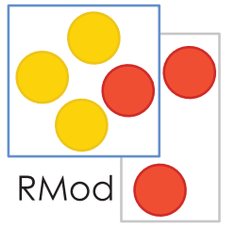
Towards Self–aware Virtual Machines

Camillo Bruni
2014-05-16

Supervisor: Stéphane Ducasse
Co-Supervisor: Marcus Denker

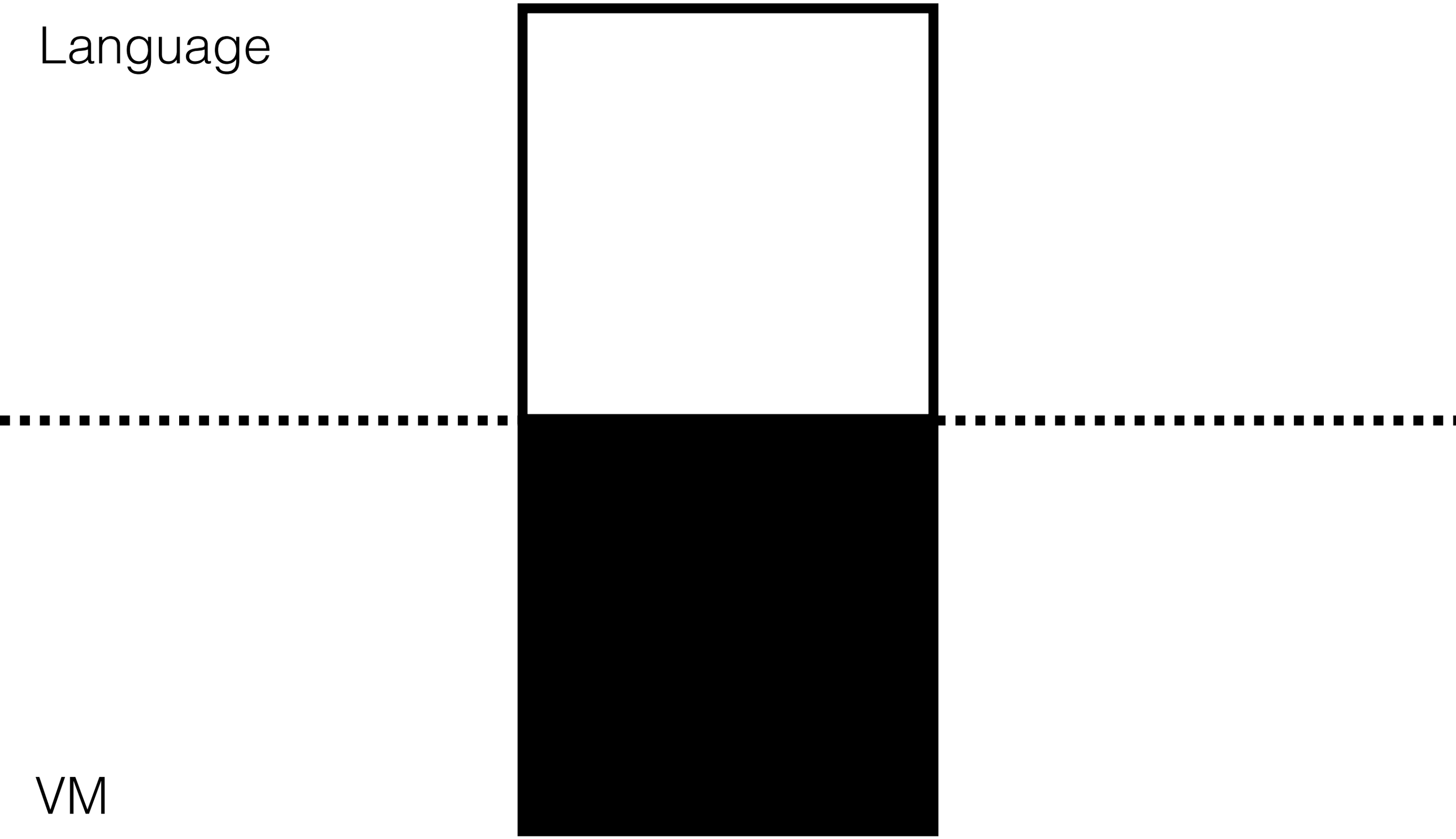


Self–aware Virtual Machines

Language

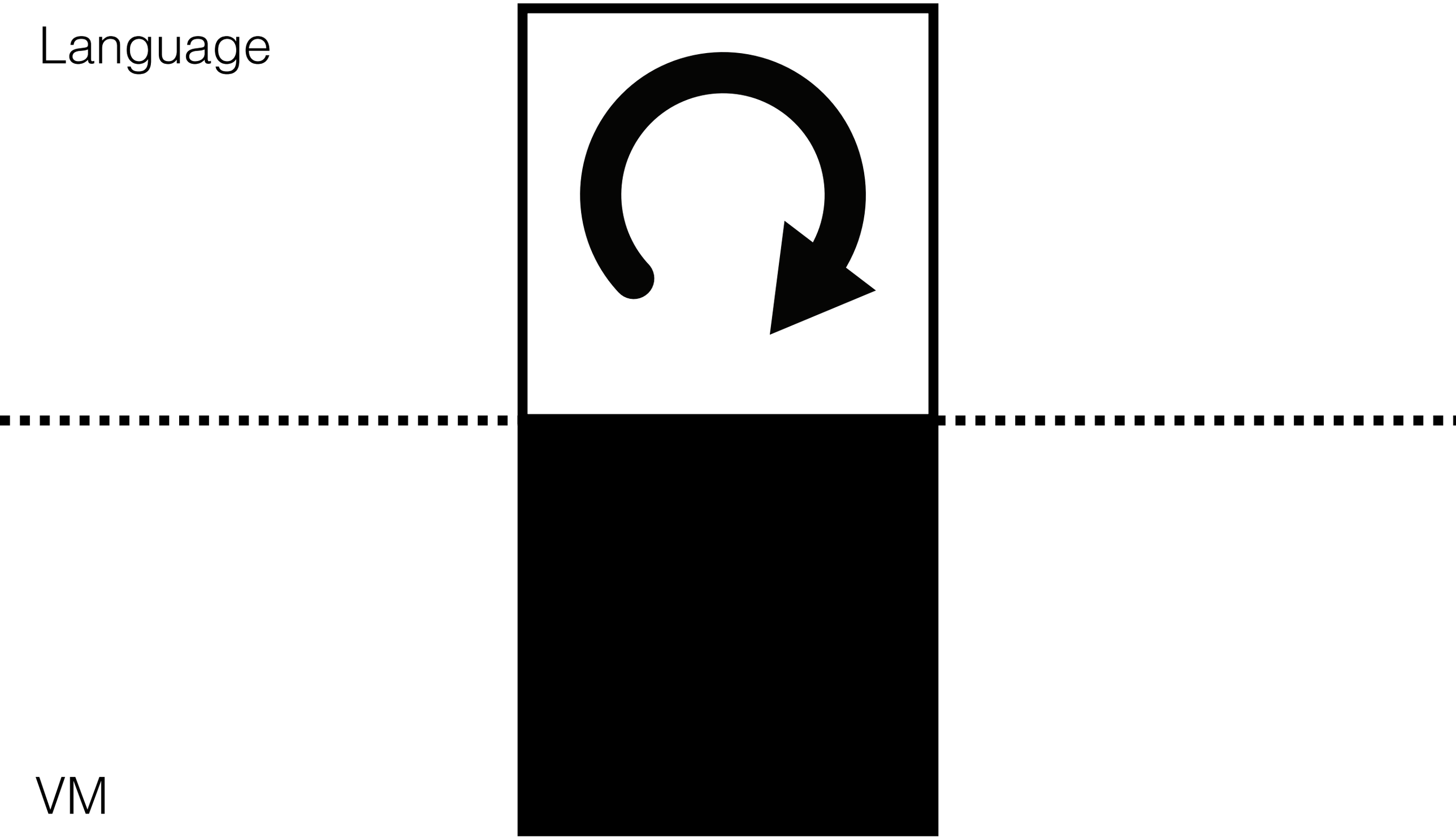
VM

Language



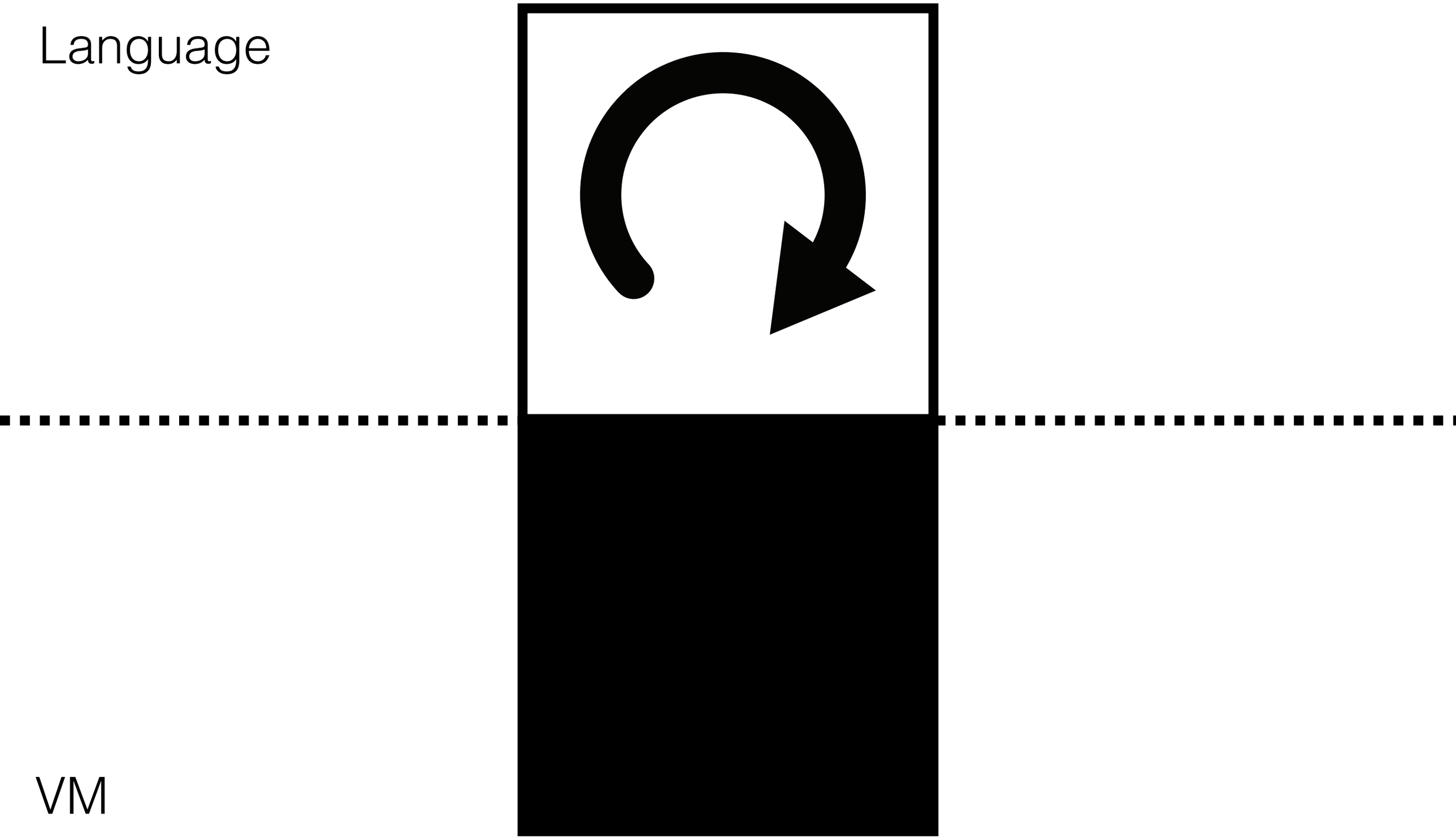
VM

Language



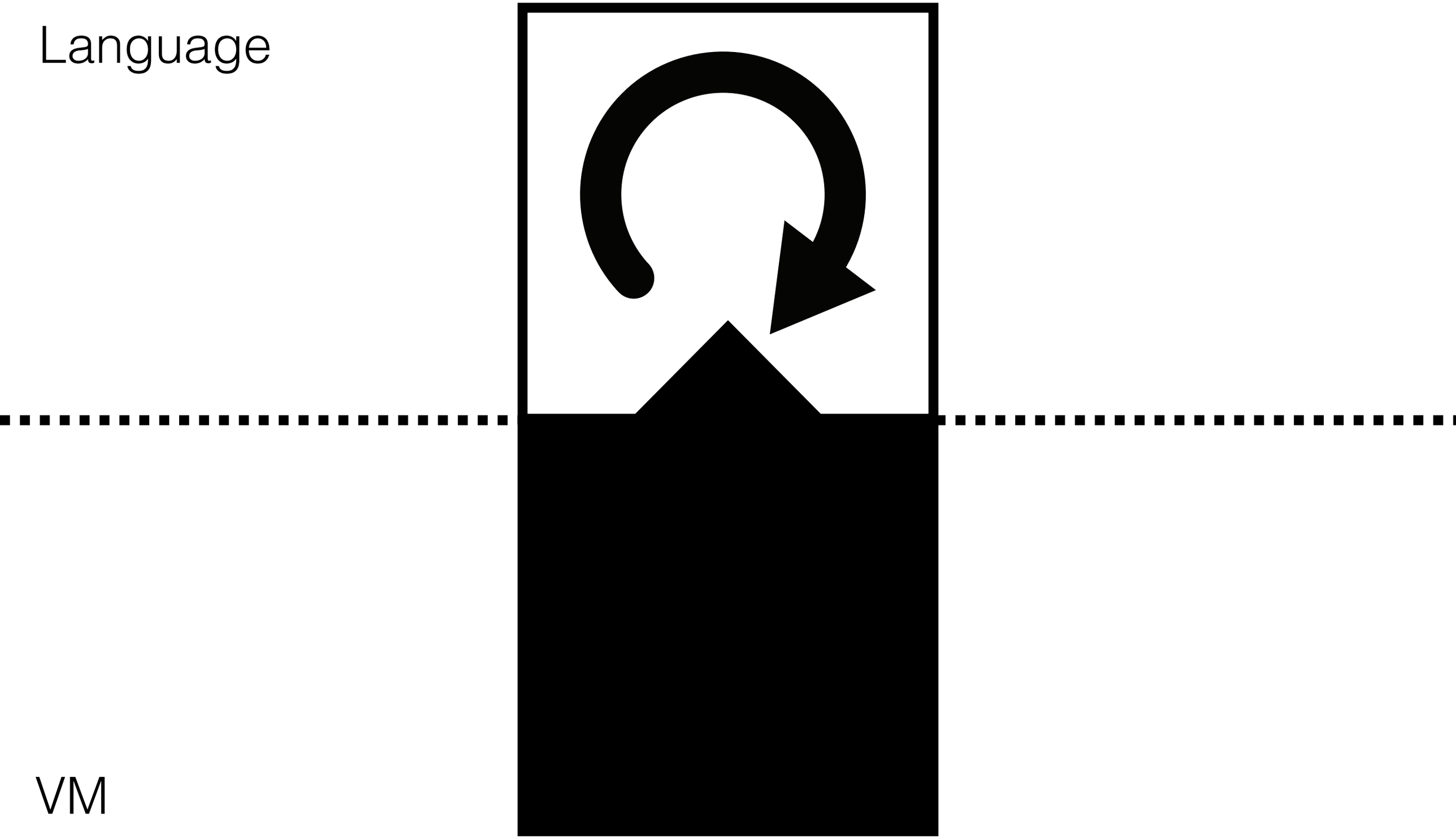
VM

Language



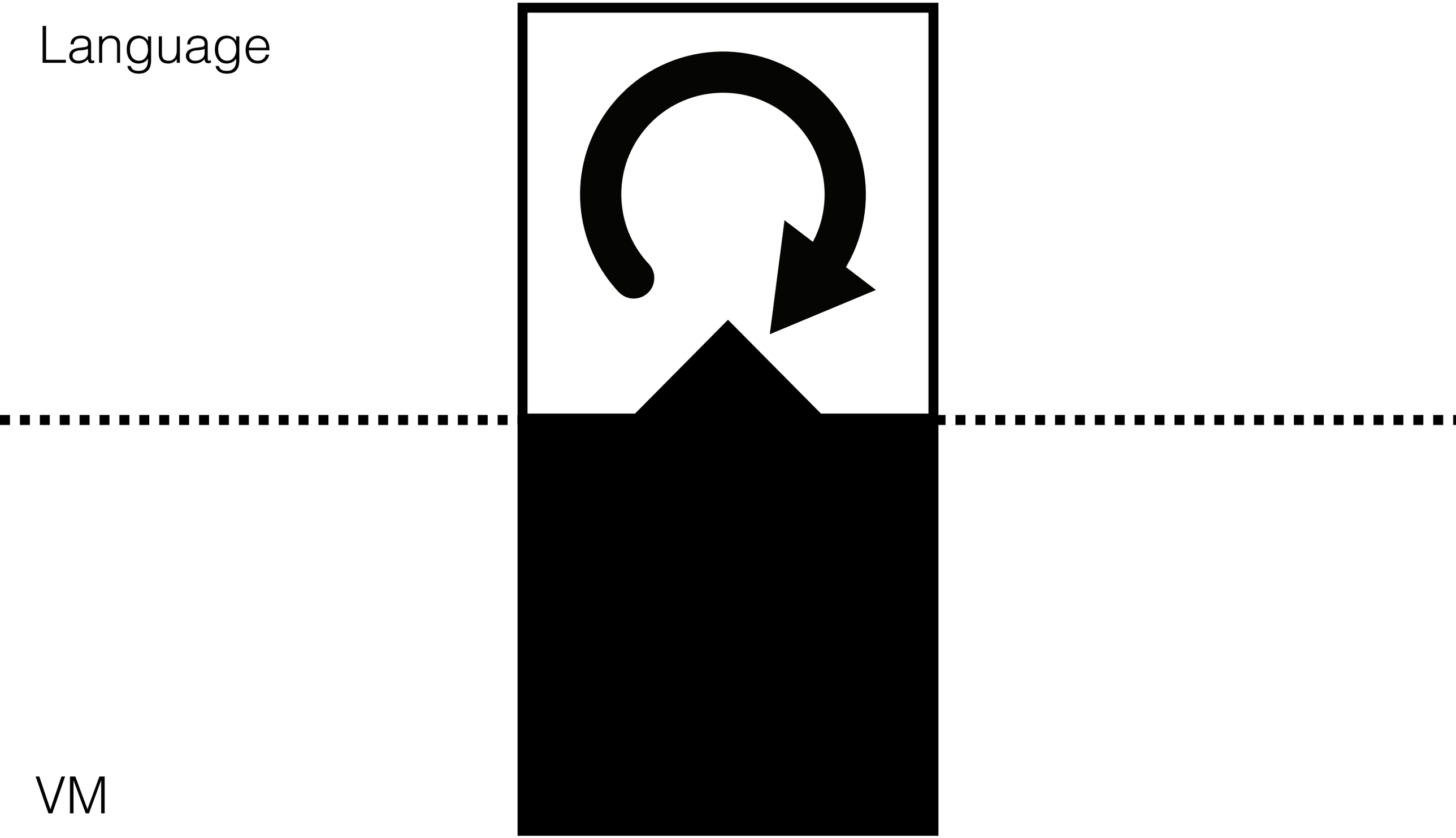
VM

Language



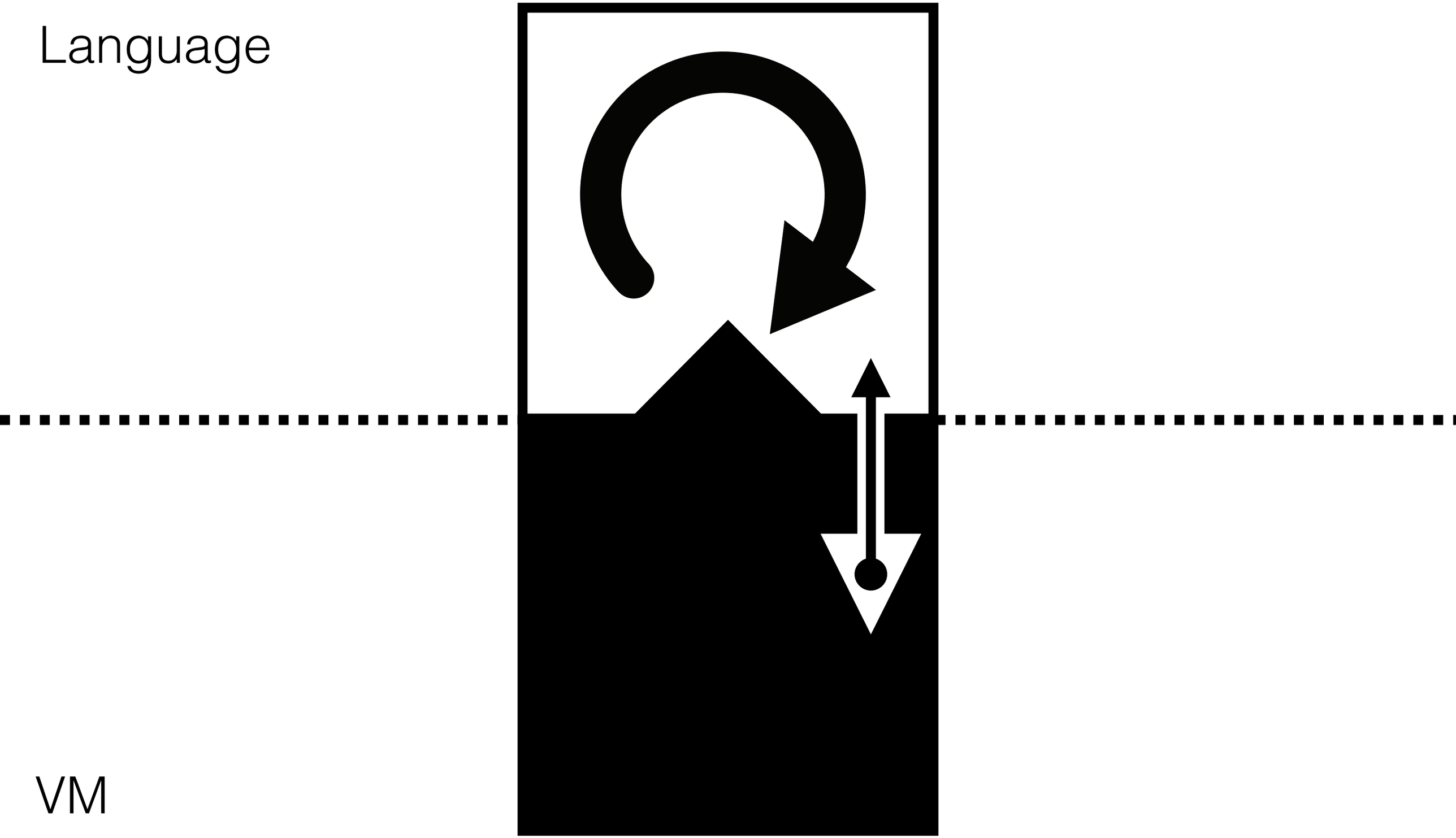
VM

Language



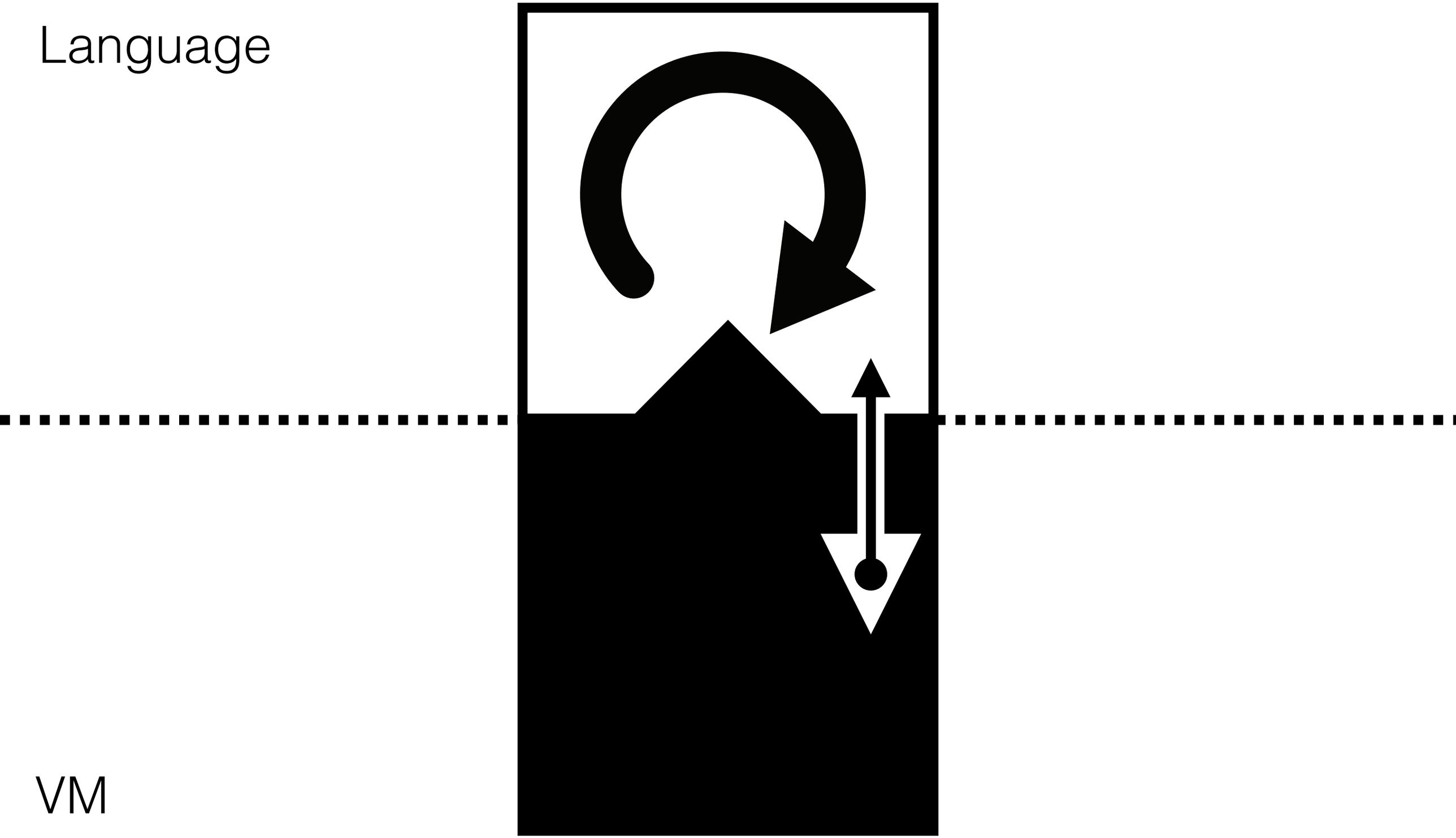
VM

Language



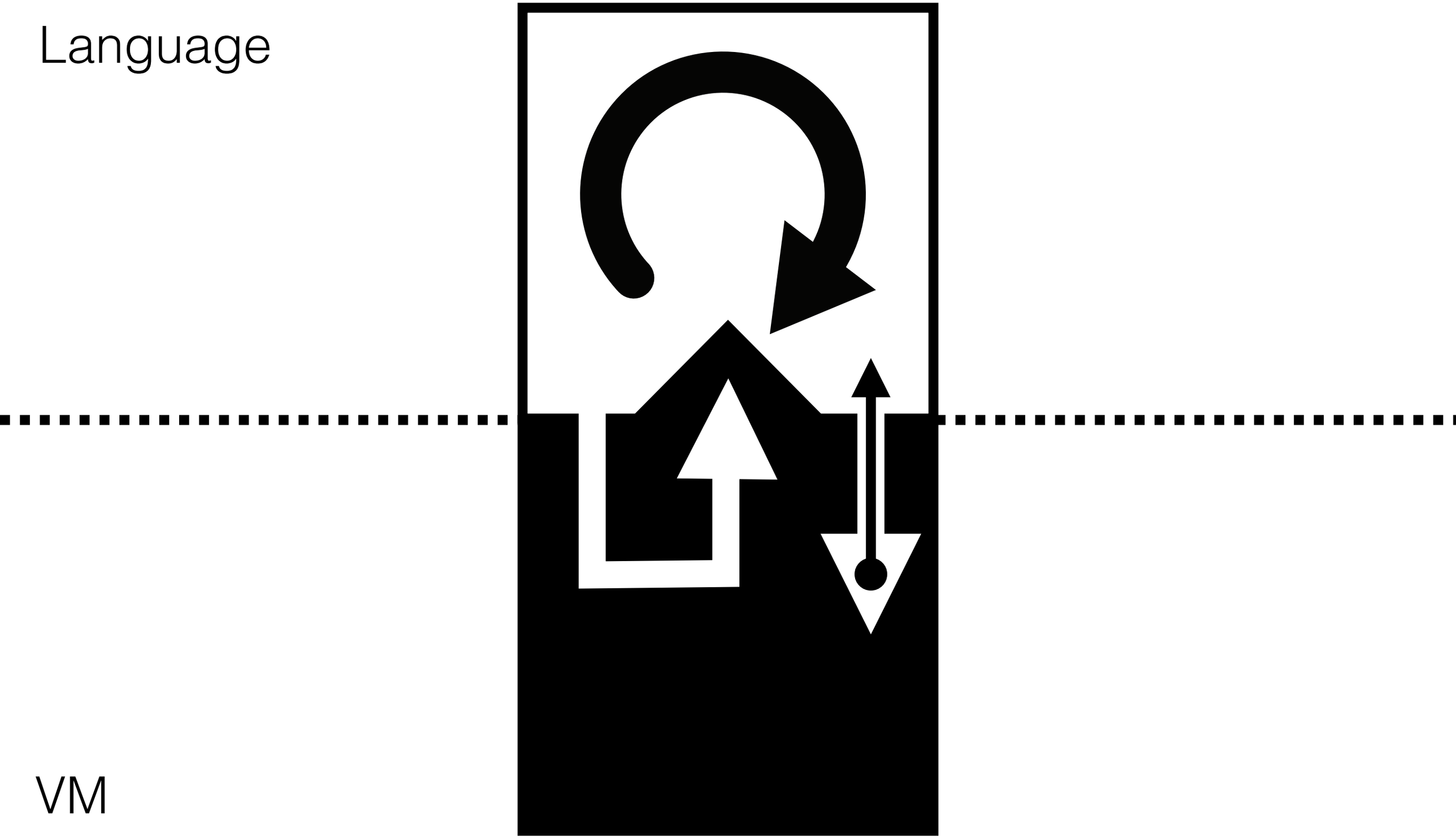
VM

Language



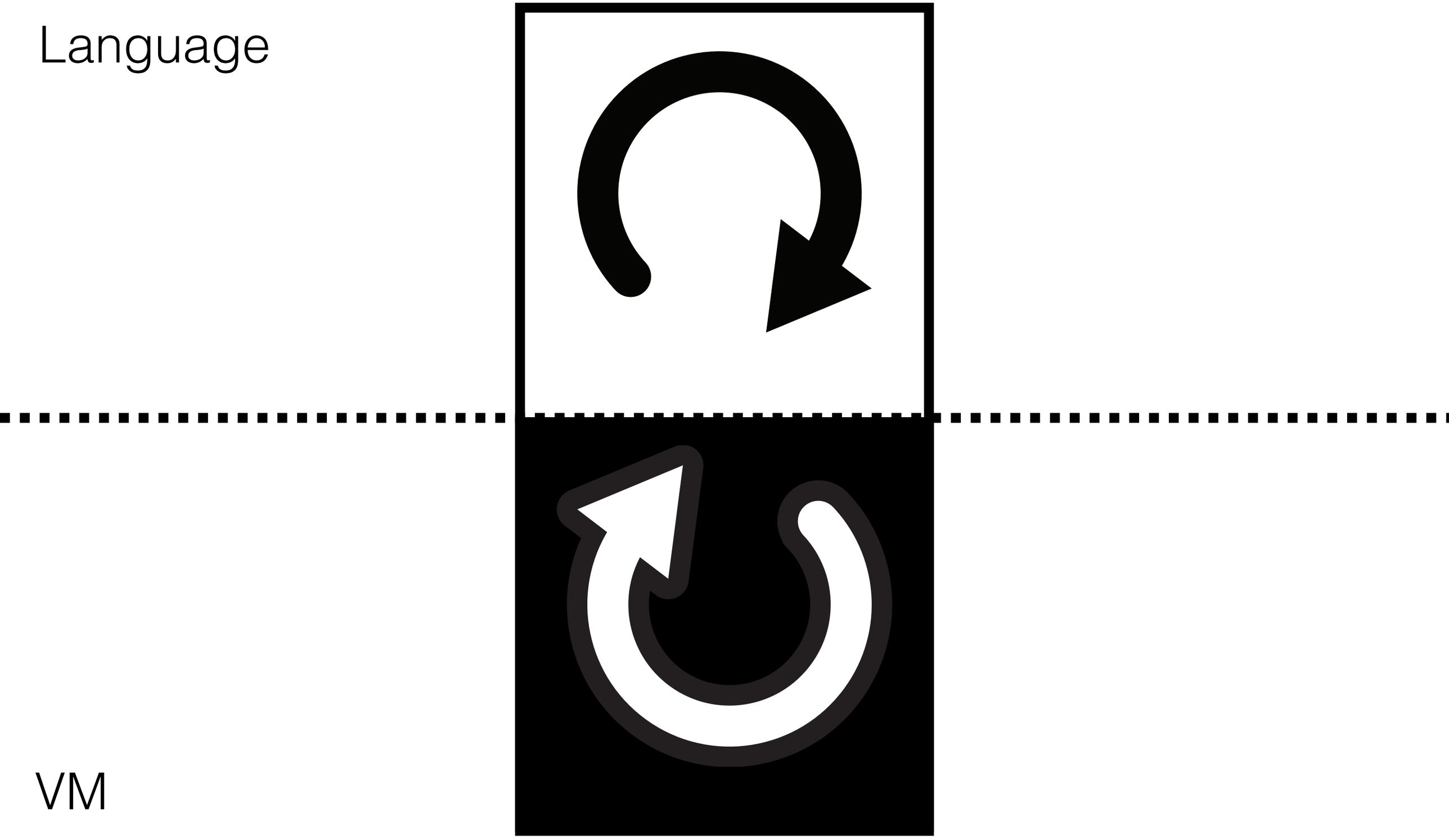
VM

Language

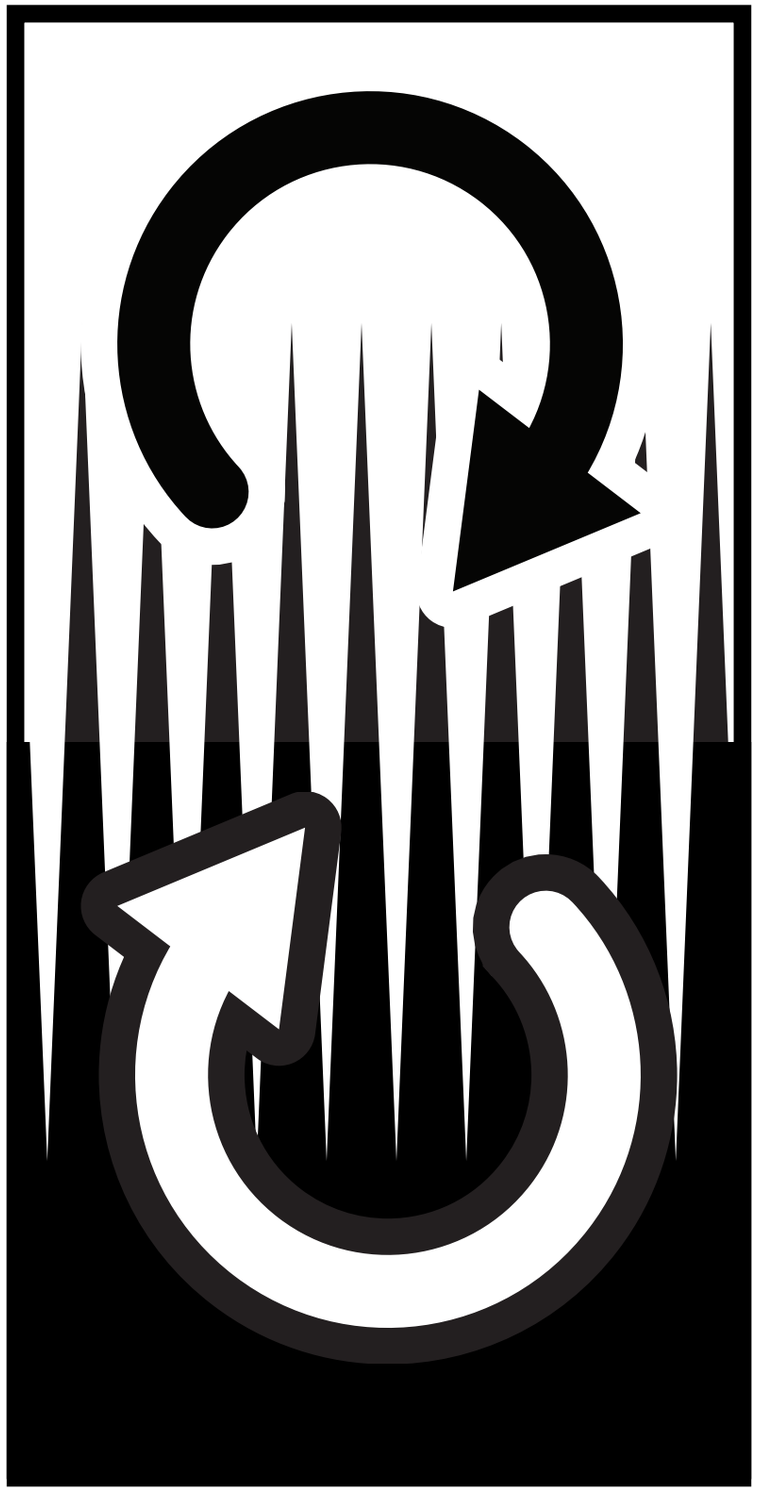


VM

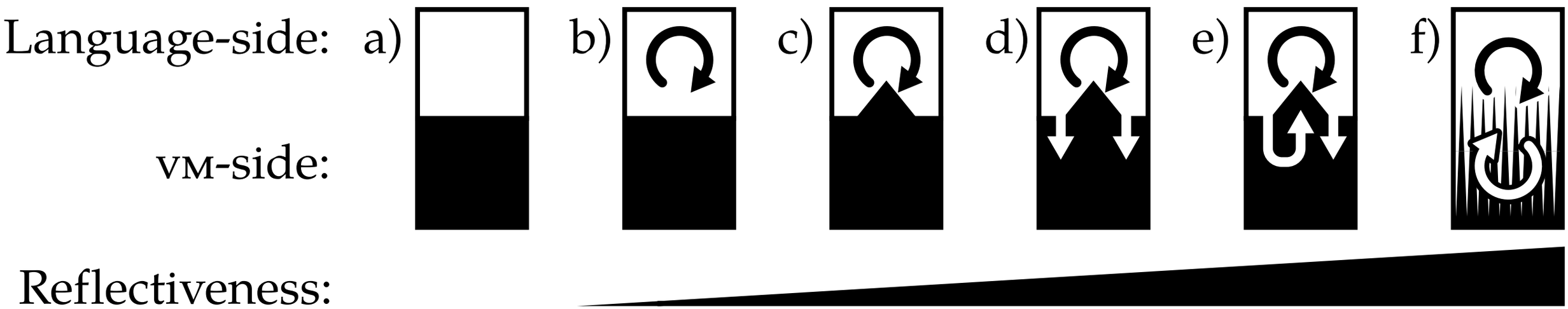
Language



VM



Towards Self–awareness



{Intercession, Introspection}

$\times$

