cog?



a logo!

What's Cog?

- a blog
  http://www.mirandabanda.org/cogblog
- Community Organised Graft
- Contributions Over<->taken Gratefully

What's Cog?

- a series of VM evolutions
  - bytecode sets
  - object representations
  - execution technology
- DTSSTCPW/KISS/Evolve
  - not a new compiler
  - not a replacement VM
  - no new GC, new CompiledMethod, etc

Where to?

- Targets:
  - HPS style fast JIT
  - Self style quick JIT
  - Integrate with Hydra
  - Integrate with Spoon

fast car

quick car

Where to?

- Targets:
  HPS style fast JIT
  Self style quick JIT
- Prerequisites
  - Closures
  - Internal stack organization
  - polymorphous Inline Caches

baby steps

- Closures + Closure VM
  - extend existing compiler
    front-end, new back-end
  - 5 new bytecodes
  - sionara BlockContext

has ANSI block syntax
nicely familiar
less work

Croquet 1.0, Squeak 3.9
www.mirandabanda.org/downloads
Deployed internally at Qwaq

closures enable stack representation

```
inject: aValue into: binaryBlock
    | next |
    next := aValue.
    self do: [:each| next := binaryBlock value: next value: each].
    ^next
```

```
inject: aValue into: binaryBlock
    | tempVector |
    tempVector := Array new: 1.
    tempVector at: 1 put: aValue.
    self do: [:each|
        tempVector
            at: 1
            put: (binaryBlock value: (tempVector at: 1) value: each)].
    ^tempVector at: 1
```

baby steps

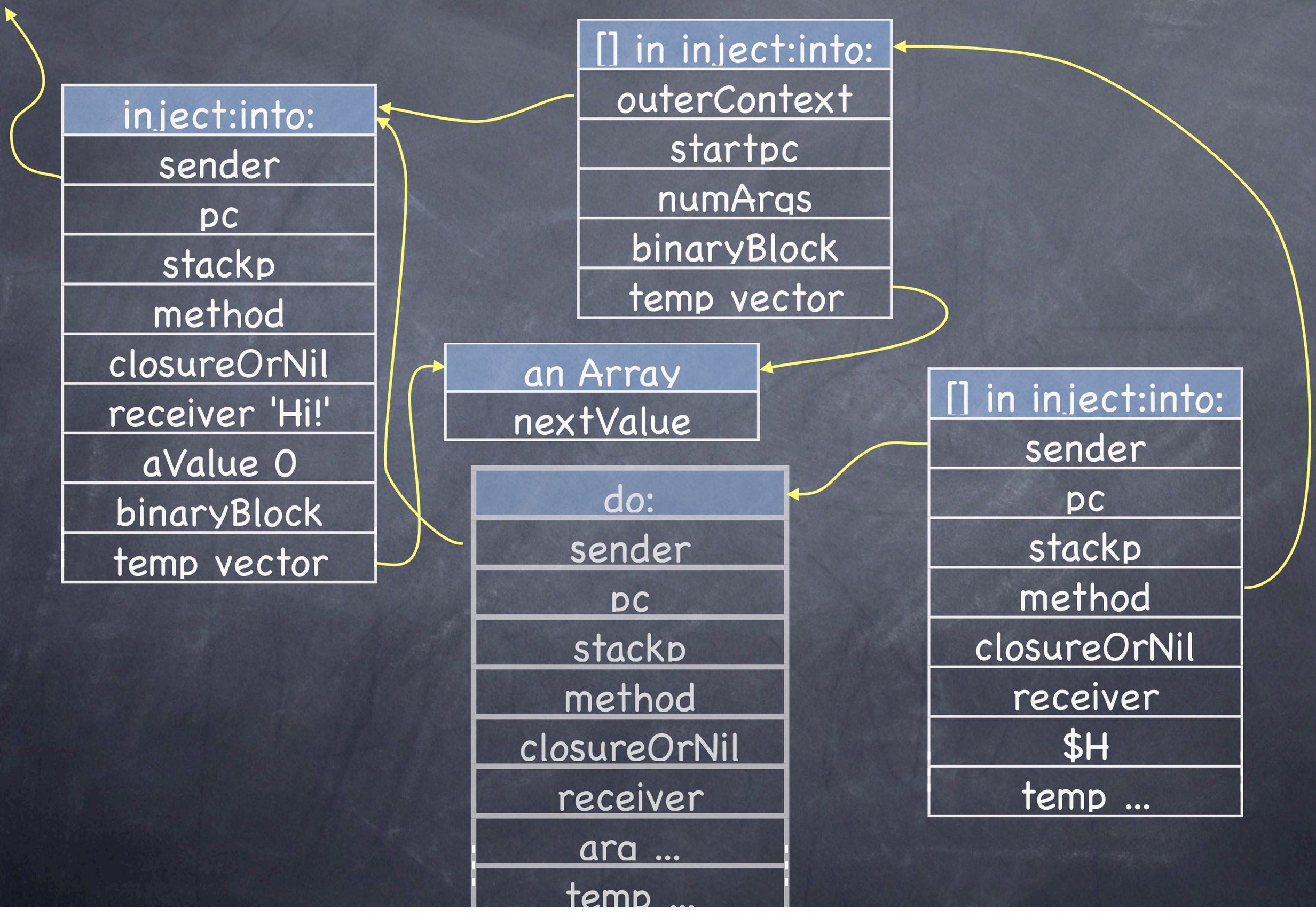
- - BlockClosure - BlockContext + BlockClosure