{Structure, Behavior}

$\times$

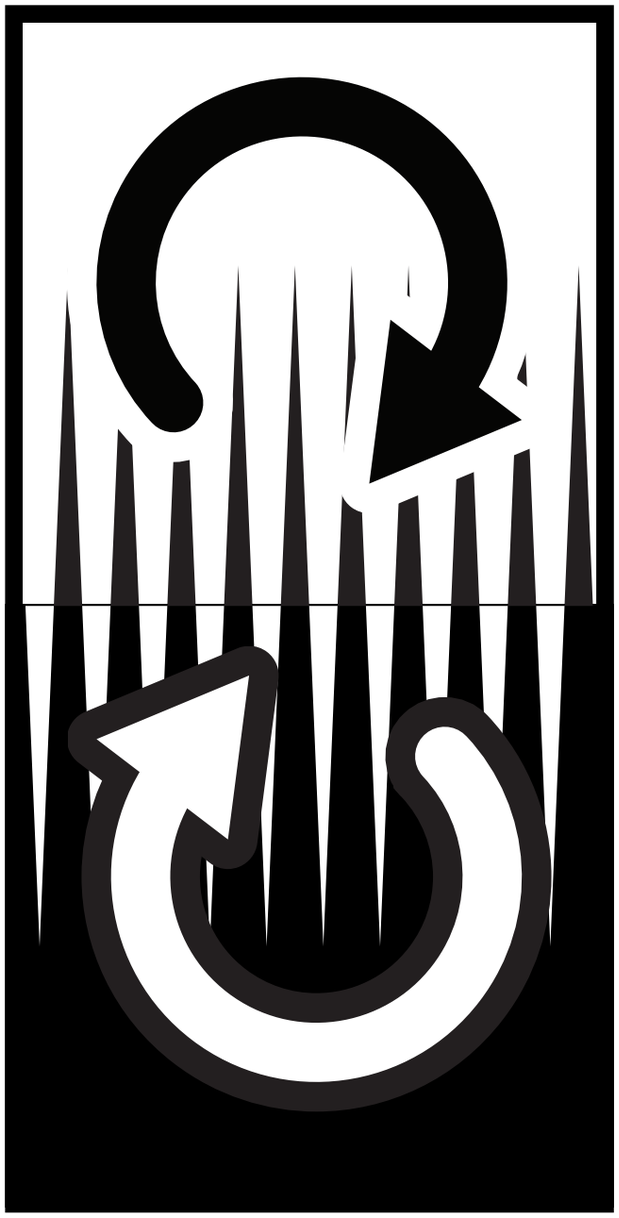
{VM, Language}

“Dynamic Reflection”

×

{VM, Language}

Vision

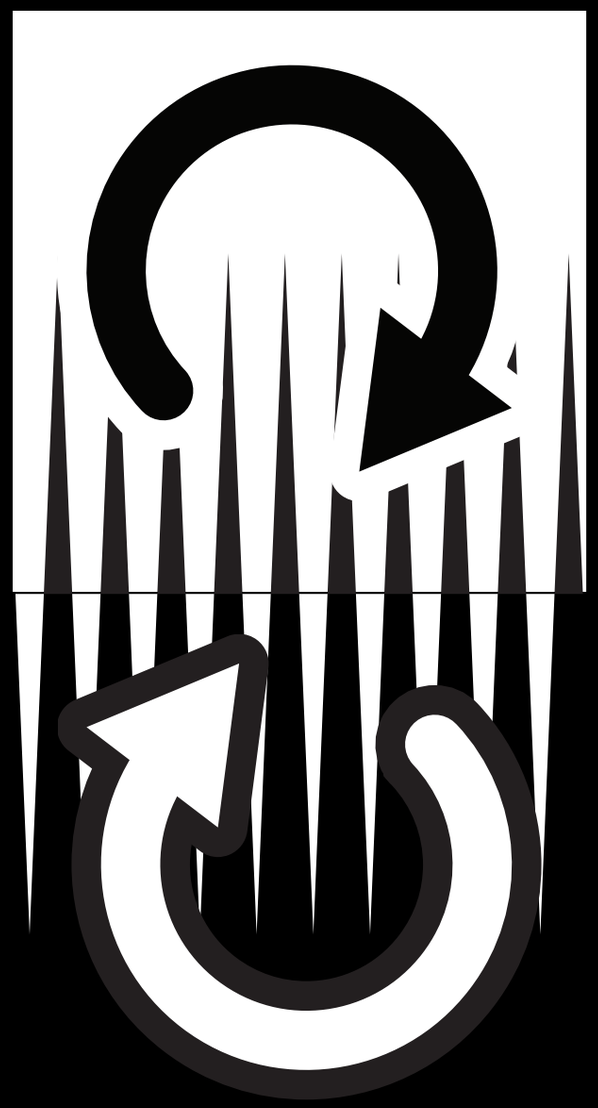


Develop the VM at Language–side

Debug the VM at Language–side

Modify the VM at Language–side

Reality



Performance

Complexity

Compatibility

Basic Requirements?

Native Code Activation

Circumventing the VM-separation

Language

VM

Language

bottom^{up}

VM

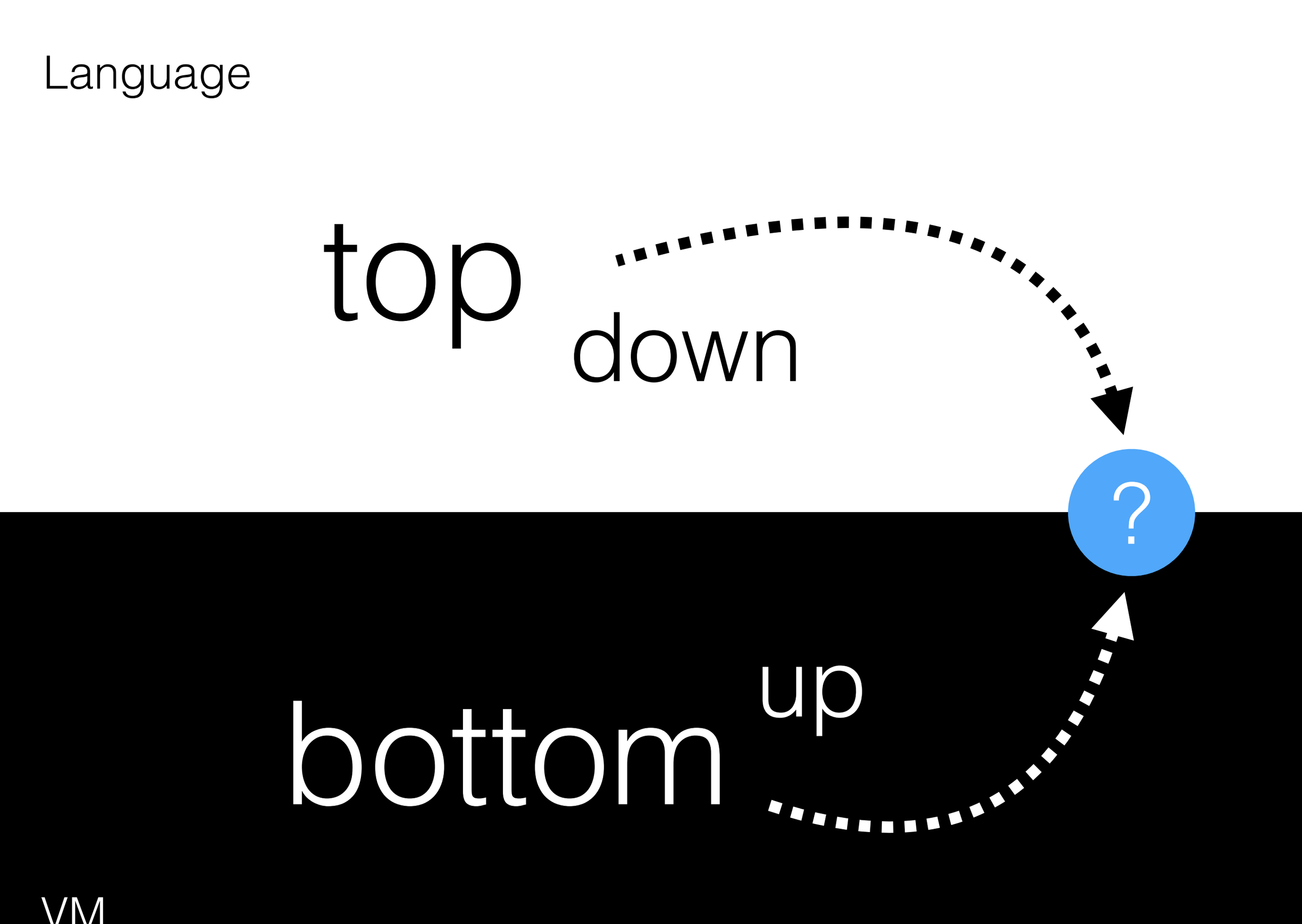
Language

top down

bottom up

VM

Language



VM

Thesis Statements

Dynamic Native Code Activation

Dynamic Intercession at VM–level

Incremental Extension

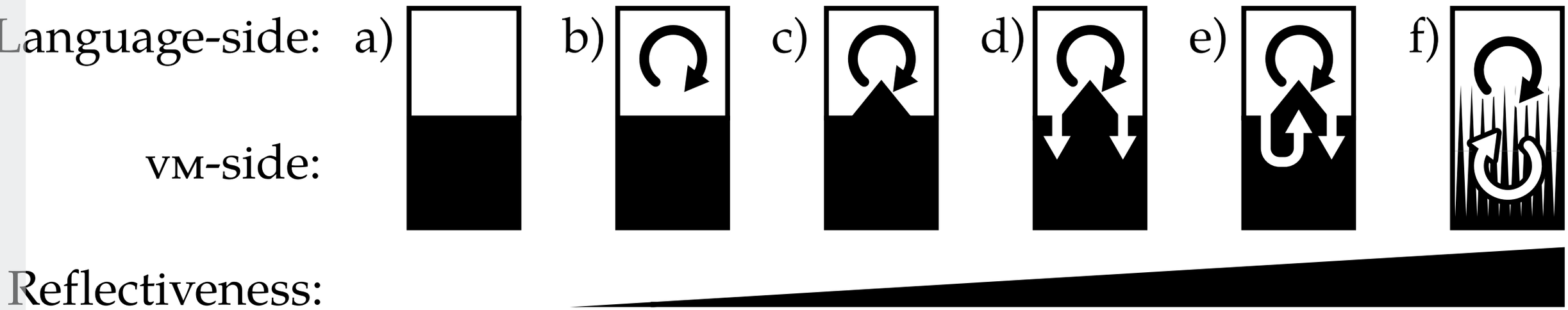
Vision

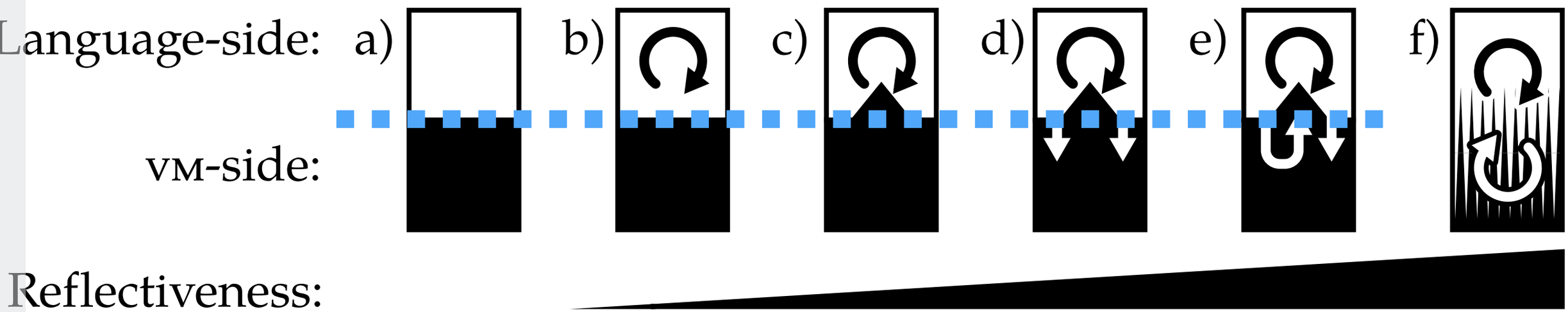
Related Work Analysis

Solution

Validation

Conclusion & Future Work


Language-side:
a)
b)
c)
d)
e)
f)
vm-side:
Reflectiveness:


Language-side:
a)
b)
c)
d)
e)
f)
vm-side:
Reflectiveness:

High-level Low-level Programming

in the JIKES RVM Memory Management Toolkit

D. Frampton et al. *Demystifying magic: high-level low-level programming.* VEE '09

Compile–time Transformation

```
@RawStorage(lengthInWords=true, length=1)
@Unboxed
class Address {
  ...
  @Intrinsic("org.vmmagic.unboxed.loadByte")
  native byte loadByte();
  ...
}
```

SQUEAK VM

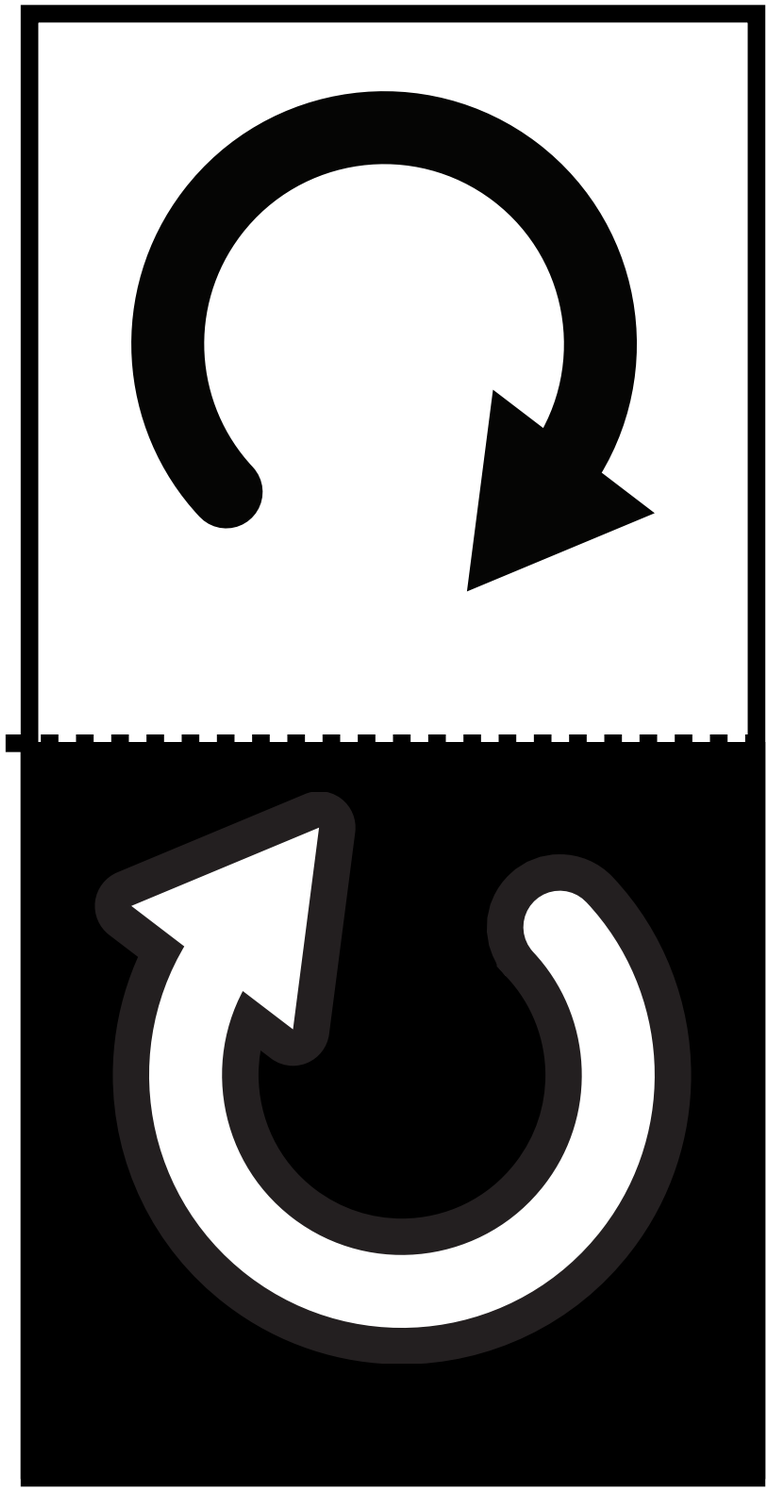
```
instructionPointerAddress
  <returnTypeC: #usqInt>
  ^ self
    cCode: [ ... ]
    inSmalltalk: [
      simulator
          readWrite: #instructionPointer
          in: self ]
```

Ungar et al. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself.* OOPSLA '97

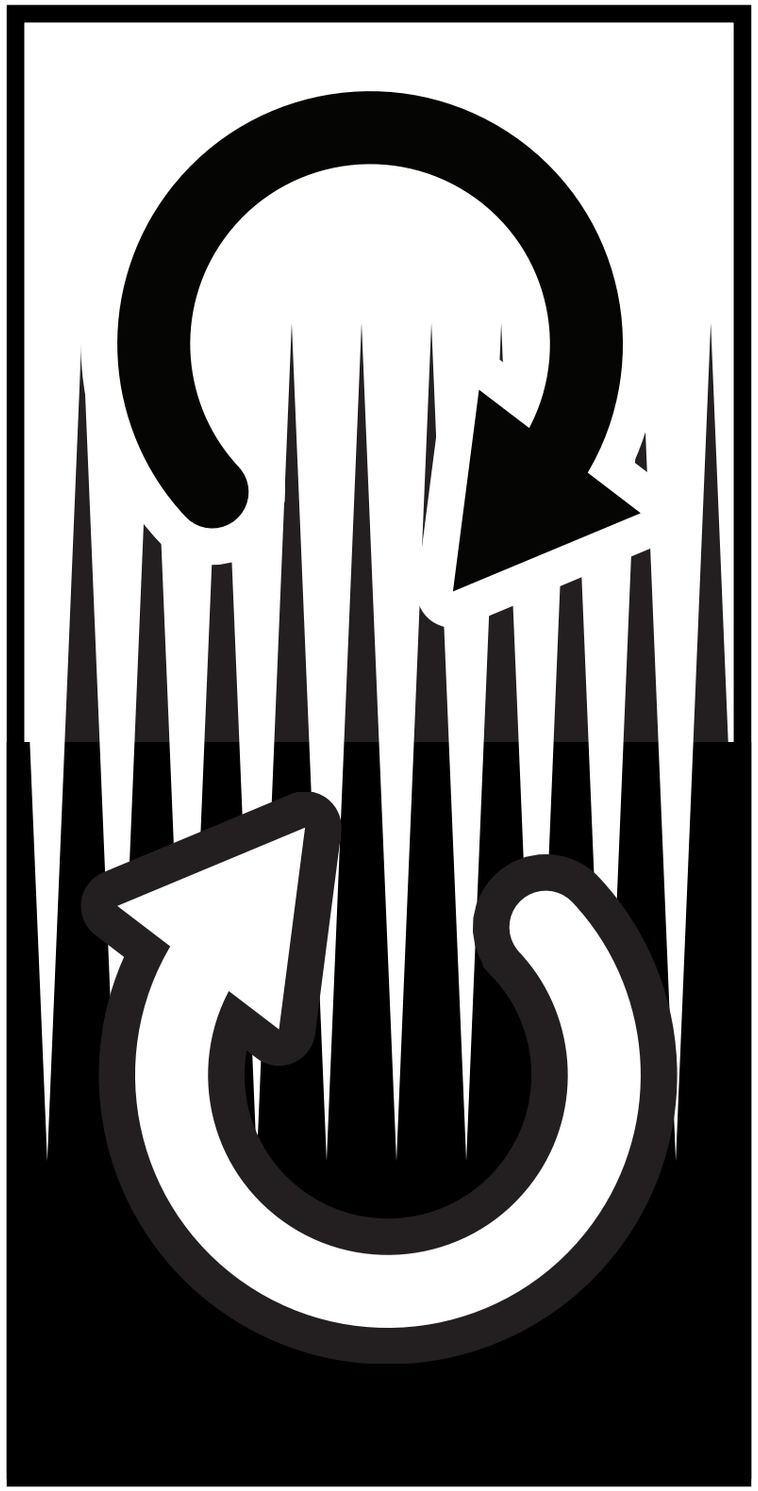
High-level Low-Level Programming Summary

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–

Language



VM



Pinocchio Smalltalk Language Runtime

First–class Interpreters

A new look at the Tower of Interpreters

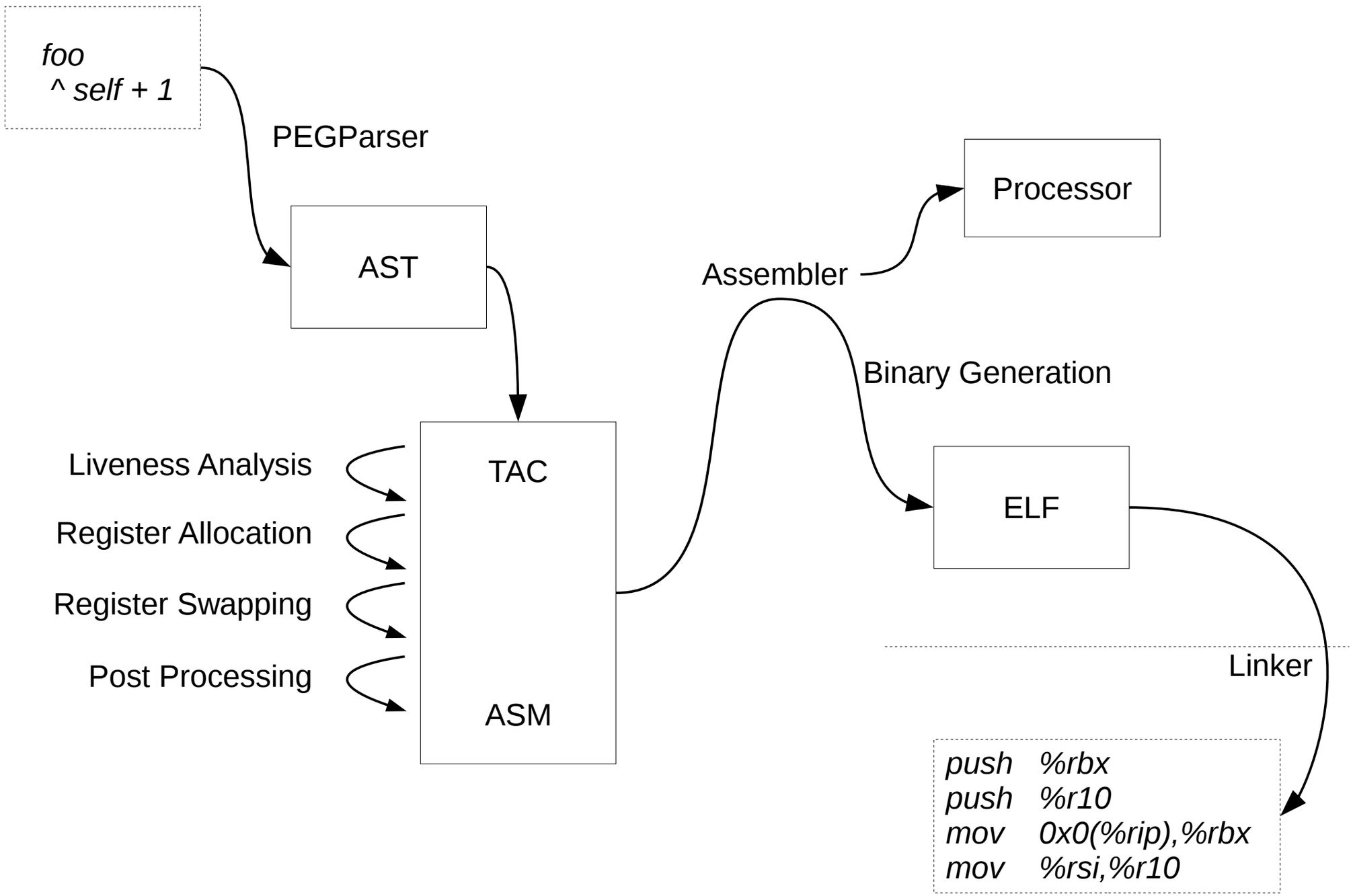
T. Verwaest, **C. Bruni,** D. Gurtner, A.Lienhard and O. Nierstrasz
OOPSLA '10

First–class Slots and Object Layouts

Reified Instance Variables
Bridging the Gap to Raw Memory

T. Verwaest, **C. Bruni,** M. Lungu and O. Nierstrasz
OOPSLA '11

Native Compiler



Pinocchio Summary

Native Compiler Complexity is Overrated

Reification for Advanced Applications

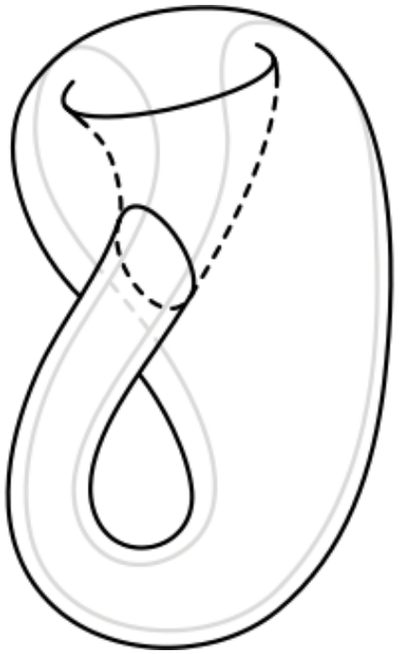
Complex One-time Operations are not Expensive

Pinocchio Summary

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
Pinocchio	**–**	**+**	**~**

KLEIN Metacircular VM

Lessons Learned from the SELF VM



Ungar et al. *Constructing a metacircular Virtual machine in an exploratory programming environment.* OOPSLA '05

Goals

Code–reuse		Metacircularity
Fast Compilation Cycle	⟷	Live VM–modification
Interactive Debugging		Mirrors

Goals

Code–reuse

Fast Compilation Cycle

Interactive Debugging

Metacircularity

Live VM–modification

Mirrors

Klein VM Summary

Self–aware System

Full Intercession Capabilities

Full Introspection Capabilities

VM–level Intercession Only Used for Debugging

Klein VM Summary

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
Pinocchio	–	+	~
Klein	**–**	**+**	**~**

Context Related Work

High-level Low-level Programming Used for Novel VM Development

Self–modification not Used

Substantial Changes Required at VM-level

Thesis Goal

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
Pinocchio	–	+	~
Klein	–	+	~
	+	+	+

Vision
Related Work Analysis
Solution
Validation
Conclusion & Future Work

Starting Point

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	**+**	–	–
PINOCCHIO	**–**	+	~
KLEIN	–	+	~
	?	**?**	**?**

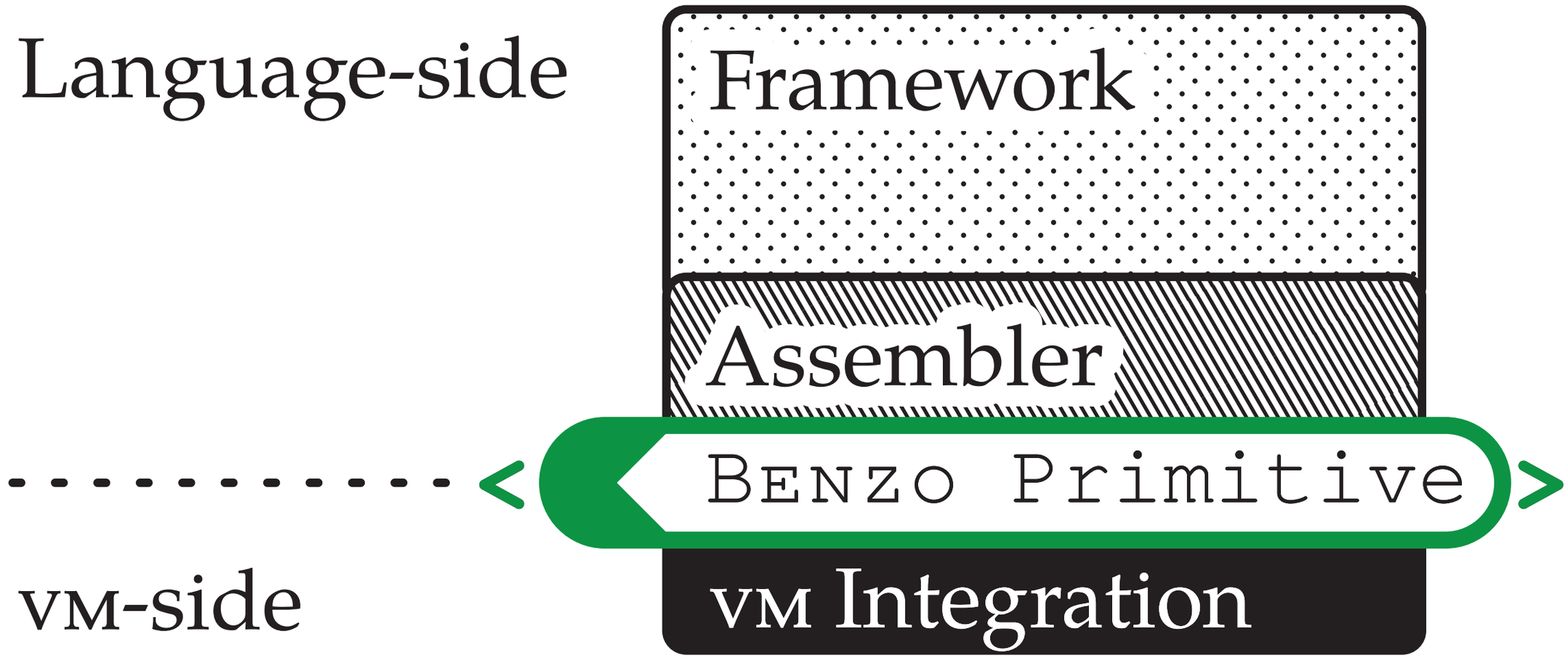
Starting Point

	Incremental	Dynamic Native Code	VM–level Intercession	
HL LL Programming	**+**	–	–	
PINOCCHIO	**–**	+	~	based on PHARO
KLEIN	–	+	~	
	?	**?**	**?**	