Object **variableSubclass:** #BlockClosure
instanceVariableNames: 'outerContext startpc numArgs'

BlockClosure methods for evaluating
value[:value:value:value:]
valueWithArguments:

ContextPart **variableSubclass:** #MethodContext
instanceVariableNames: 'method receiverMap closureOrNil receiver'

closure activation



baby steps

- 5 new bytecodes

  pushNewArrayOfSize:/pushConsArrayWithElements:

  pushClosureCopyNumCopiedValues:numArgs:blockSize:

  pushRemoteTemp:inVectorAt:

  popRemoteTemp:inVectorAt:

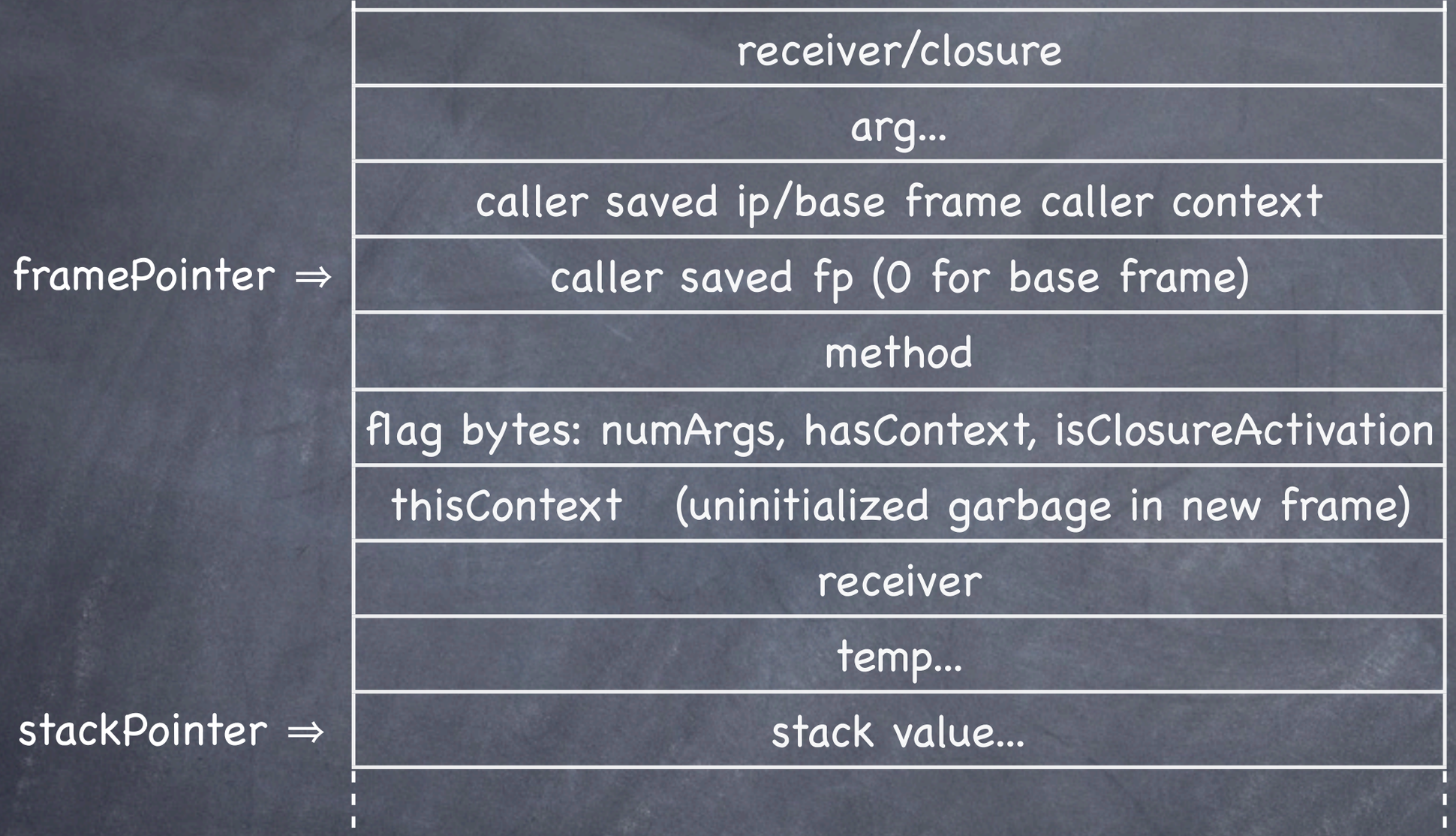
  storeRemoteTemp:inVectorAt:

- change return bytecode to do non-local return if closureOrNil not nil

Stack Interpreter

- Closure VM + Internal Stack Organization
  - activations are stack frames on stack pages
  - contexts on heap are proxies for stack frames
- Streamlined GC, no pop/pushRemappableOop:

stack frame



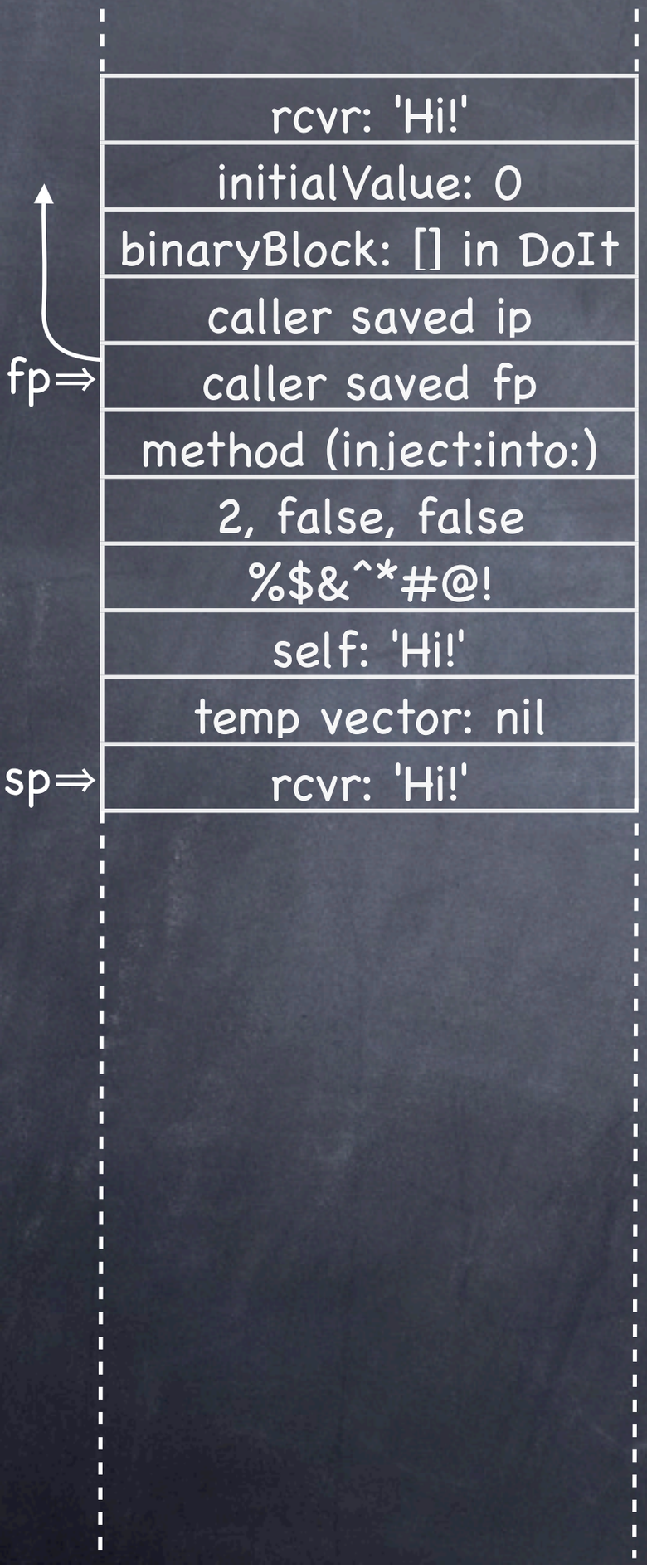
- no argument copying
- lazy context creation
- slower argument access!! (in interpreter, not in JIT)
- epsilon away from fast JIT organization (maybe coexist)

stack activation

```
'Hi!' inject: 0 into:
    [:sum :char| sum + char asInteger]

inject: thisValue into: binaryBlock
    | nextValue |
    nextValue := thisValue.
    self do:
        [:each |
        nextValue := binaryBlock value: nextValue value: each].
    ^nextValue
```