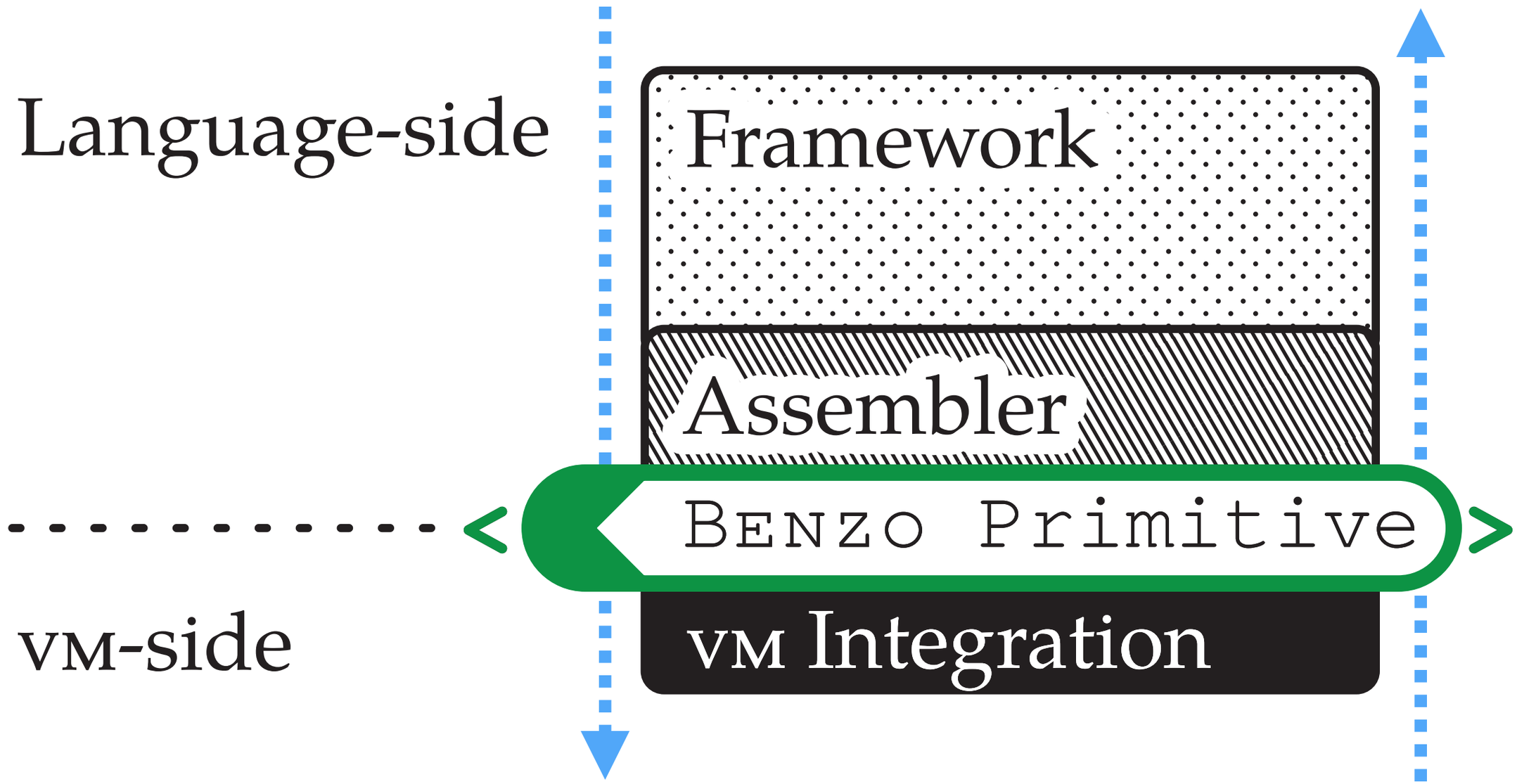
BENZO

High-level Low-level Programming in PHARO

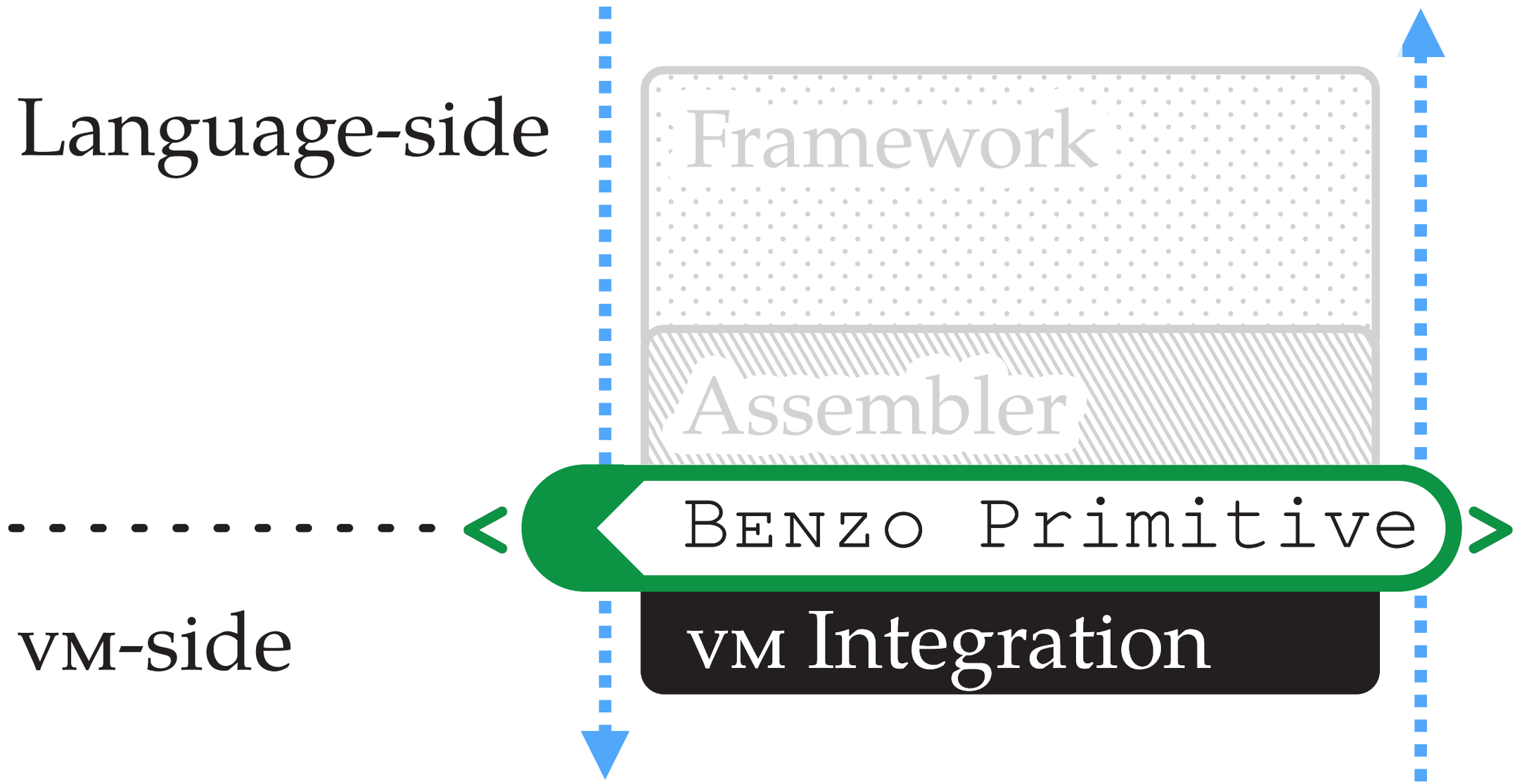
General Design



General Design



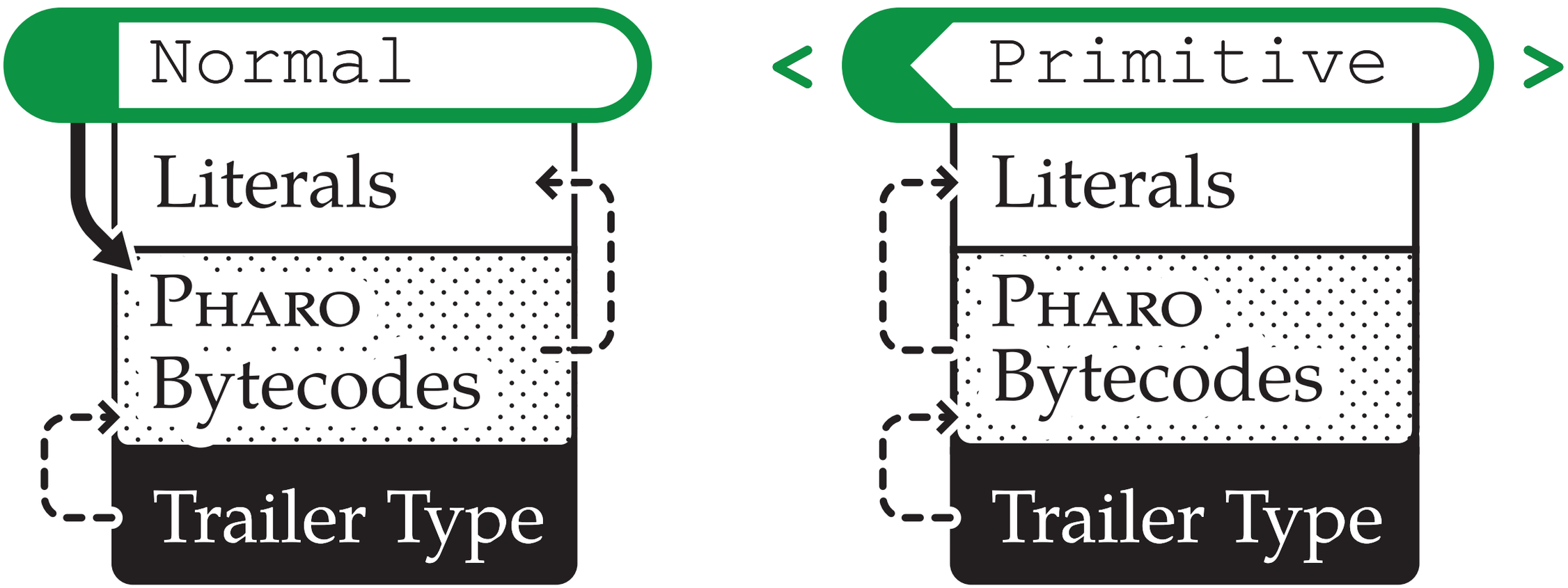
General Design



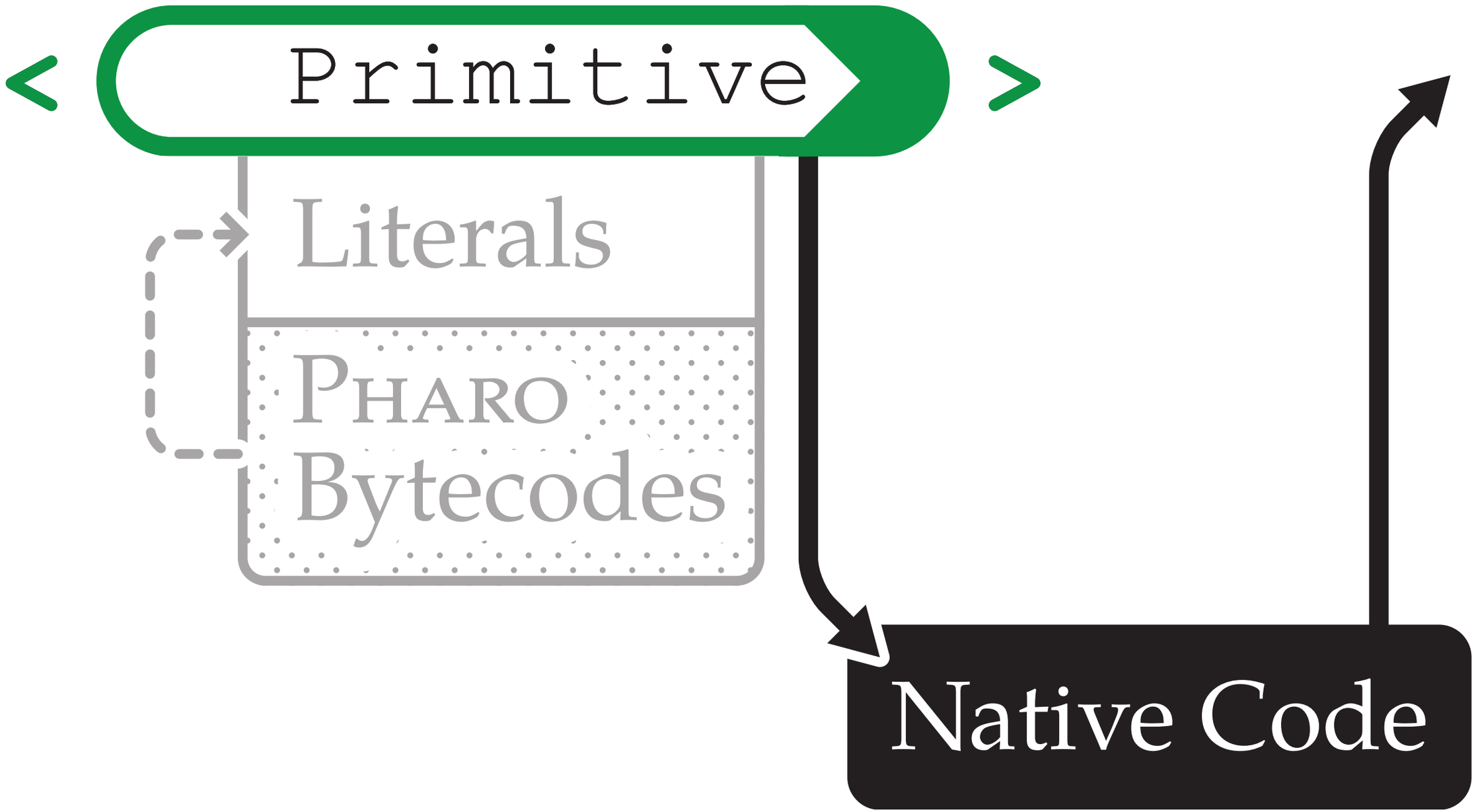
Part I: Native Code Activation

Part II: Native Code Generation

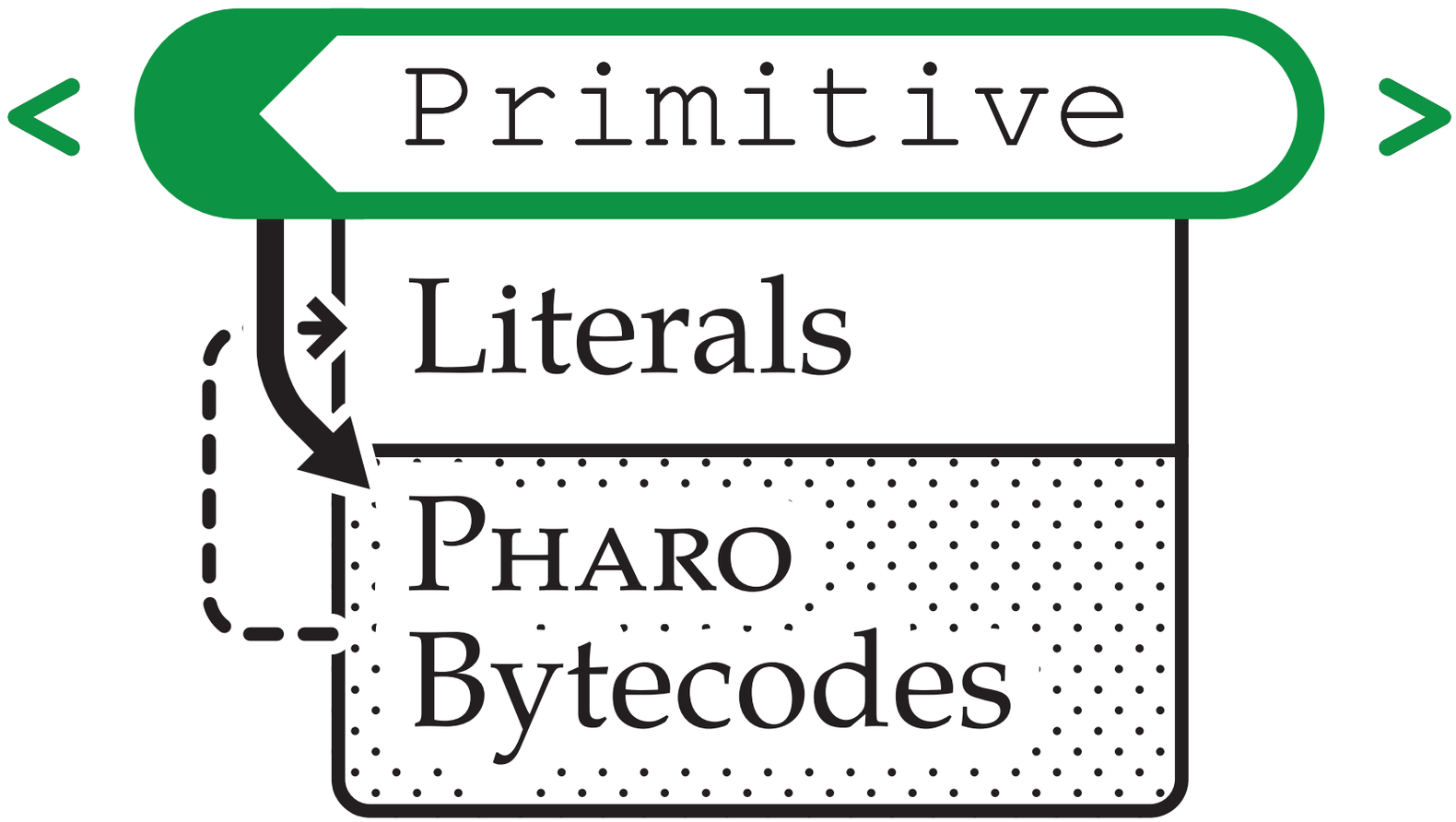
PHARO Compiled Method



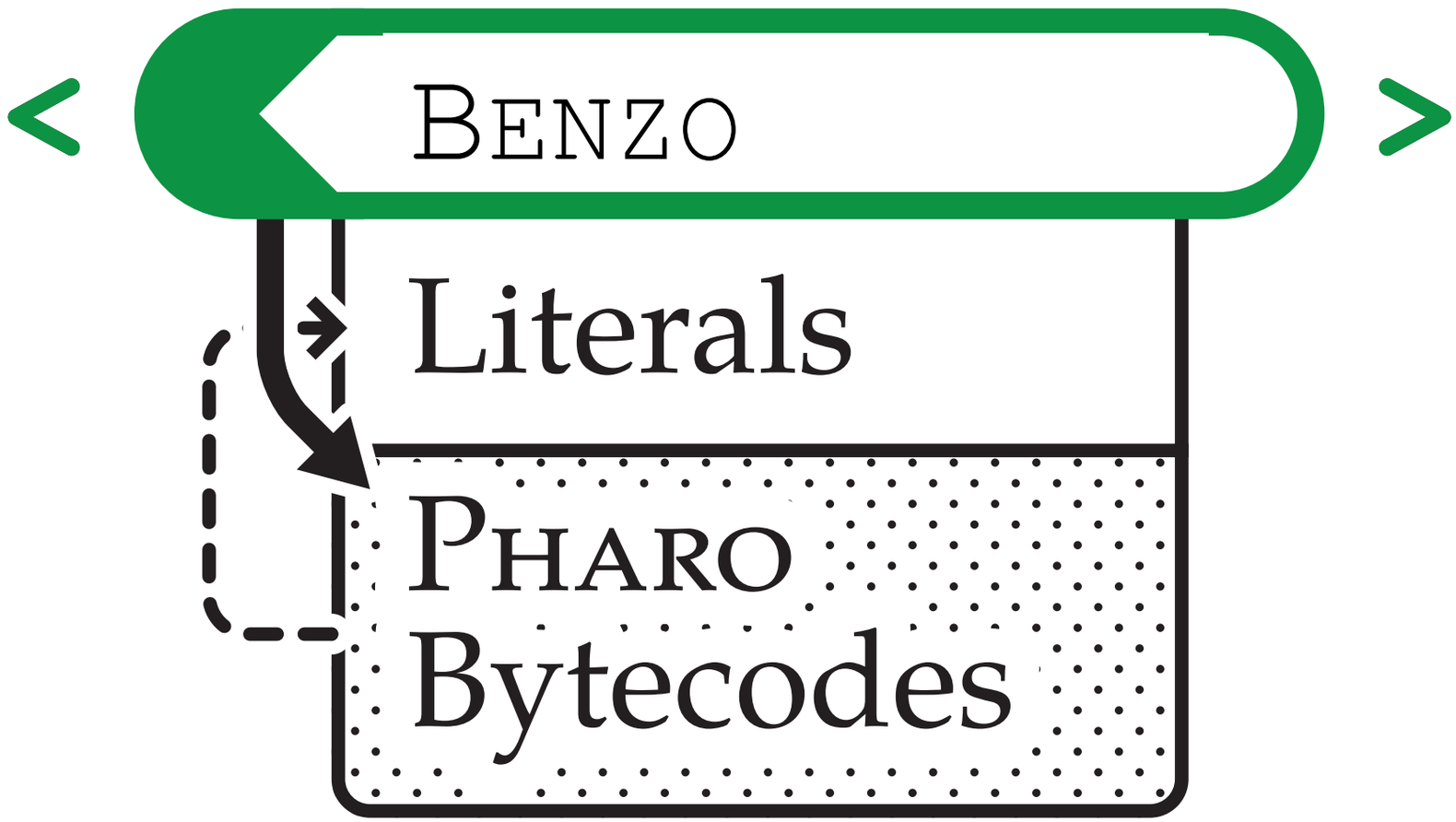
Primitive Activation



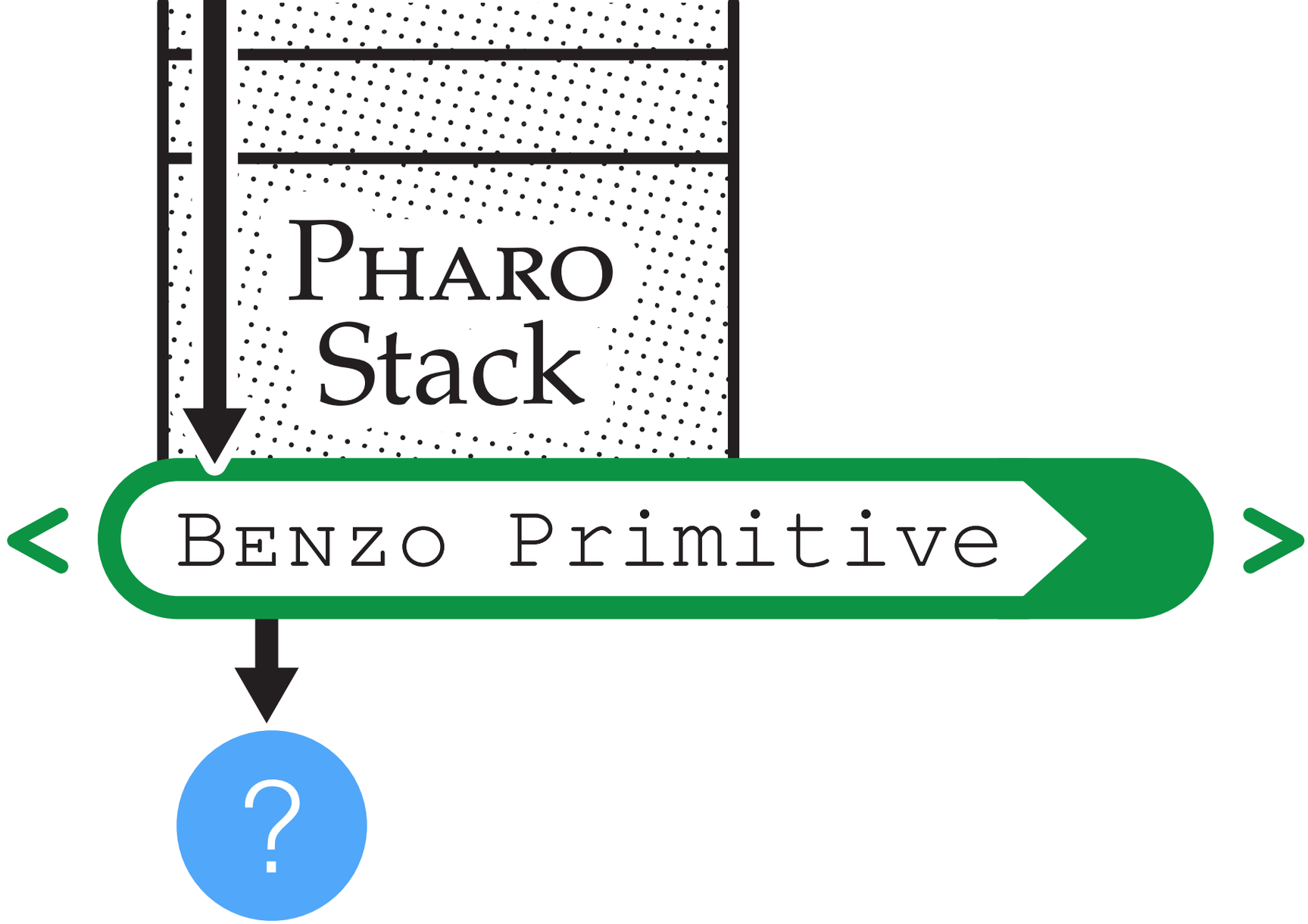
Primitive Fallback



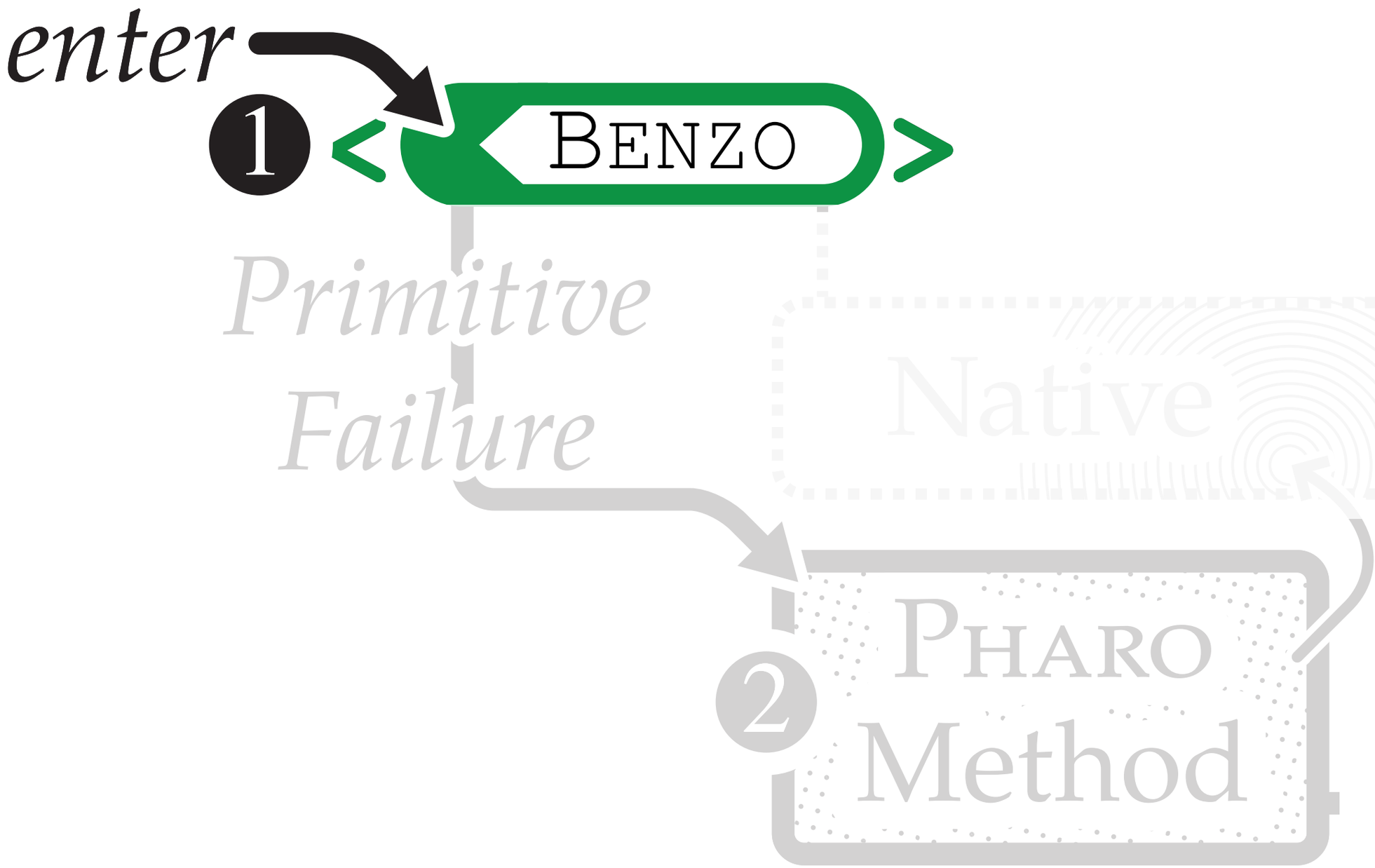
Primitive Fallback



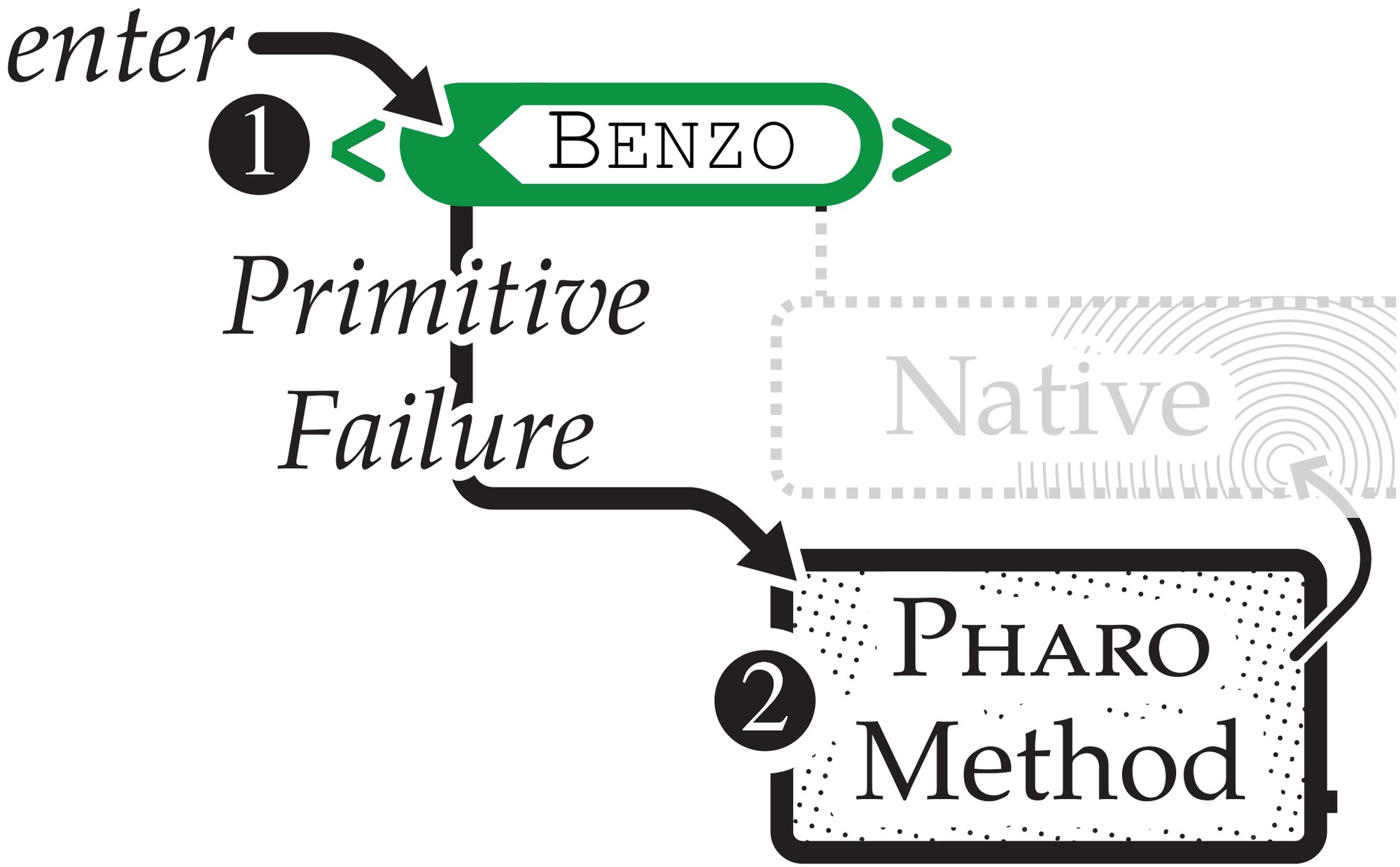

Pharo
Stack
<
Benzo Primitive
>


Pharo
Stack
<
Benzo Primitive
>
?

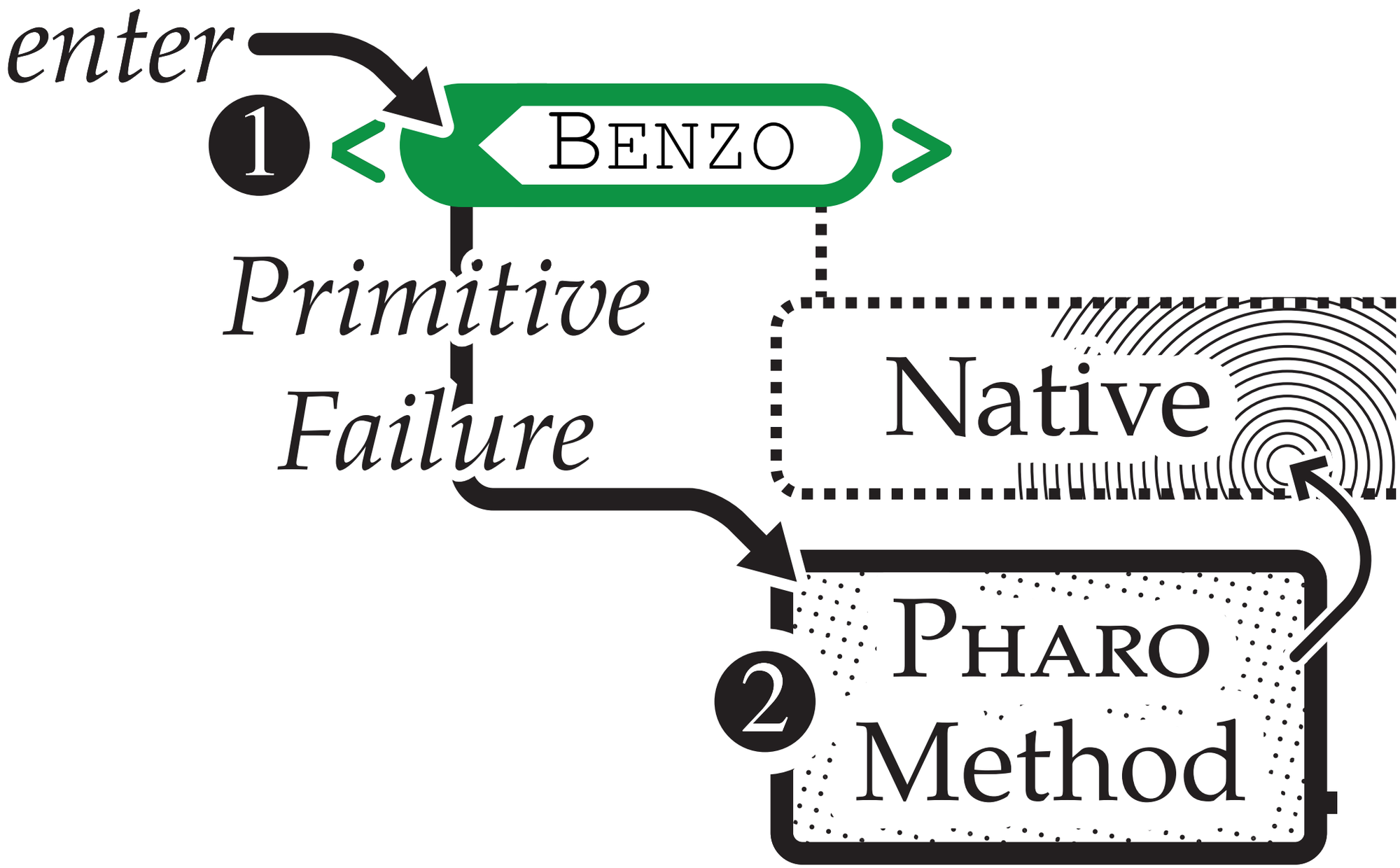
Native Code Generation

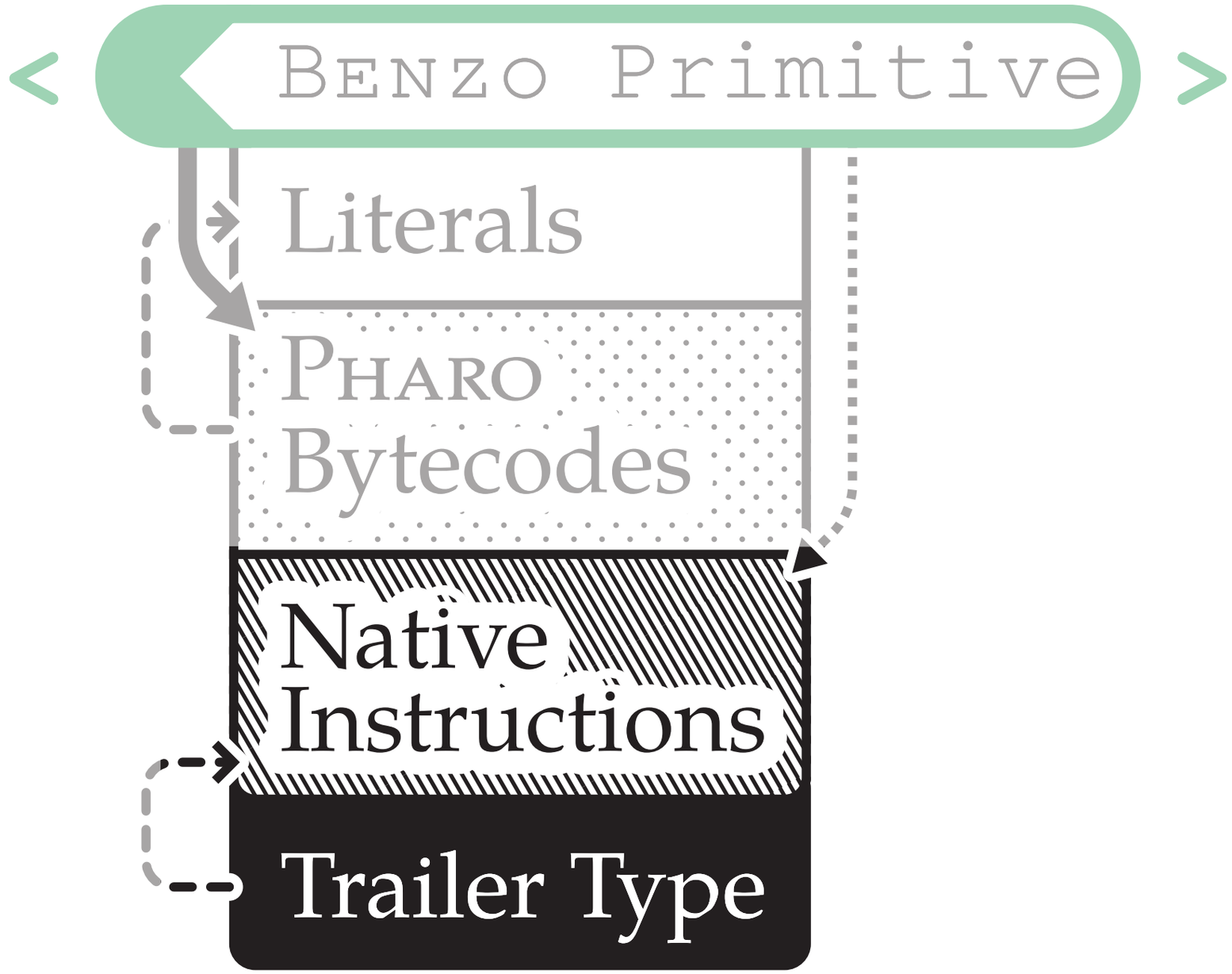


Native Code Generation

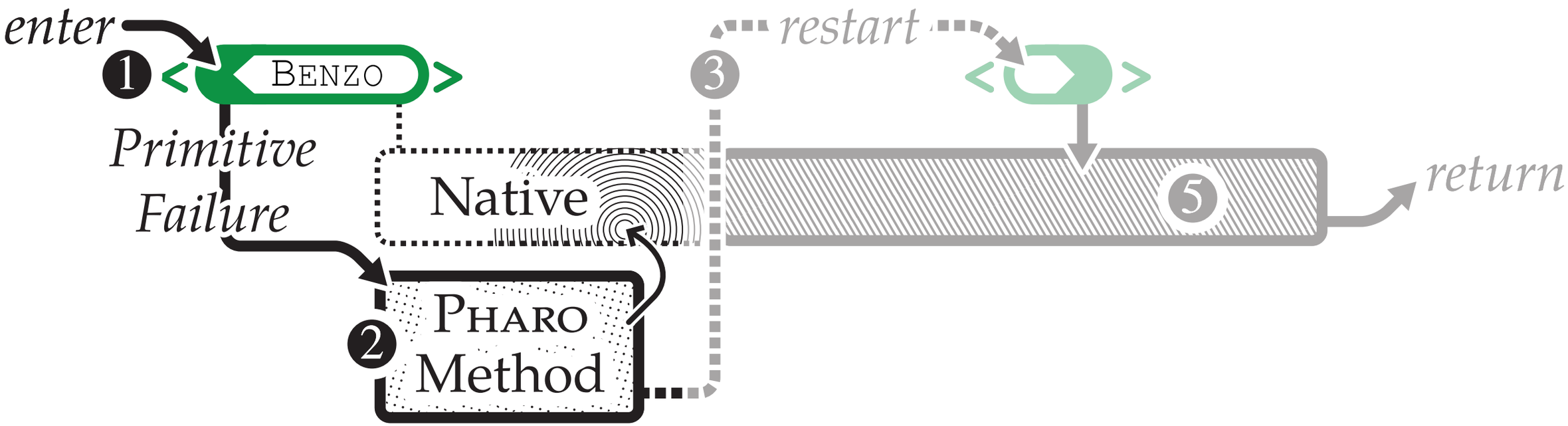


Native Code Generation

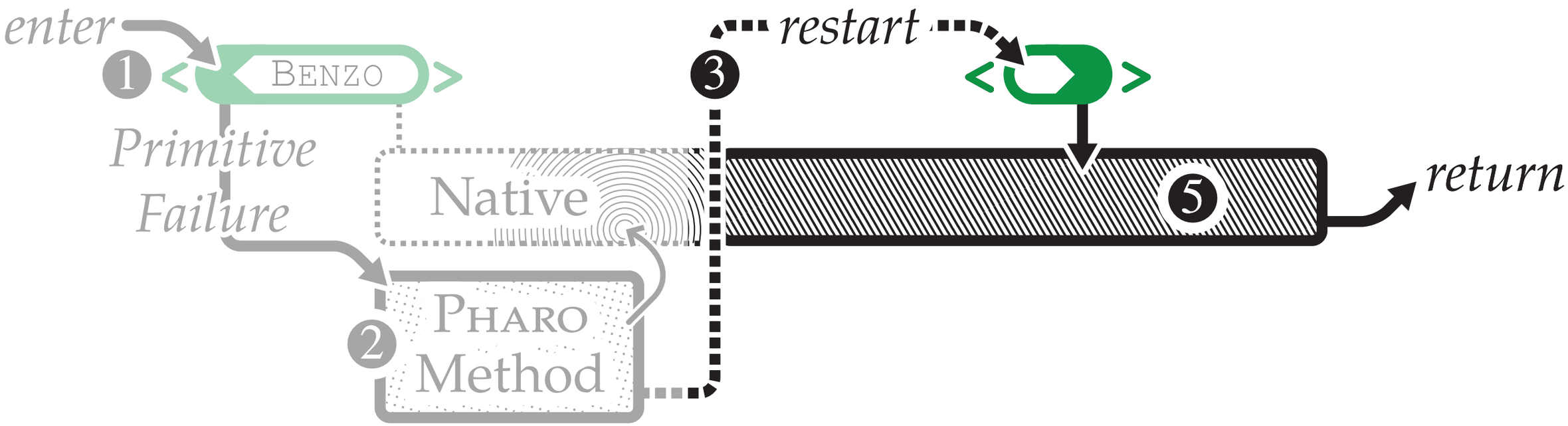


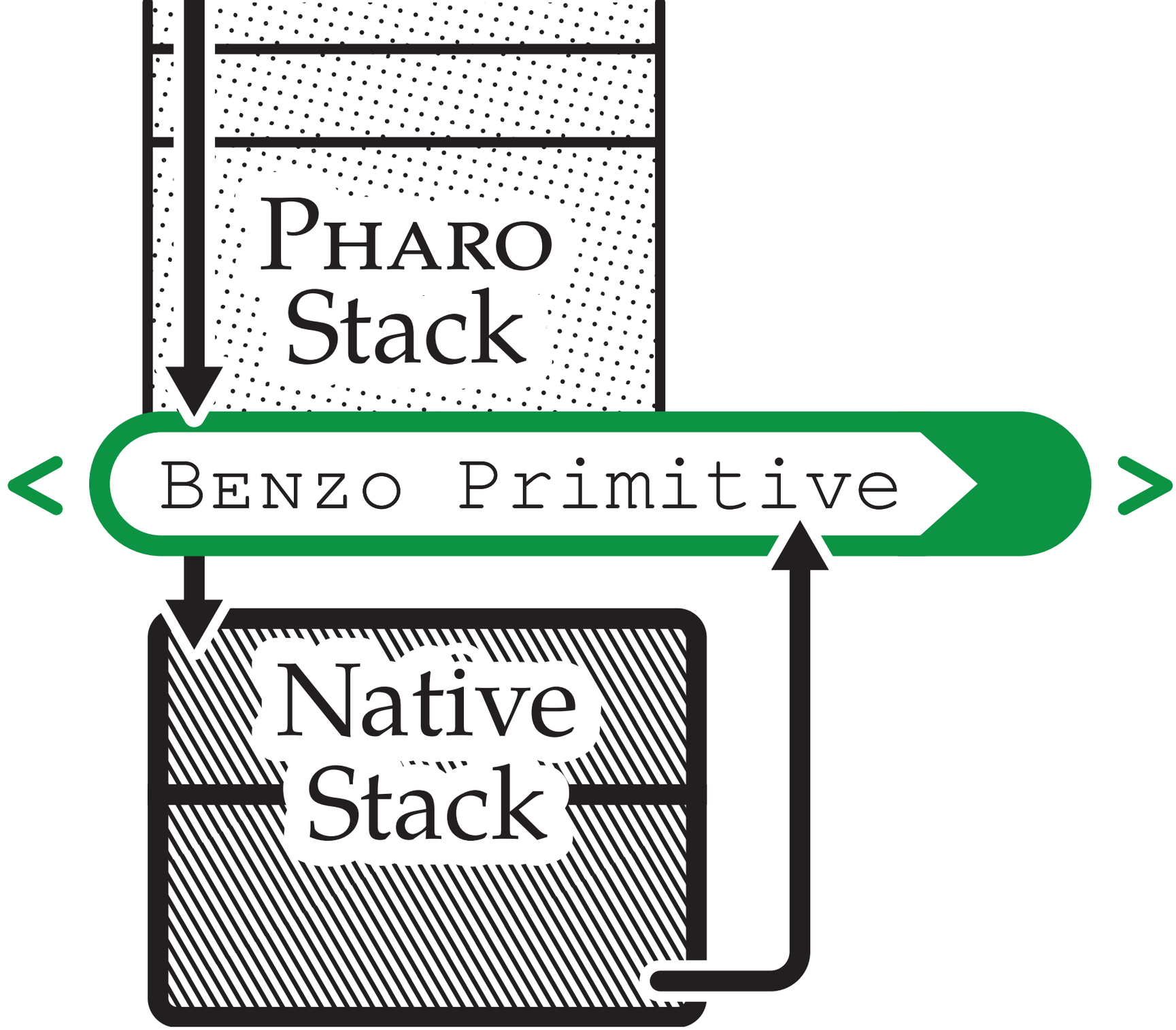

Benzo Primitive
Literals
Pharo Bytecodes
Native Instructions
Trailer Type

Native Code Activation

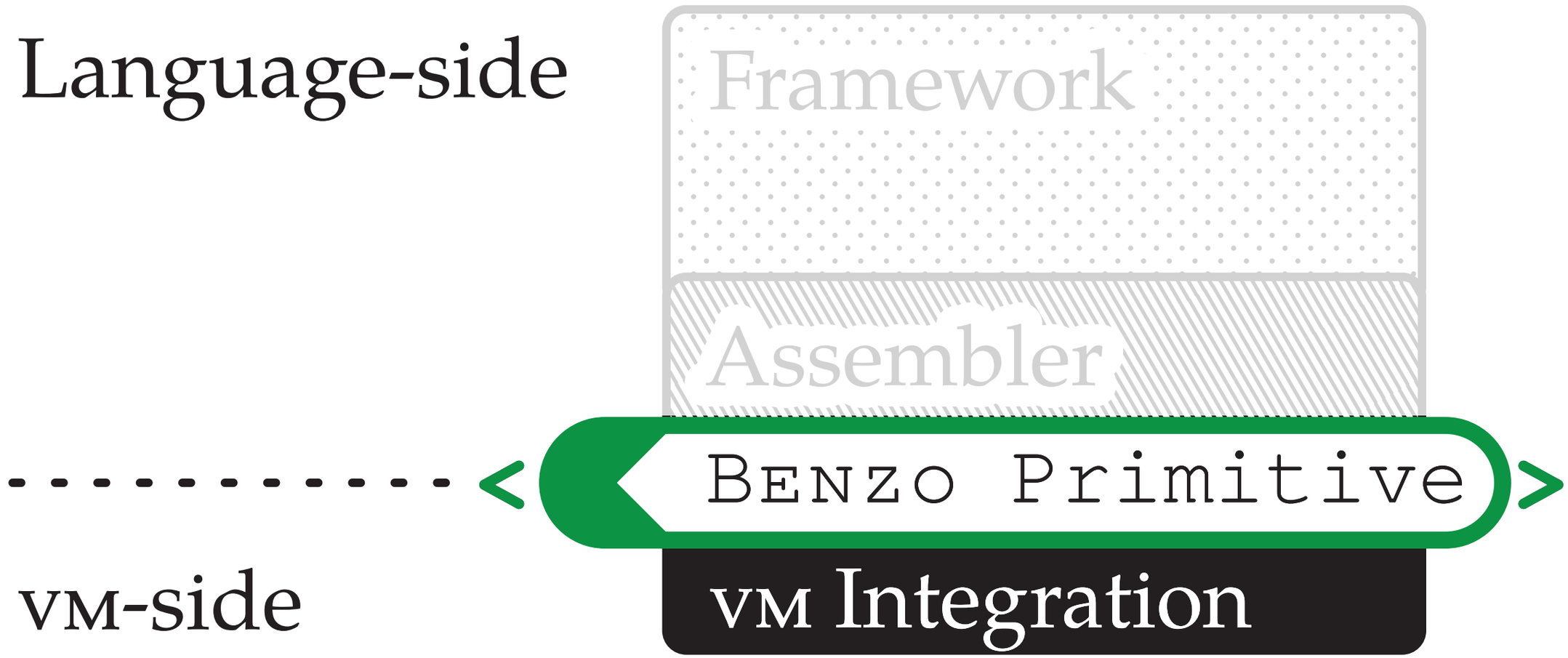


Native Code Activation

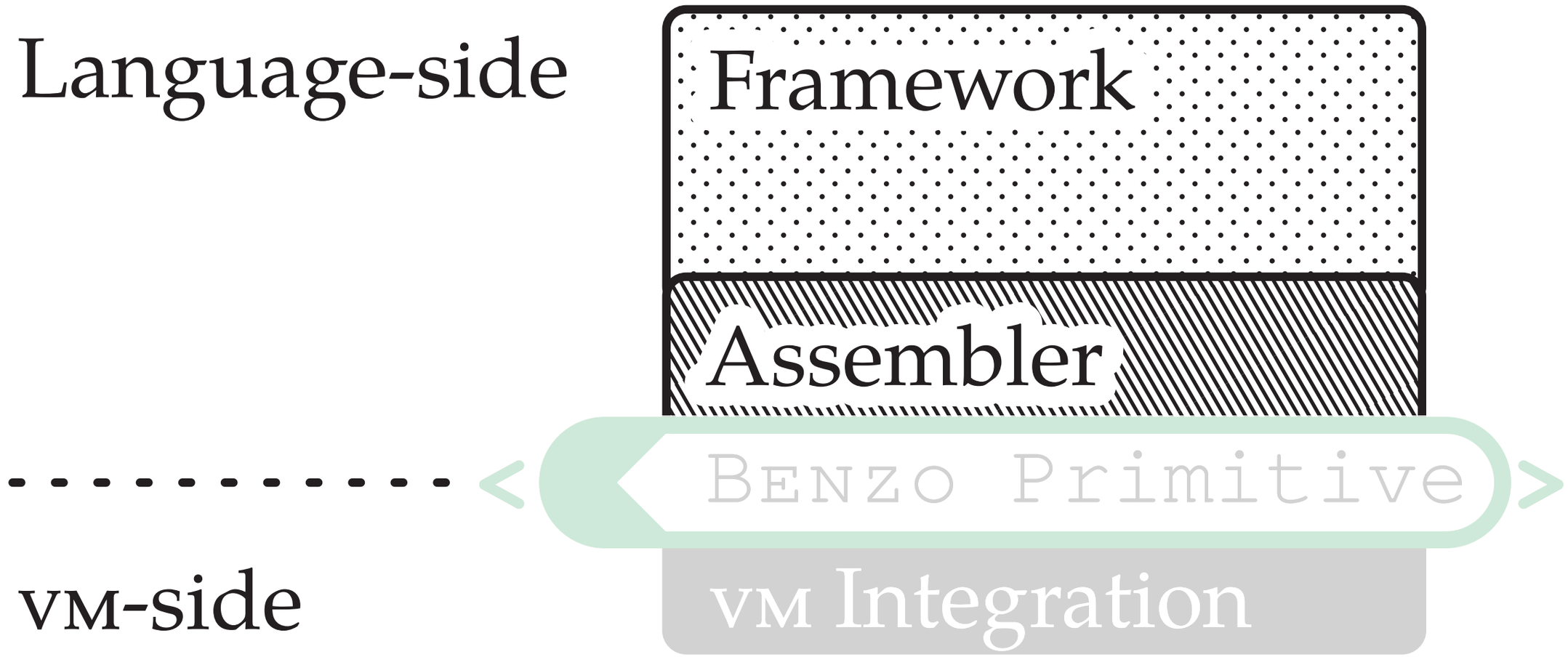



Pharo
Stack
<
Benzo Primitive
>
Native
Stack

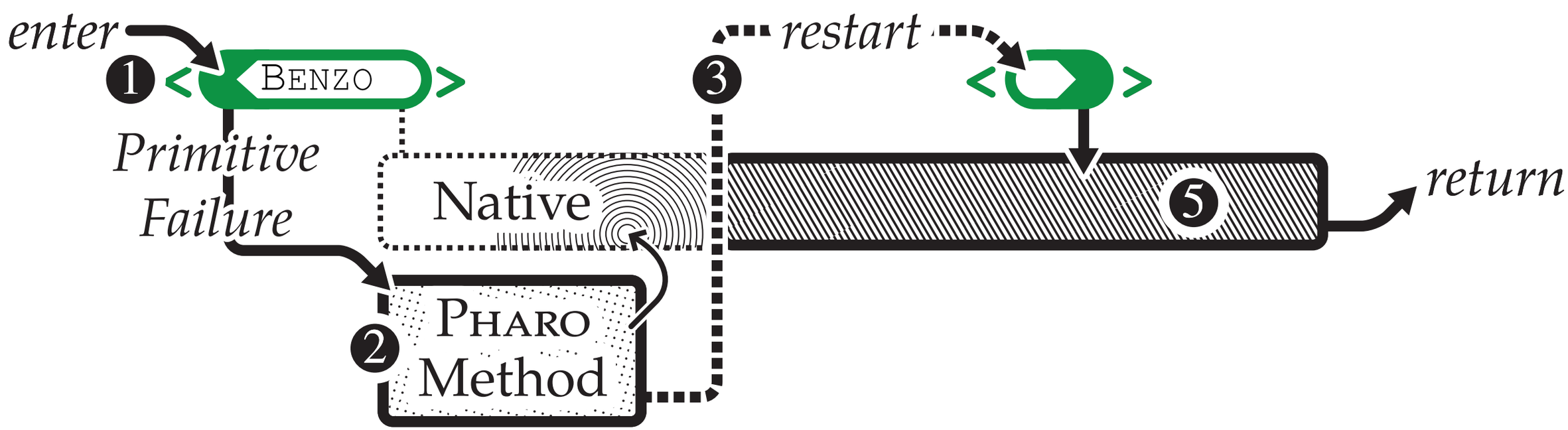
General Design



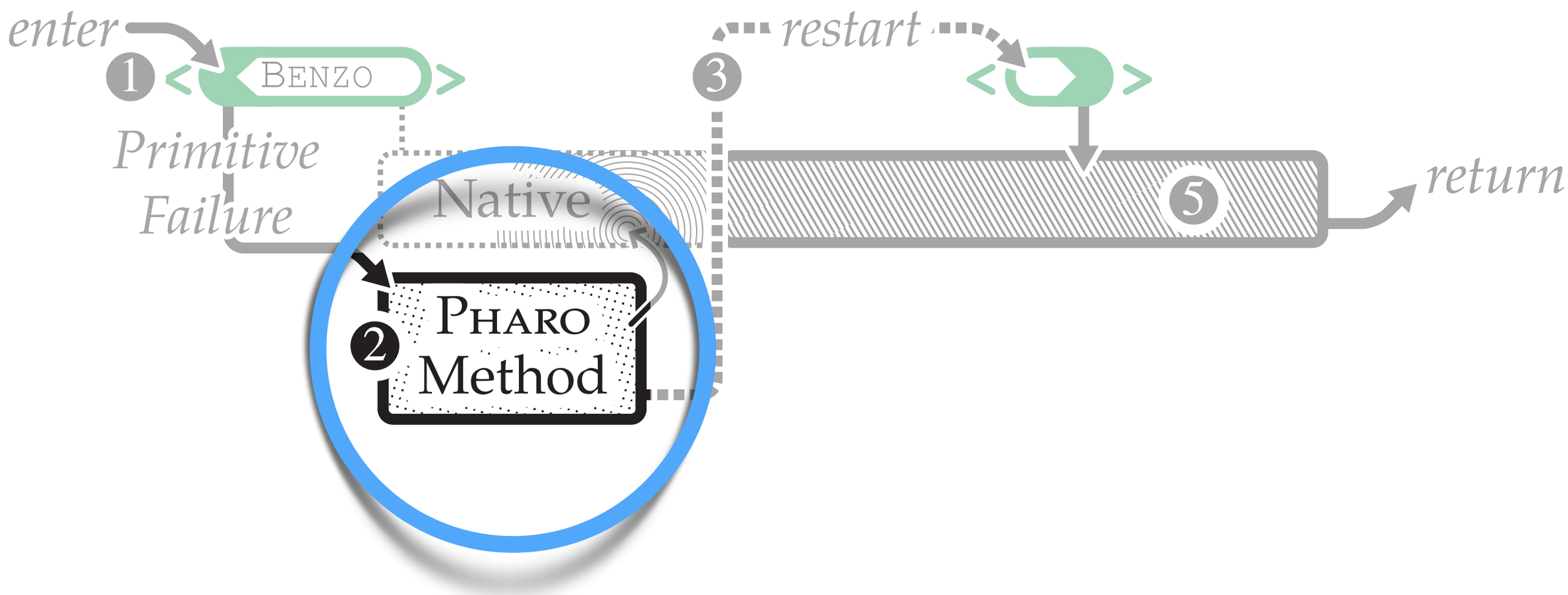
General Design



One-time Overhead



One-time Overhead



Benzo Example

```
<primitive: #nativeCode module: #Benzo>

Benzo x86 generate: [ :asm :helper |
  asm
    mov: 1 asUImm
    to: asm registers EAX ].
```

Benzo Example

```
<primitive: #nativeCode module: #Benzo>

Benzo x86 generate: [ :asm :helper |
  asm
    mov: 1 asUImm
    to: asm resultRegister ].
```

BENZO Example

```
<primitive: #nativeCode module: #Benzo>

Benzo x86 generate: [ :asm :helper |
   | arrayRegister |

      arrayRegister := helper classArray.
      arrayRegister := helper
            instantiateClass: register
            indexableSize: 10.
      asm
         mov: arrayRegister
         to: asm resultRegister ].
```