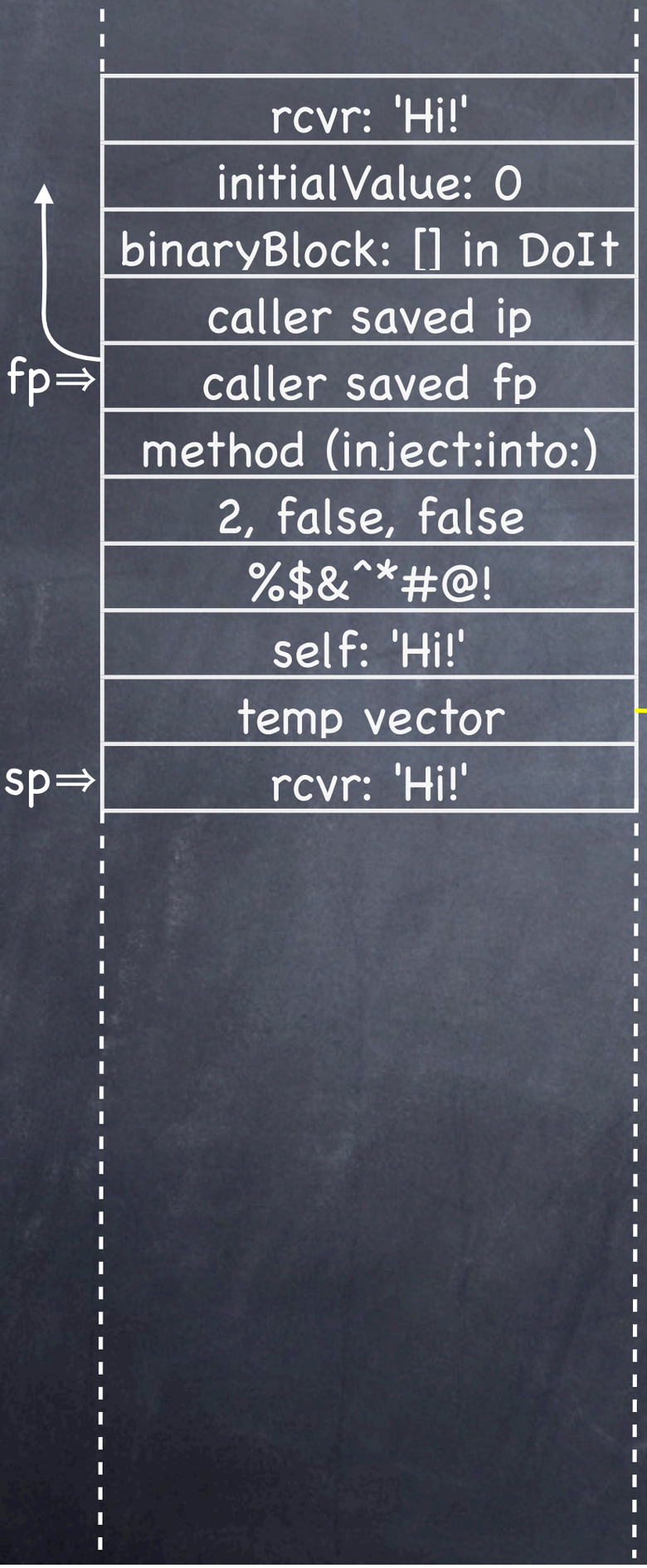
stack activation




```
'Hi!' inject: 0 into:
    [:sum :char| sum + char asInteger]

inject: thisValue into: binaryBlock
    | nextValue |
    nextValue := thisValue.
    self do:
        [:each |
        nextValue := binaryBlock value: nextValue value: each].
^nextValue
```

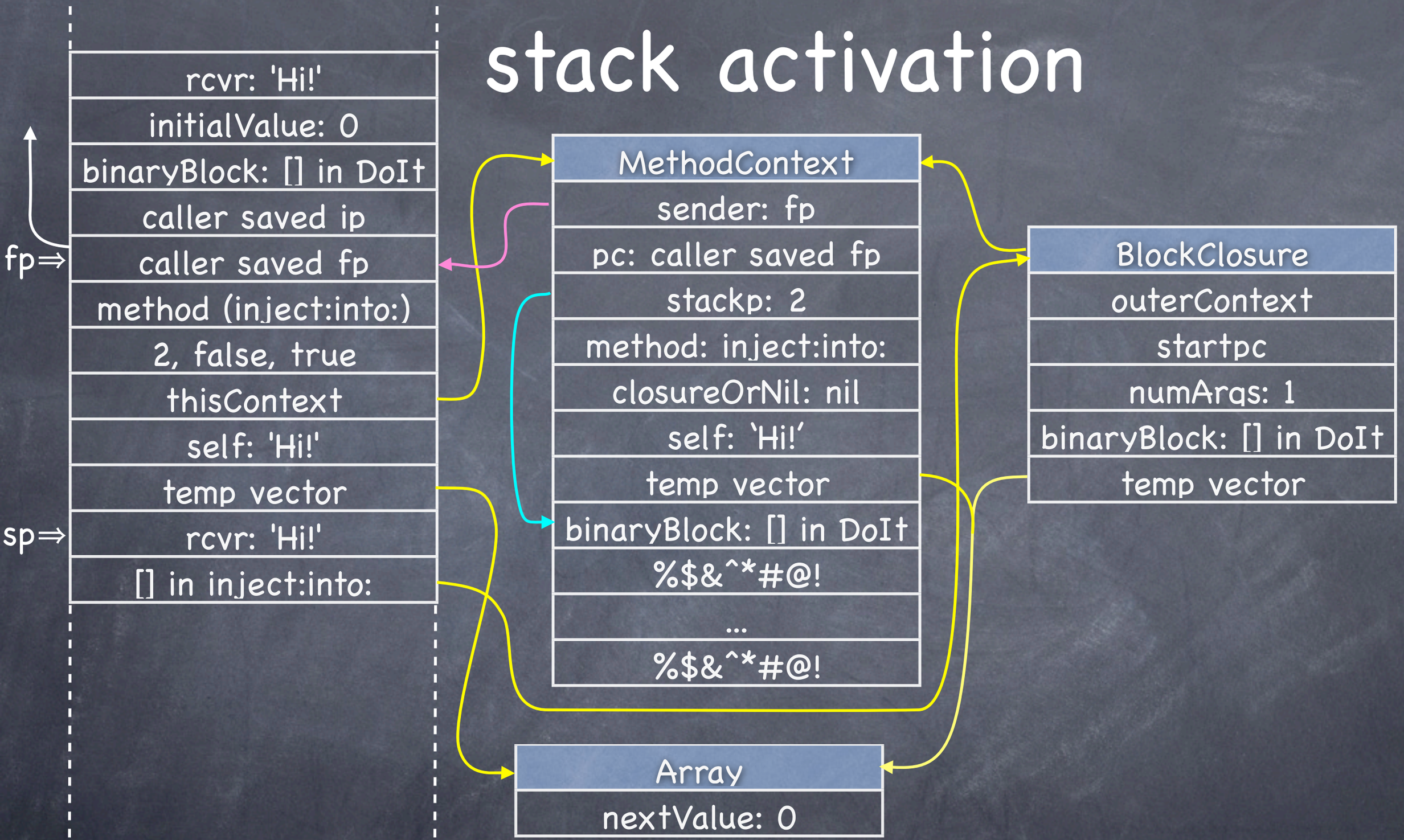
stack activation



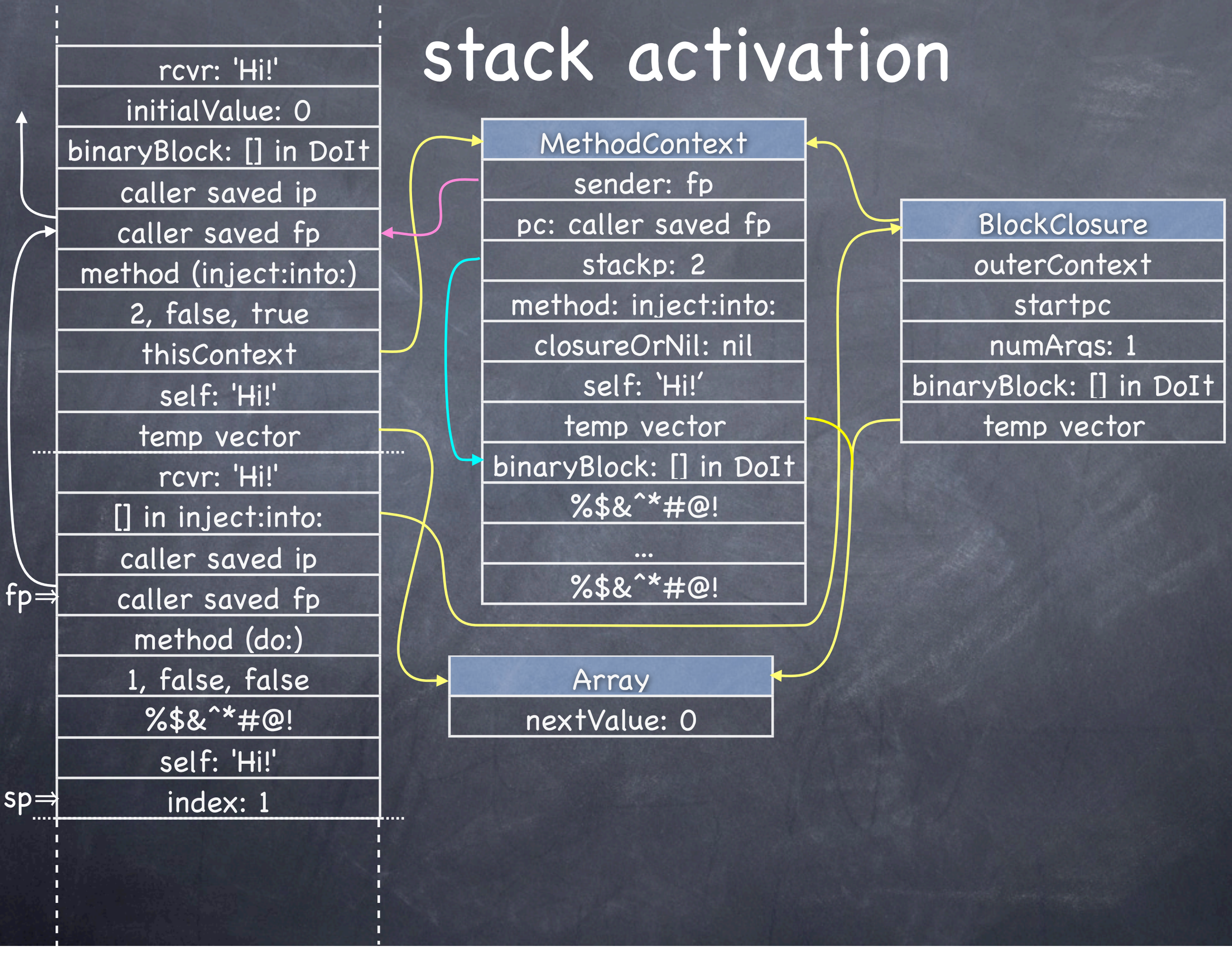

```
'Hi!' inject: 0 into:
    [:sum :char| sum + char asInteger]

inject: thisValue into: binaryBlock
    | nextValue |
    nextValue := thisValue.
    self do:
        [:each |
        nextValue := binaryBlock value: nextValue value: each].
^nextValue
```

stack activation



stack activation



stack activation

rcvr: 'Hi!'
initialValue: 0
binaryBlock: [] in DoIt
caller saved ip
caller saved fp
method (inject:into:)
2, false, true
thisContext
self: 'Hi!'
temp vector
rcvr: 'Hi!'
[] in inject:into:
caller saved ip
caller saved fp
method (do:)
1, false, false
%$&^*#@!
self: 'Hi!'
index: 1

MethodContext
sender: fp
pc: caller saved fp
stackp: 2
method: inject:into:
closureOrNil: nil
self: 'Hi!'
temp vector
binaryBlock: [] in DoIt
%$&^*#@!
...
%$&^*#@!

Array
nextValue: 0

BlockClosure
outerContext
startpc
numArgs: 1
binaryBlock: [] in DoIt
temp vector
clsr: [] in inject:into:
each: $H
caller saved ip
caller saved fp ⇐fp
method (inject:into:)
1, true, false
%$&^*#@!
self: 'Hi'
[] in DoIt
temp vector ⇐sp