BENZO Summary

Access to Native Code

One-time Code Generation Overhead

Language-side Control

Minimal VM Modifications

Compatible with Existing Runtime

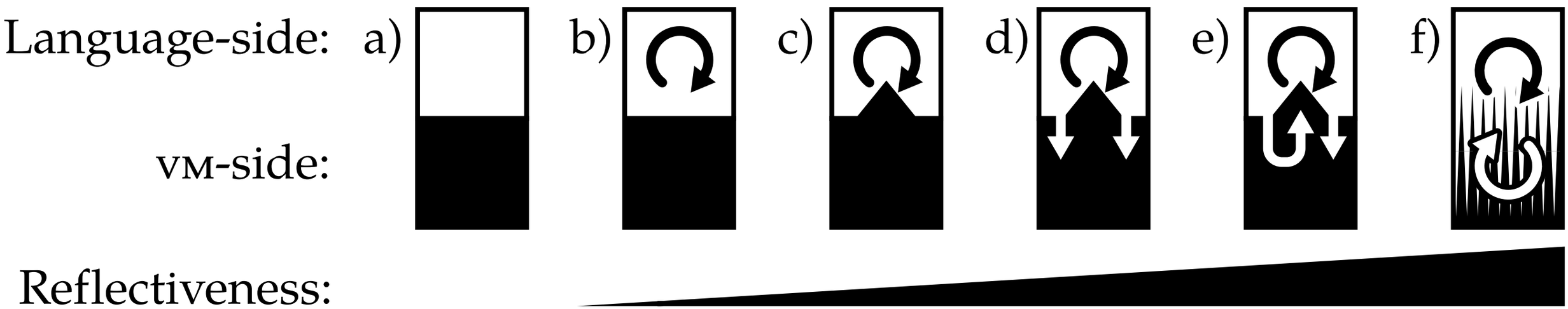
BENZO Summary

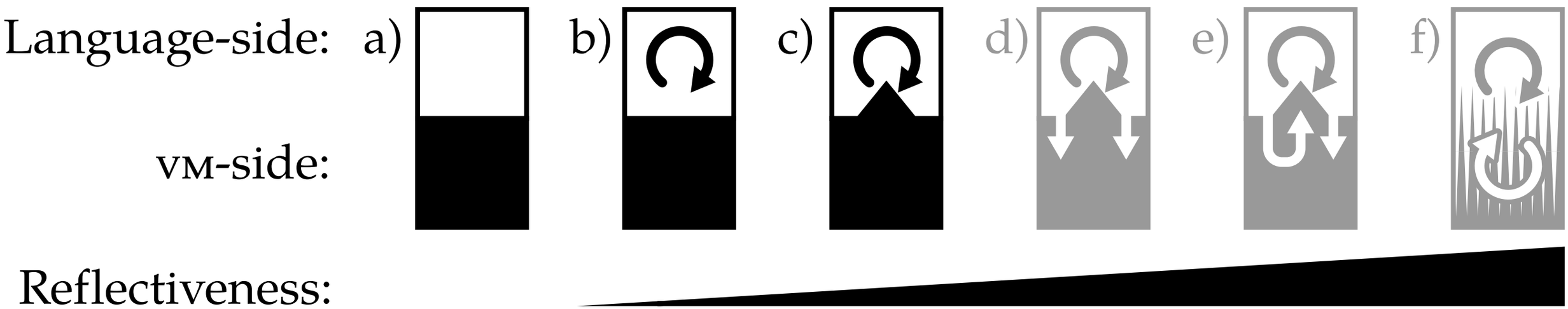
	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
PINOCCHIO	–	+	~
KLEIN	–	+	~
BENZO	+	+	?

Vision
Related Work Analysis
Solution
Validation
Conclusion & Future Work

Experiment Design

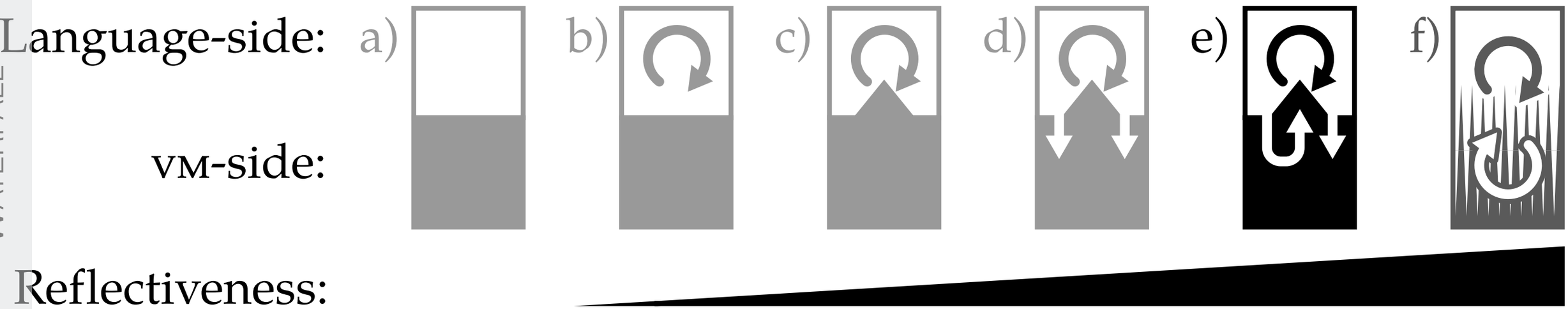
Towards Self–awareness




Language-side:
a)
b)
c)
d)
e)
f)
VM-side:
Reflectiveness:

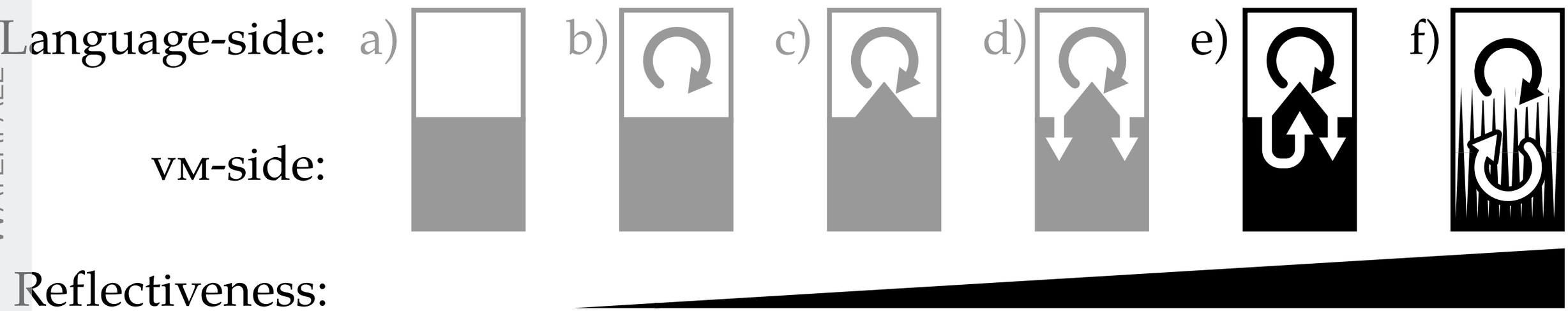
NativeBoost FFI

Glue Code to Existing External Functionality


Language-side:
vm-side:
Reflectiveness:
a)
b)
c)
d)
e)
f)

Waterfall
Dynamic Primitives

Modifying Language Functionality on the Fly


Language-side:
vm-side:
Reflectiveness:
a)
b)
c)
d)
e)
f)

Nabujito

Language-side JIT

Controlling Code Execution

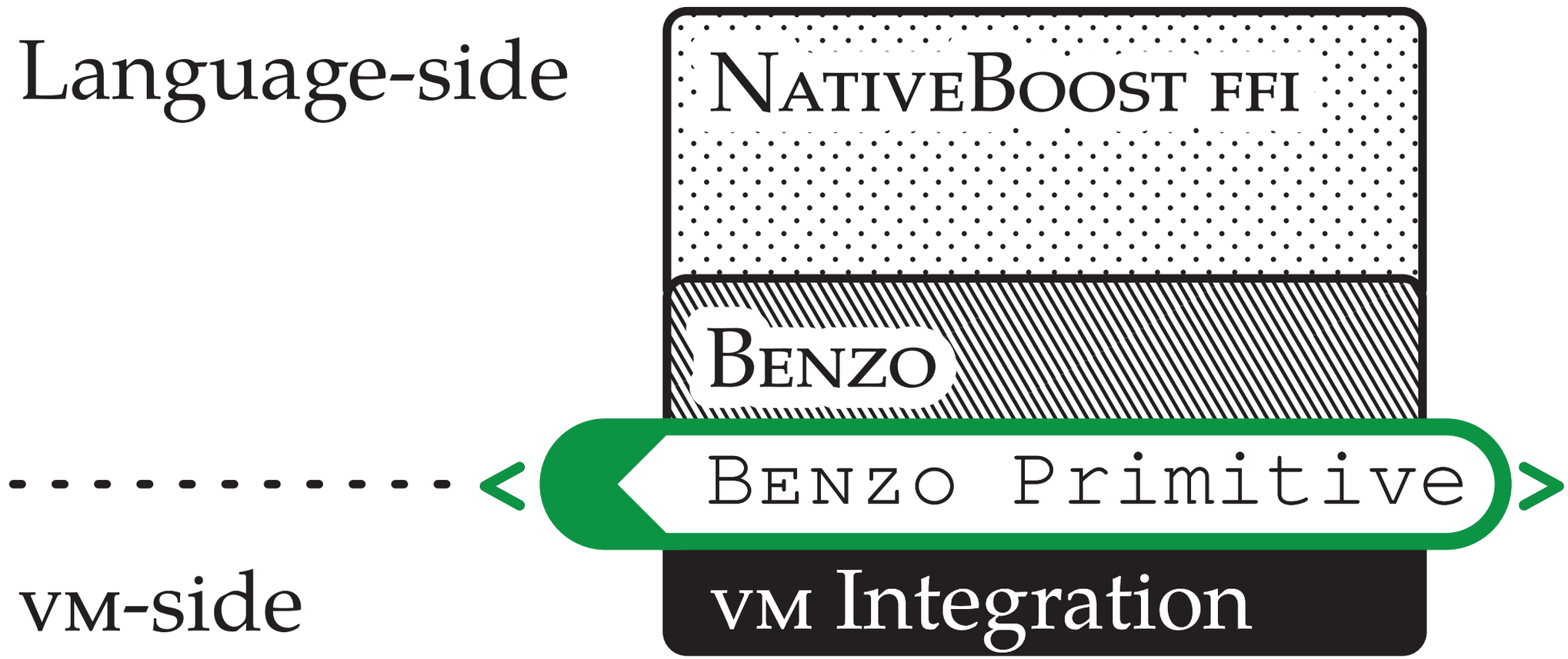
Benzo Experiment Details

NativeBoost FFI

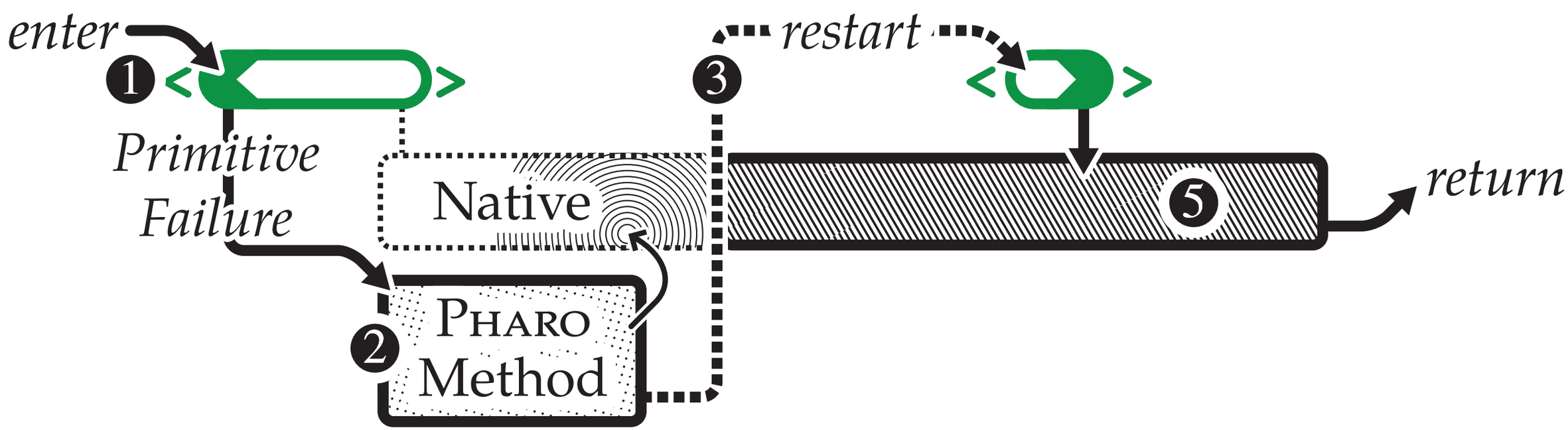
with Igor Stasenko

C. Bruni, S. Ducasse, I. Stasenko and L. Fabresse
IWST '13

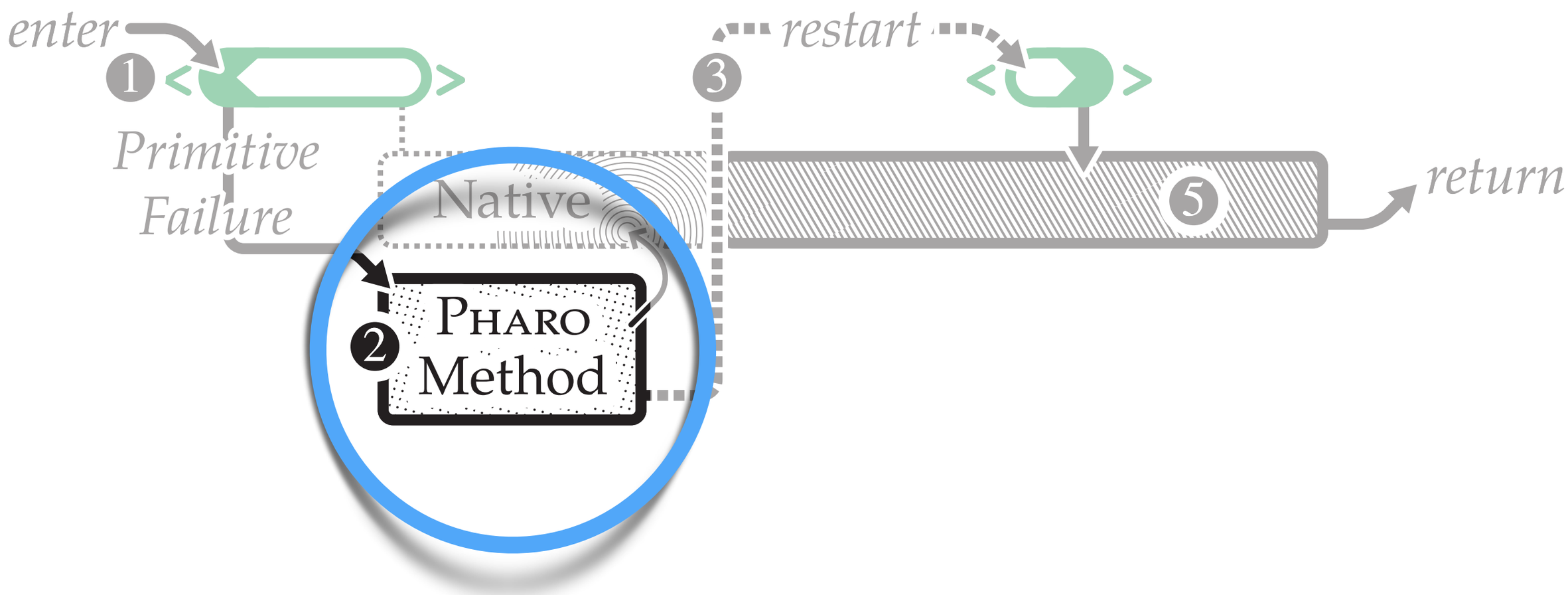
Implementation



One-time Overhead



One-time Overhead



Example

```
abs: anInteger
    <primitive: #nativeCode module: #Benzo>
    ^ FFI call: #(uint abs(int anInteger))
```