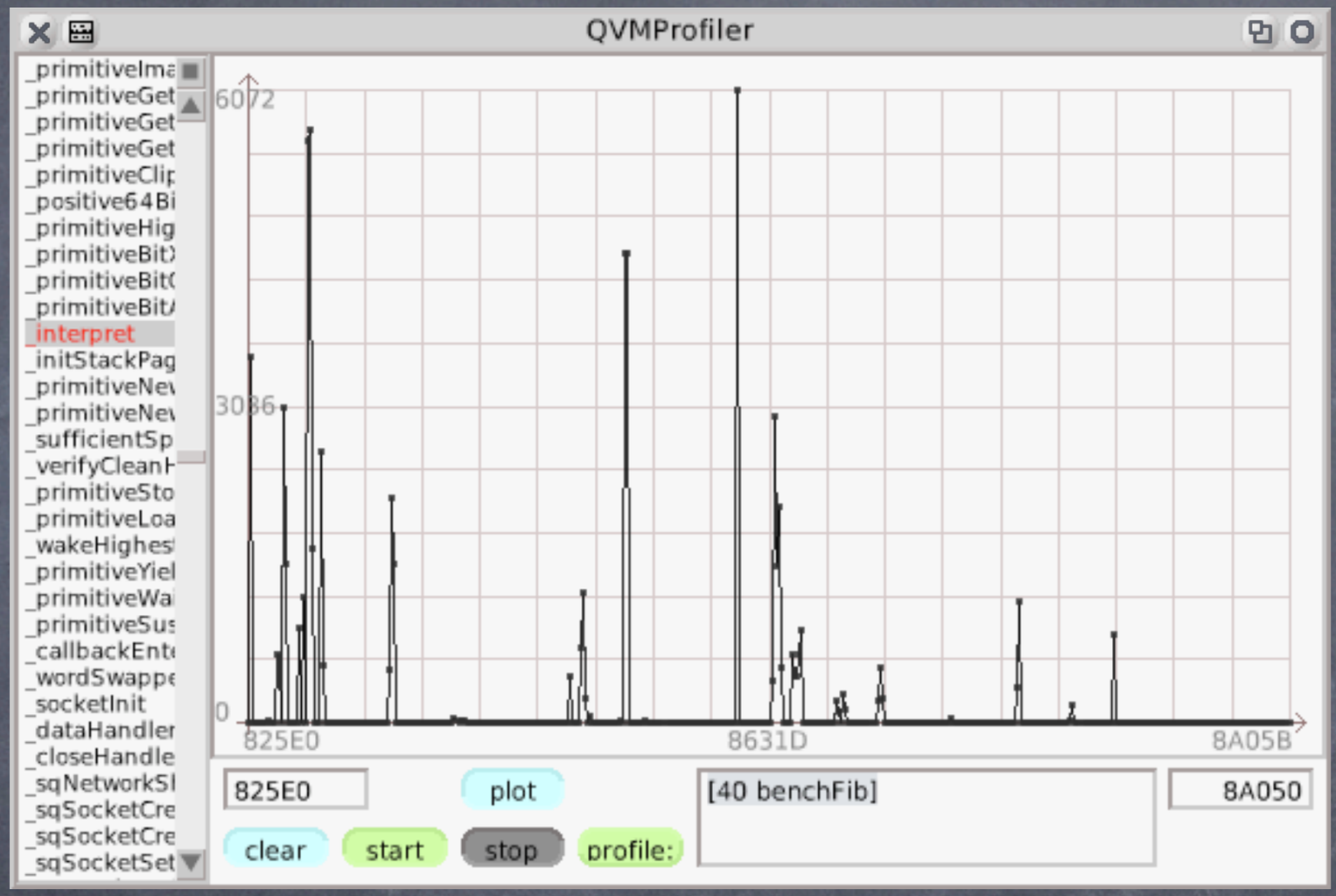
Stack Interpreter

- Nowish
- Detailed blog posts + code real soon now™
- Underwhelming Performance
  ≈ 10% ⇔ 68% faster benchmark performance

  as yet no faster for client experience

  bit of a mystery...

the VM gardener's spade



many a mickle …

… makes a muckle

fast JIT

- April 2009 (for x86)
- a la HPS
  - simple deferred code gen (stack => register)
  - three register (self + 2 args) calling convention
    - most kernel prims all in regs (at: at:put: + * etc)
    - in-line caches: open & closed PICs
- two word object header
- "open" translator
- retain interpreter

The life-cycle of the lesser spotted Inline Cache

- egg -> monomorph -> polymorph -> megamorph
- the egg is one of many instructions laid by the JIT when it "compiles" a bytecode method into a native code method

```
<B0> send #=

movl %rCache,#=
call linkSend1Args
```

The life-cycle of the lesser spotted Inline Cache

- if executed the egg hatches into a larval monomorphic inline cache. Typically only 70% of eggs will hatch

```
#(true false nil 0 0.0 #zero 'zero' $0 #()) select:
    [:each| each = 0]
```

```
movl %rCache,#=   ⇒   movl %rCache,True
call linkSend1Args ⇒   call Object.=.entry
```