Example

```
abs: anInteger
    <primitive: #nativeCode module: #Benzo>
    ^ FFI call: #(uint abs(int anInteger))
```

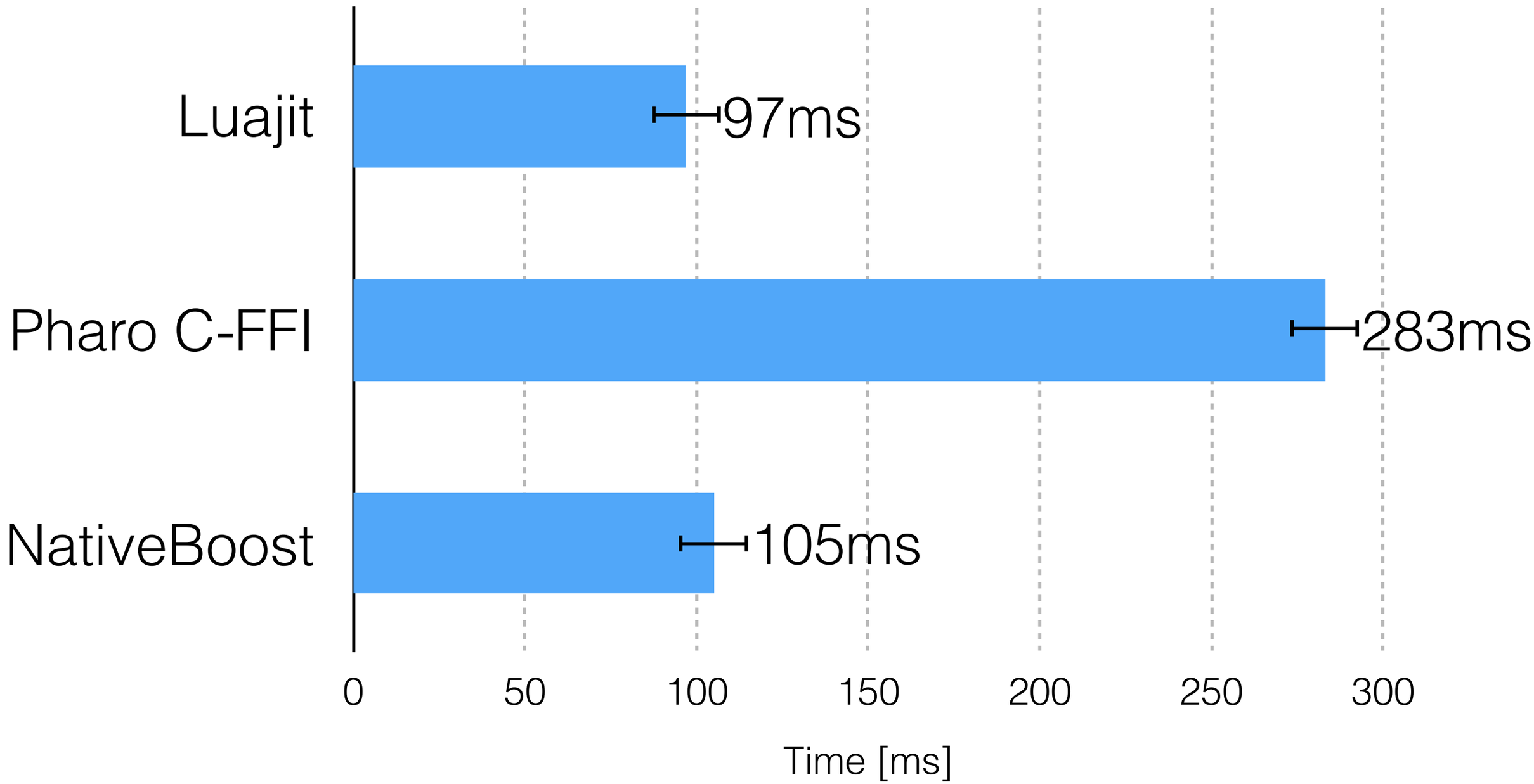
Functionality

Marshalling

Managed Struct Access

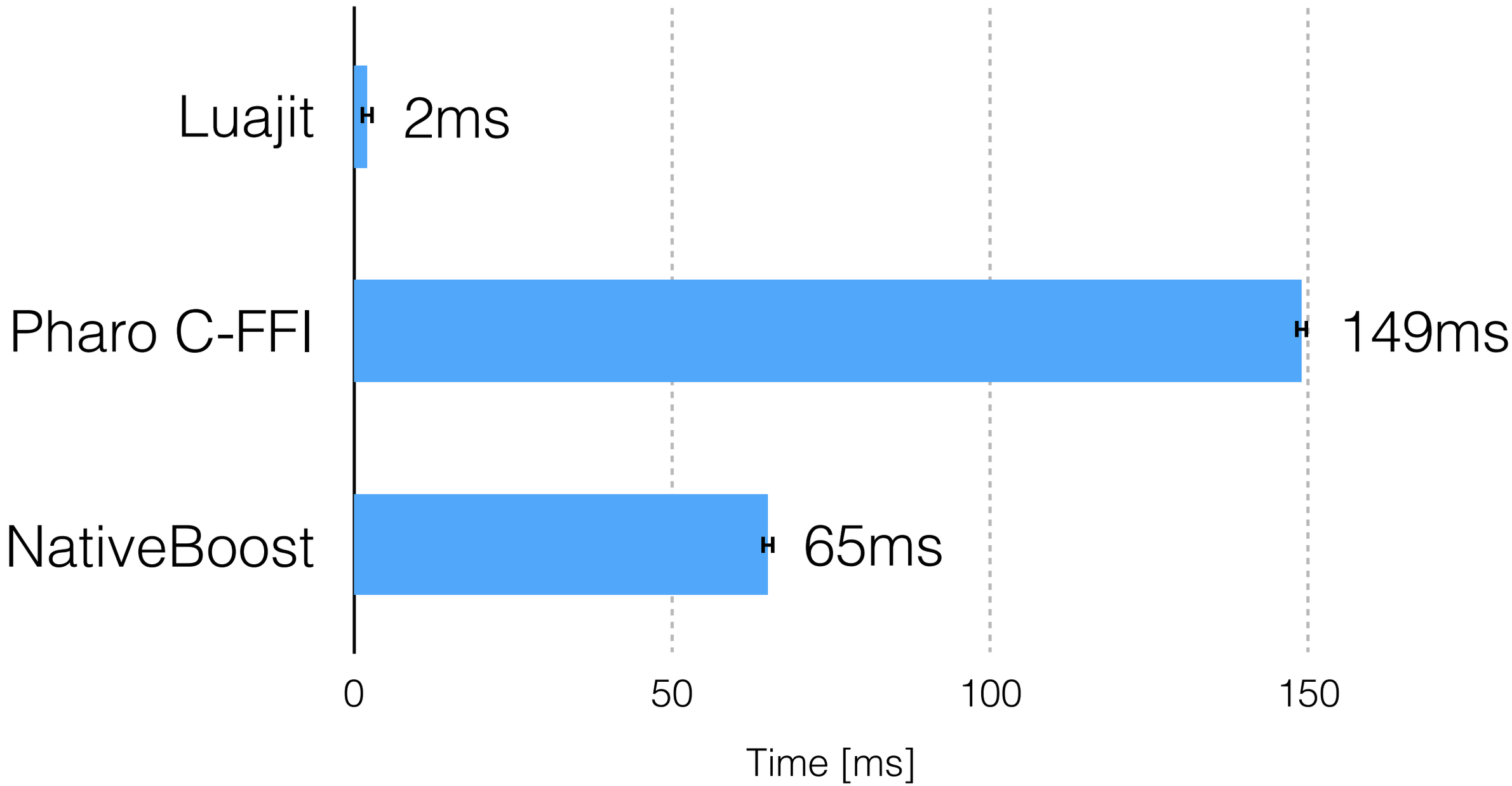
Basic Debugging with Error Codes

`getenv('PWD')` Performance

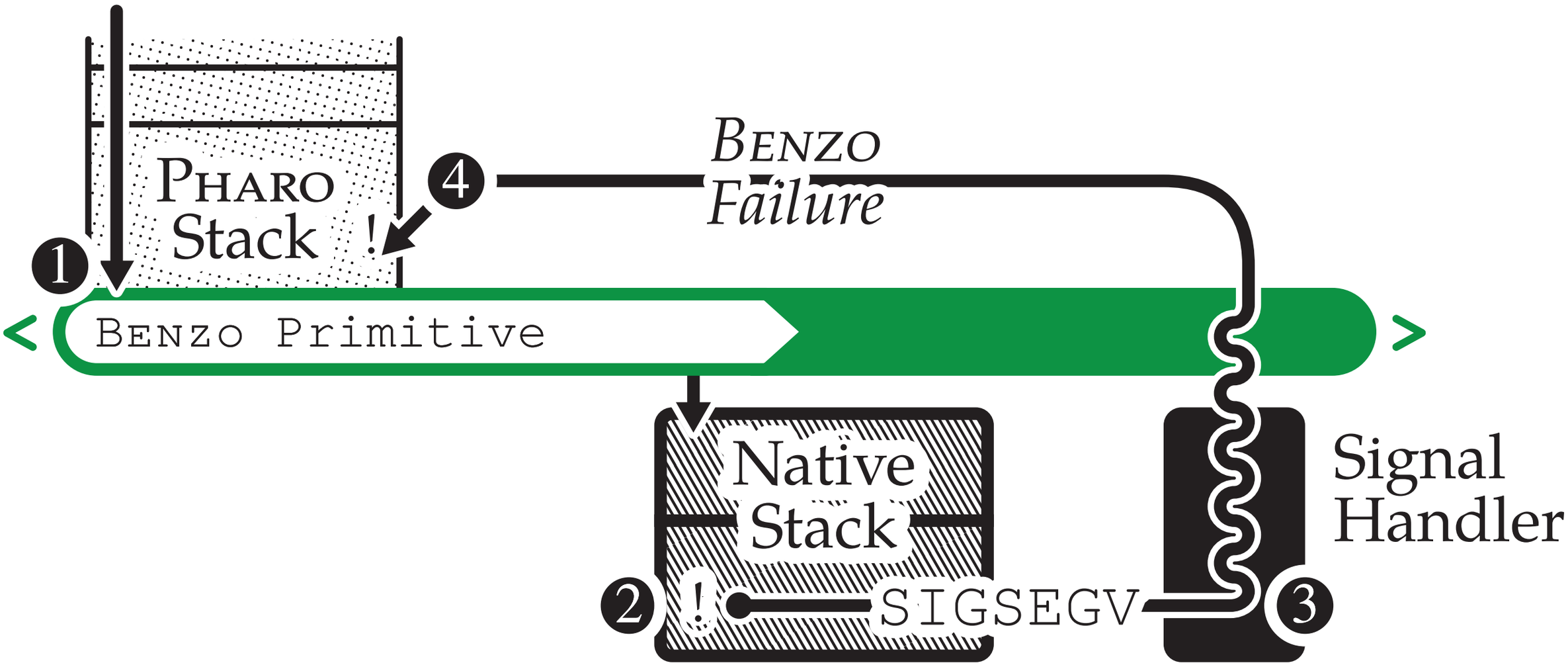


Limitations

abs(1) Performance



Missing Error Recovery

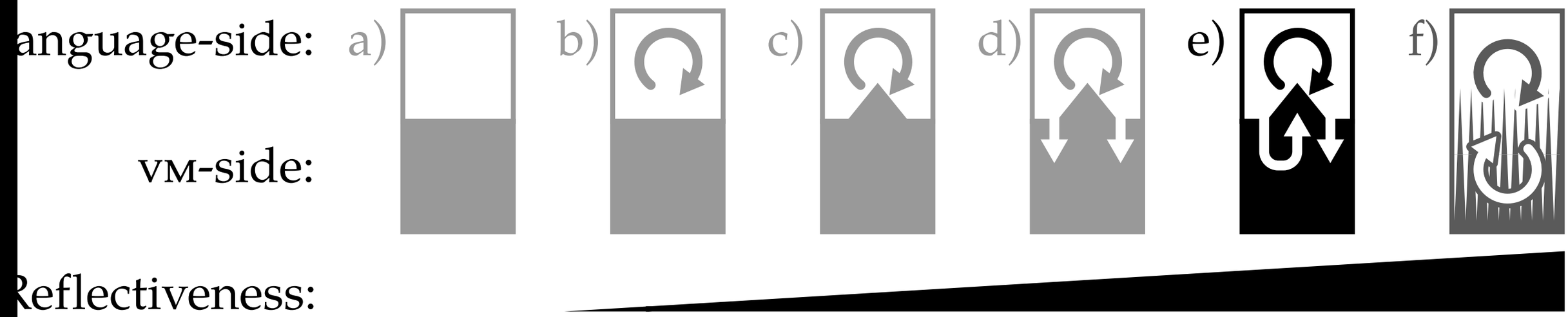


BENZO–based FFI

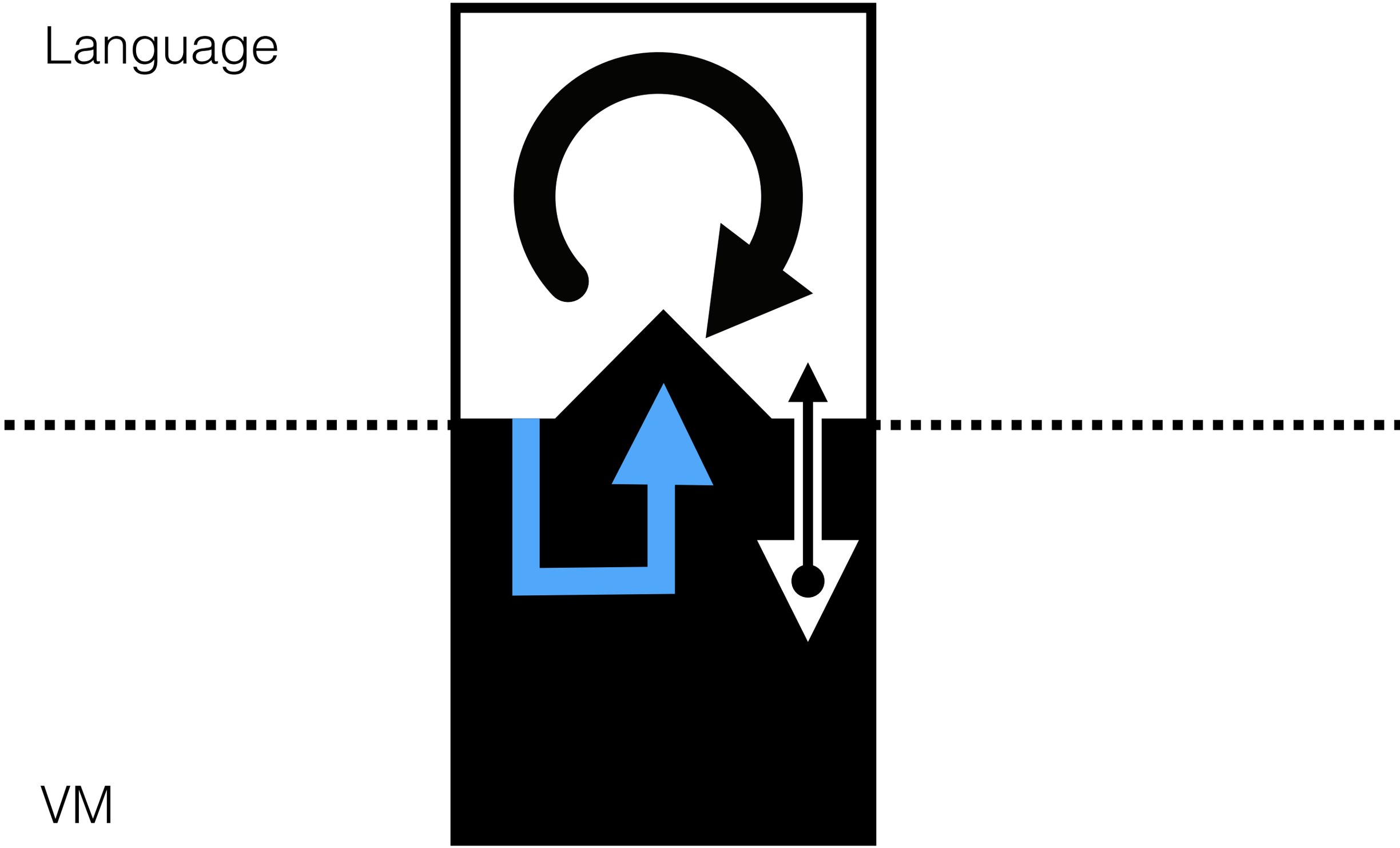
	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
PINOCCHIO	–	+	~
KLEIN	–	+	~
BENZO	**+**	**+**	**?**

Dynamic Primitives

with Guido Chari

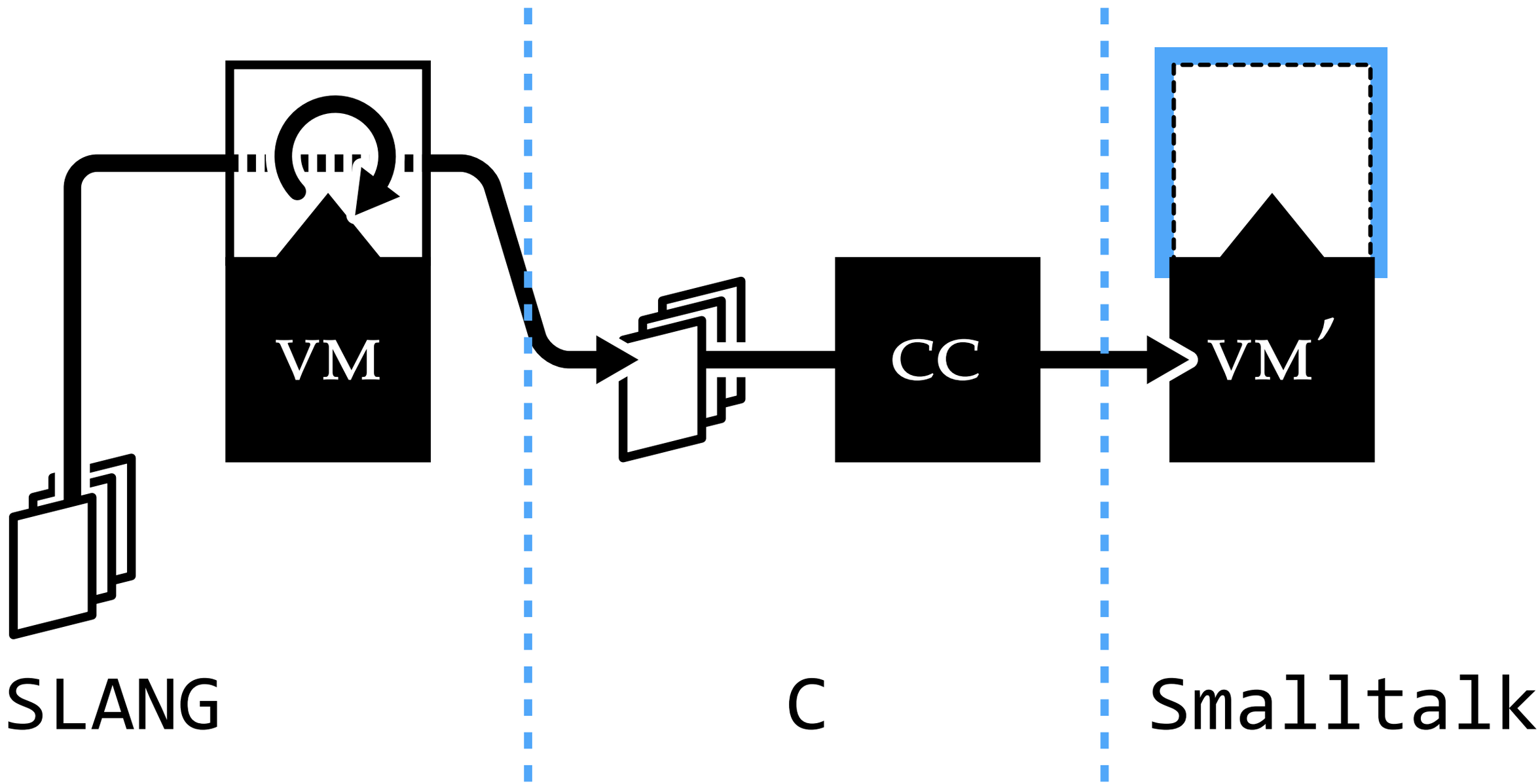

anguage-side:
vm-side:
Reflectiveness:
a)
b)
c)
d)
e)
f)