The life-cycle of the lesser spotted Inline Cache

```
movl %rCache,True
call Object.=.entry

Object.=.entry:
    mov %rTemp,%rSelf
    and %rTemp,#3
    jnz L1
    mov %rTemp,%rSelf[#ClassOffset]
L1: cmp %rTemp,%rCache
    jnz LCallFixSendFailure
Object.=.noCheckEntry:
    rock and roll
```

The life-cycle of the lesser spotted Inline Cache

- if the monomorph encounters another kind it changes into a nymph polymorphic cache with 2 cases. Only 10% of monomorphs will metamorphose into Closed PICs

movl %rCache,True ⇒ movl %rCache,True

call Object.=.entry ⇒ call aClosedPIC.=.entry

The life-cycle of the lesser spotted Inline Cache

```
aClosedPIC.=.entry:
    mov %rTemp,%rSelf
    and %rTemp,#3
    jnz L1
    mov %rTemp,%rSelf[#ClassOffset]
L1: cmp %rTemp,%rCache
    jz Object.=.noCheckEntry
    cmp %rCache, #False
    jz Object.=.noCheckEntry
    jmp extendClosedPIC
    ....
```

The life-cycle of the lesser spotted Inline Cache

```
aClosedPIC.=.entry:
    mov %rTemp,%rSelf
    and %rTemp,#3
    jnz L1
    mov %rTemp,%rSelf[#ClassOffset]
L1: cmp %rTemp,%rCache
    jz Object.=.noCheckEntry
    cmp %rCache, #False
    jz Object.=.noCheckEntry
    cmp %rCache, #UndefinedObject
    jz Object.=.noCheckEntry
    cmp %rCache, #SmallInteger
    jz SmallInteger.=.noCheckEntry
    jmp extendClosedPIC
```

The life-cycle of the lesser spotted Inline Cache

- if the nymph polymorph has a rich enough life and encounters more than (say) 8 classes of self then it blossoms into a magnificent Open PIC

movl %rCache,True ⇒ movl %rCache,True

call aClosedPIC.=.entry ⇒ call anOpenPIC.=.entry

The life-cycle of the lesser spotted Inline Cache

- An adult Open PIC is adept at probing the first-level method lookup cache to find the target method for each self

- Since the Open PIC started life for a single selector it knows the constant value of its slector and its selector's hash

- Only 1% of monomorphs will complete the arduous journey to Open PIC

The epiphenomena of the lesser spotted Inline Cache

- in the steady state there is no code modification
- monomorphs and closed PICs eliminate indirect branches, allowing the processor's prefetch logic to gorge itself on instructions beyond each branch
- eggs, monomorphs and polymorphs record concrete type information that an be harvested by an adaptive optimizer that speculatively inlines target methods

two-word object header

e.g.

class...	...table...	...index	flags etc
identity...	...hash...	...field	slot size

- "common" 32-bit/64-bit object header
- classTableIndex is lookup key in in-line caches & first-level method lookup caches. GC doesn't move them => simplified inline cache mgmt
- index sparse class table with classTableIndex only for class hierarchy search & class primitive
- no table access to instantiate known classes
- A class's id hash is its classTableIndex => no lookup to classTableIndex for **new:**

"Open" fast JIT

- bytecode set translation via a table of functions not a switch statement
- object model is an ADT
- should be able to configure JIT for different bytecode sets, different GCs, object representations
- hence Croquet, Squeak and Newspeak and...?

quick ~~DT~~ JIT

- target: good floating-point performance
- AOStA again... Adaptive Optimization =>
  SIStA Speculative Inlining
- reify PIC state
- count conditional branches
  6x less frequent than sends
  taken & untaken counts => basic block frequency
- image-level optimizer

quick ~~DC~~ JIT

- image-level optimizer (many benefits)
  - VM extended with go-faster no-check primitive bytecodes, e.g.
    add known non-overflowing SmallIntegers
    in-bounds pointer at: known SmallInteger index
- could marry well with LLVM

quick ~~DS~~ JIT

model for good floating-point performance:

- OptimizedContext has two stacks object & byte data
- VM extended with unboxed float bytecodes, e.g. bdsDoubleAt:putProductOfBdsDblAt:andBdsDblAt: pushBoxDbsDoubleAt:
- code gen maps byte data stack to floating-point regs

quick ~~DX~~ JIT

- 2010, 2011?
- Sooner if you join the Cognoscenti...