Language

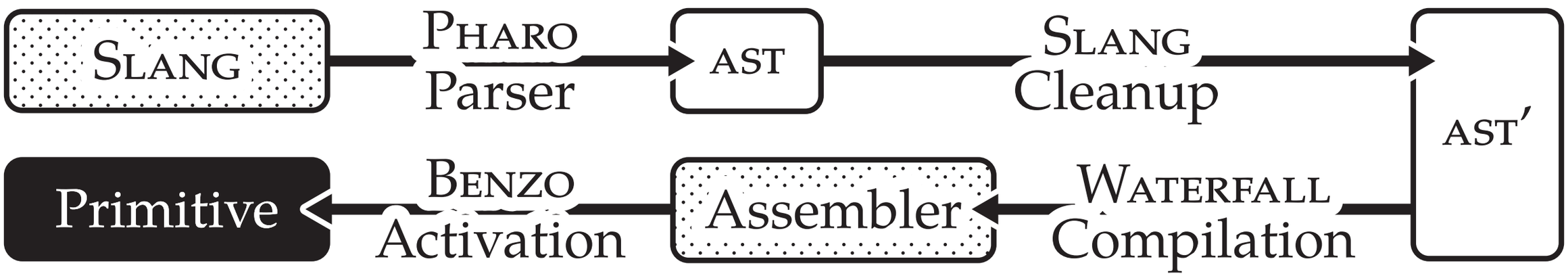


VM

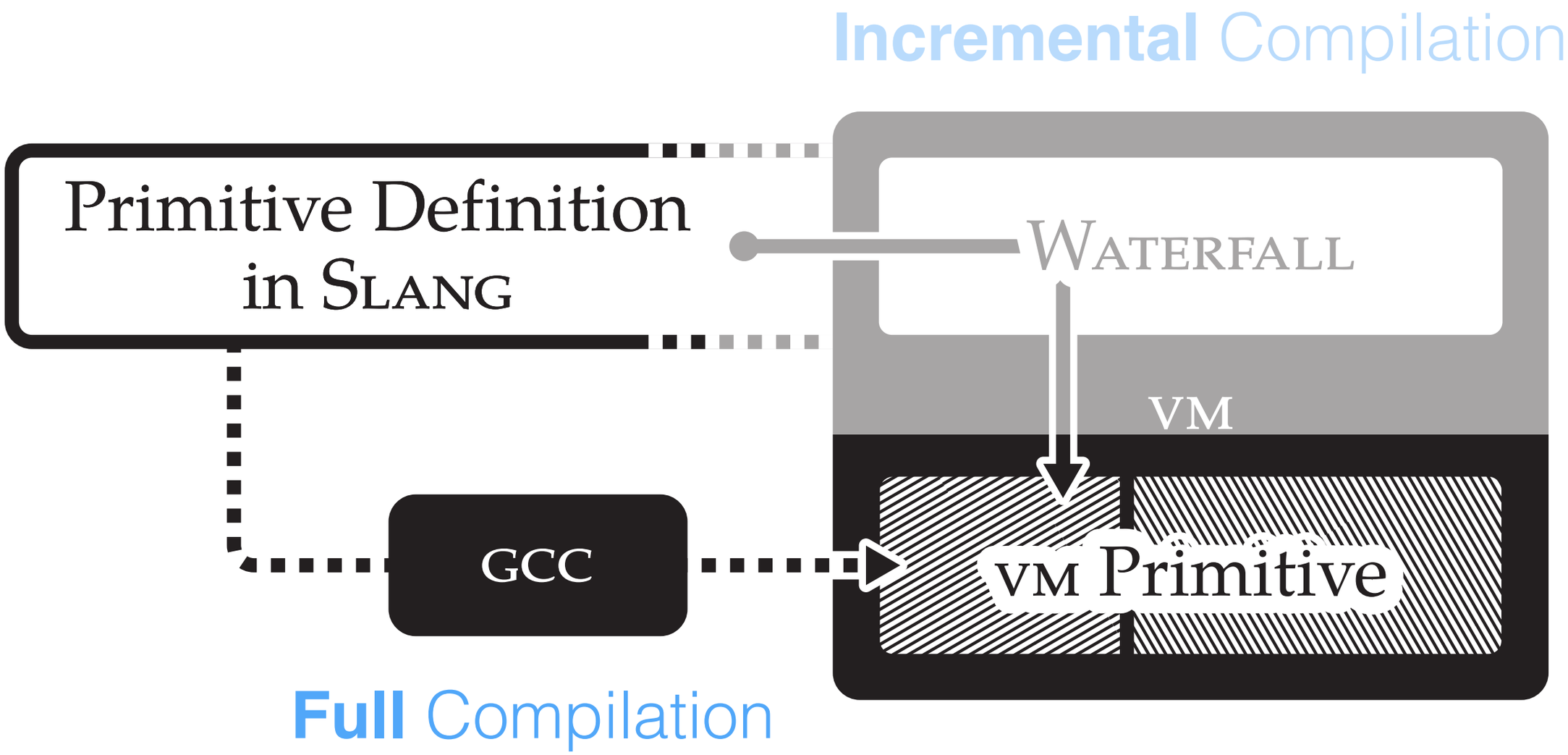
COG VM Bootstrap



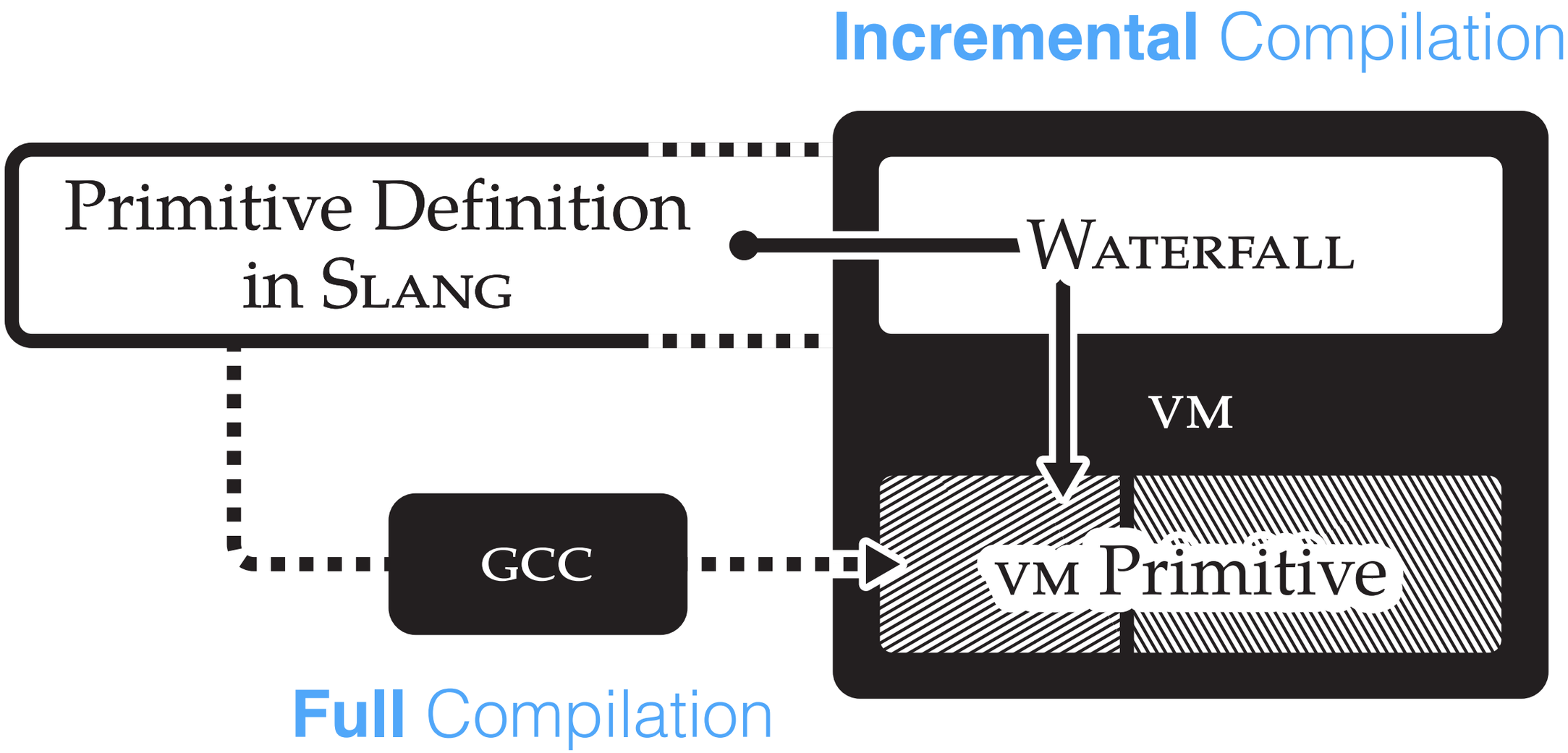
Compiler Pipeline



Code Reuse


Incremental Compilation
Primitive Definition in SLANG
WATERFALL
VM
GCC
VM Primitive
Full Compilation

Code Reuse

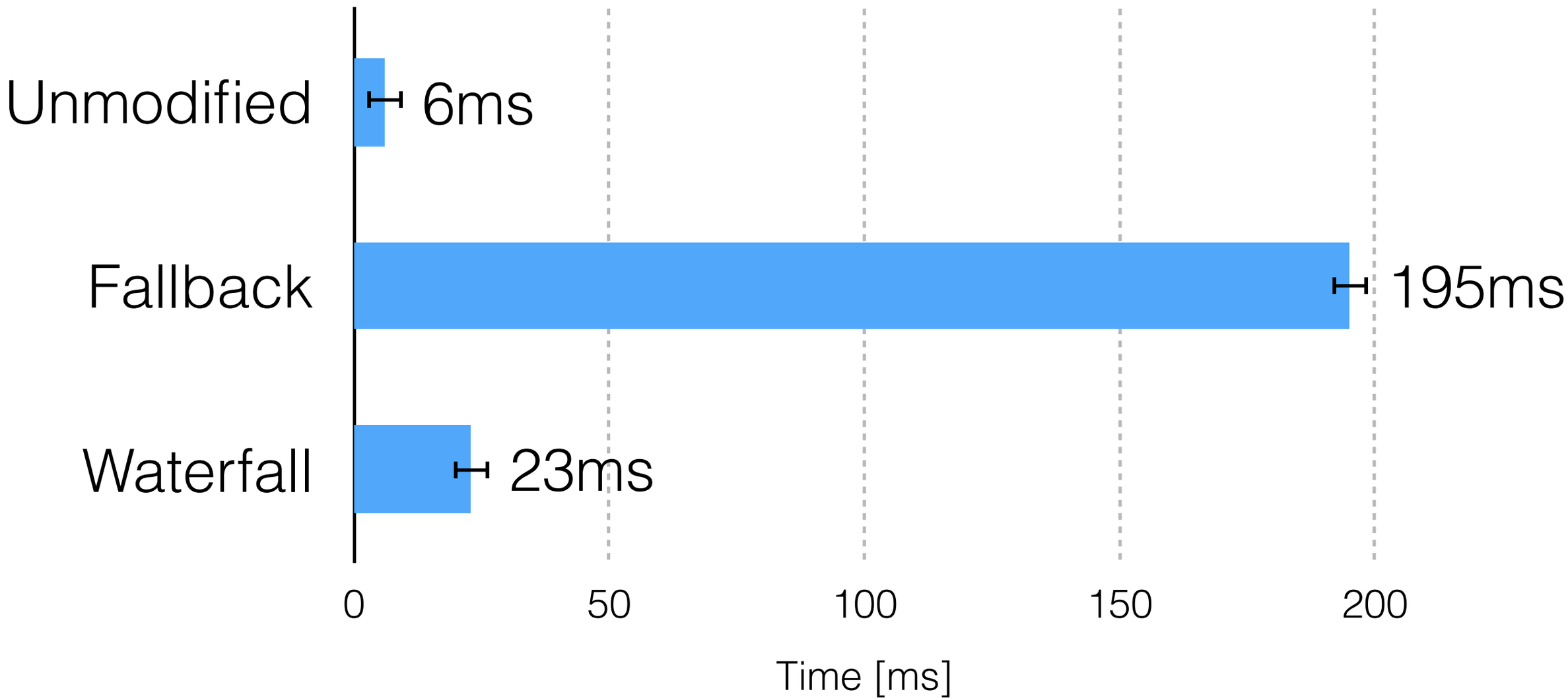

Incremental Compilation
Primitive Definition in SLANG
WATERFALL
VM
GCC
VM Primitive
Full Compilation

Dynamic Primitives Performance

Custom `Integer` Primitive

```
> aNumber
  <primitive: 4>
  aNumber isInteger
    ifFalse: [
      ↑ aNumber adaptToInteger: self andCompare: #> ]
  self negative == aNumber negative
    ifFalse: [ ↑ aNumber negative ].
  self negative
    ifTrue: [ ↑(self digitCompare: aNumber) < 0 ]
    ifFalse: [ ↑(self digitCompare: aNumber) > 0 ].
```

Custom `Integer` Primitive



Limitations

Lack of Static Optimizations

Minimal Debugging and Error Handling

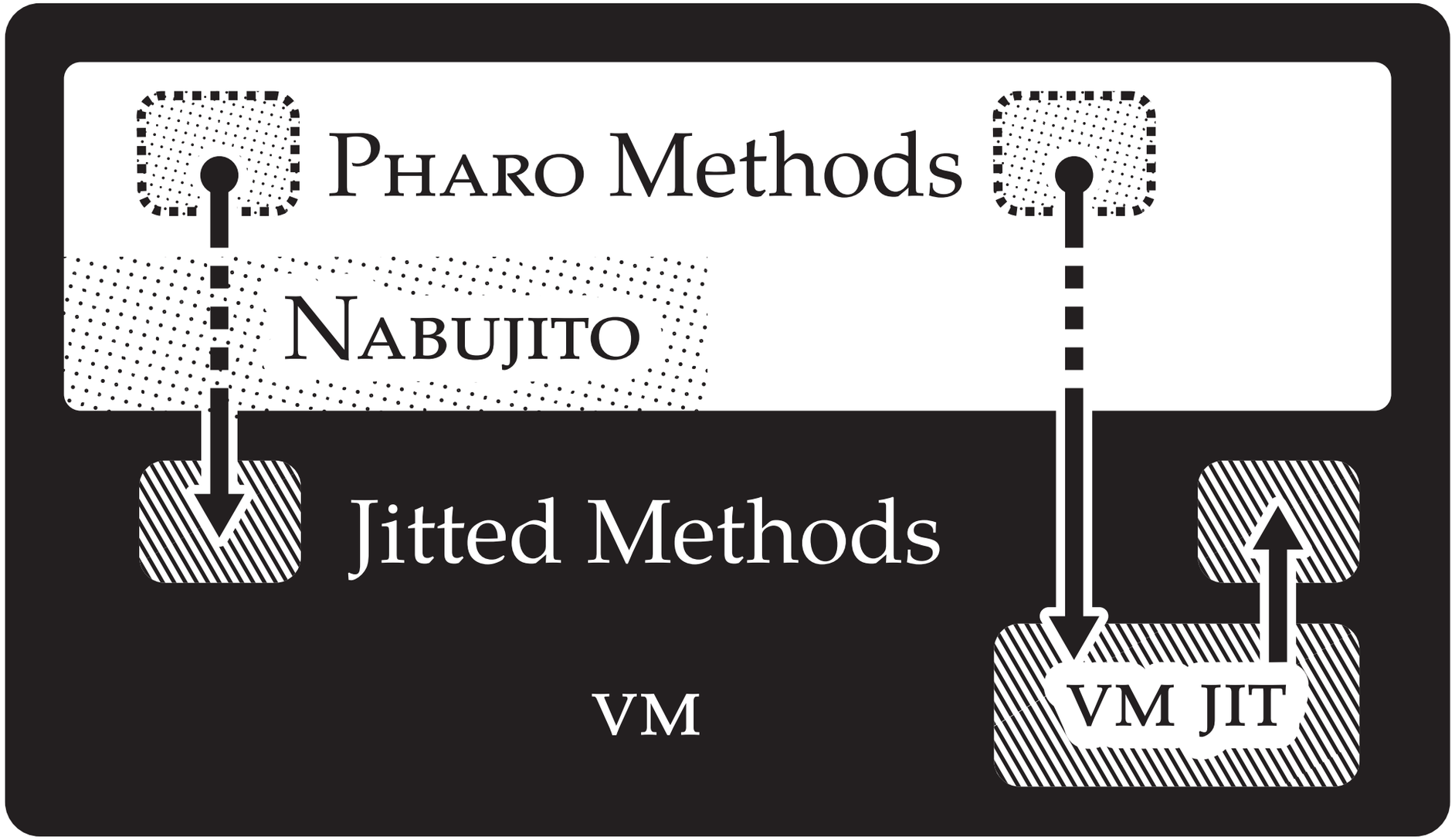
BENZO–based Dynamic Primitives

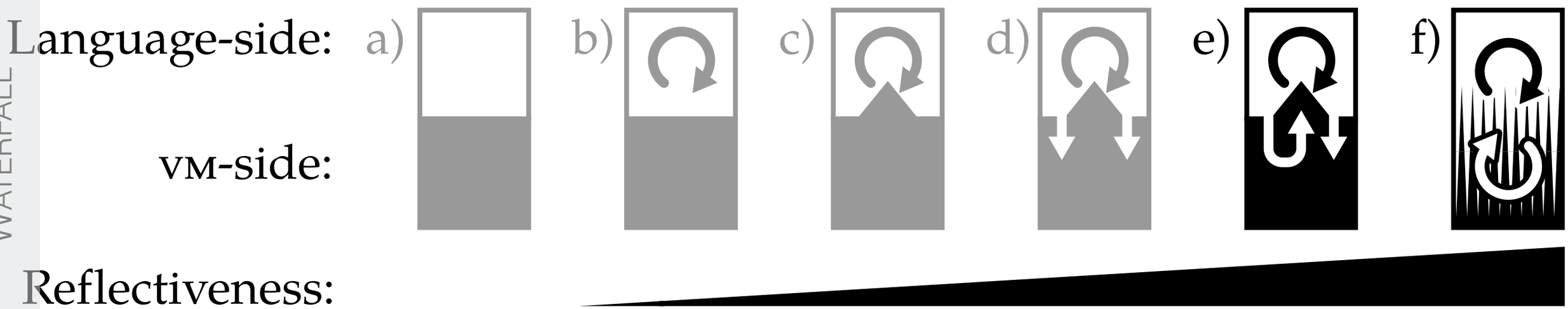
	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
PINOCCHIO	–	+	~
KLEIN	–	+	~
BENZO	**+**	**+**	**~**

Language-side JIT

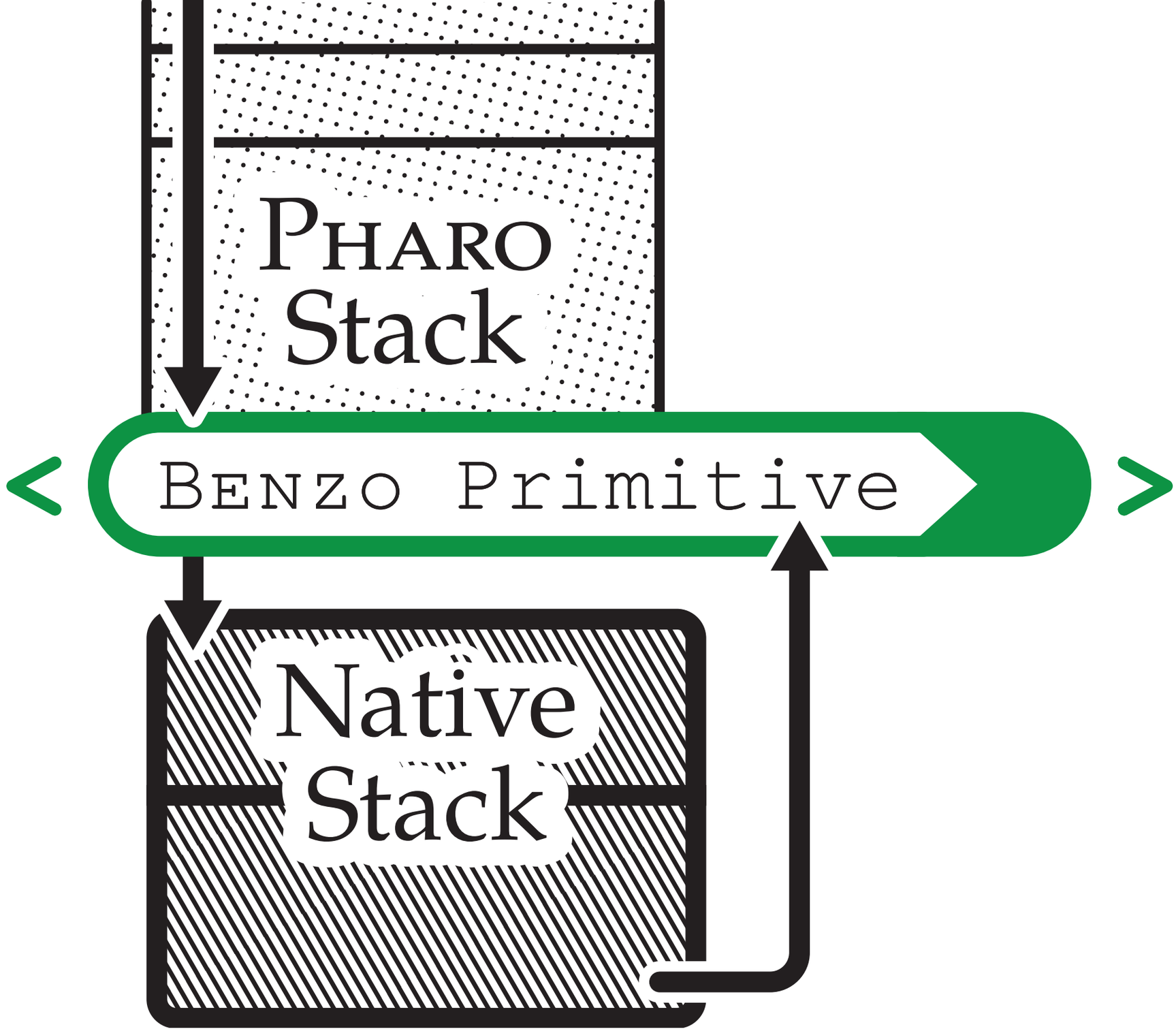
by Camillo Bruni

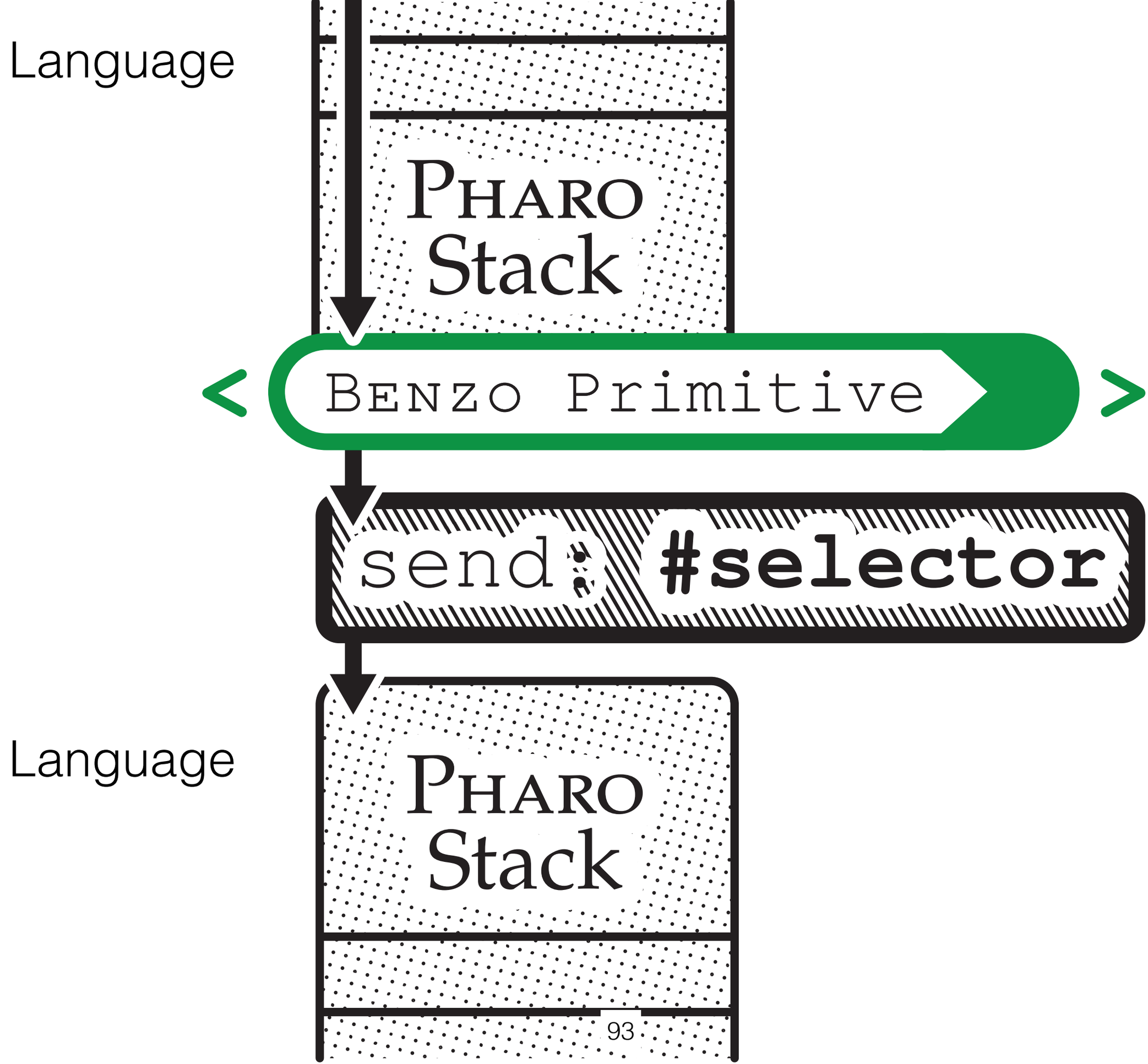
Implementation

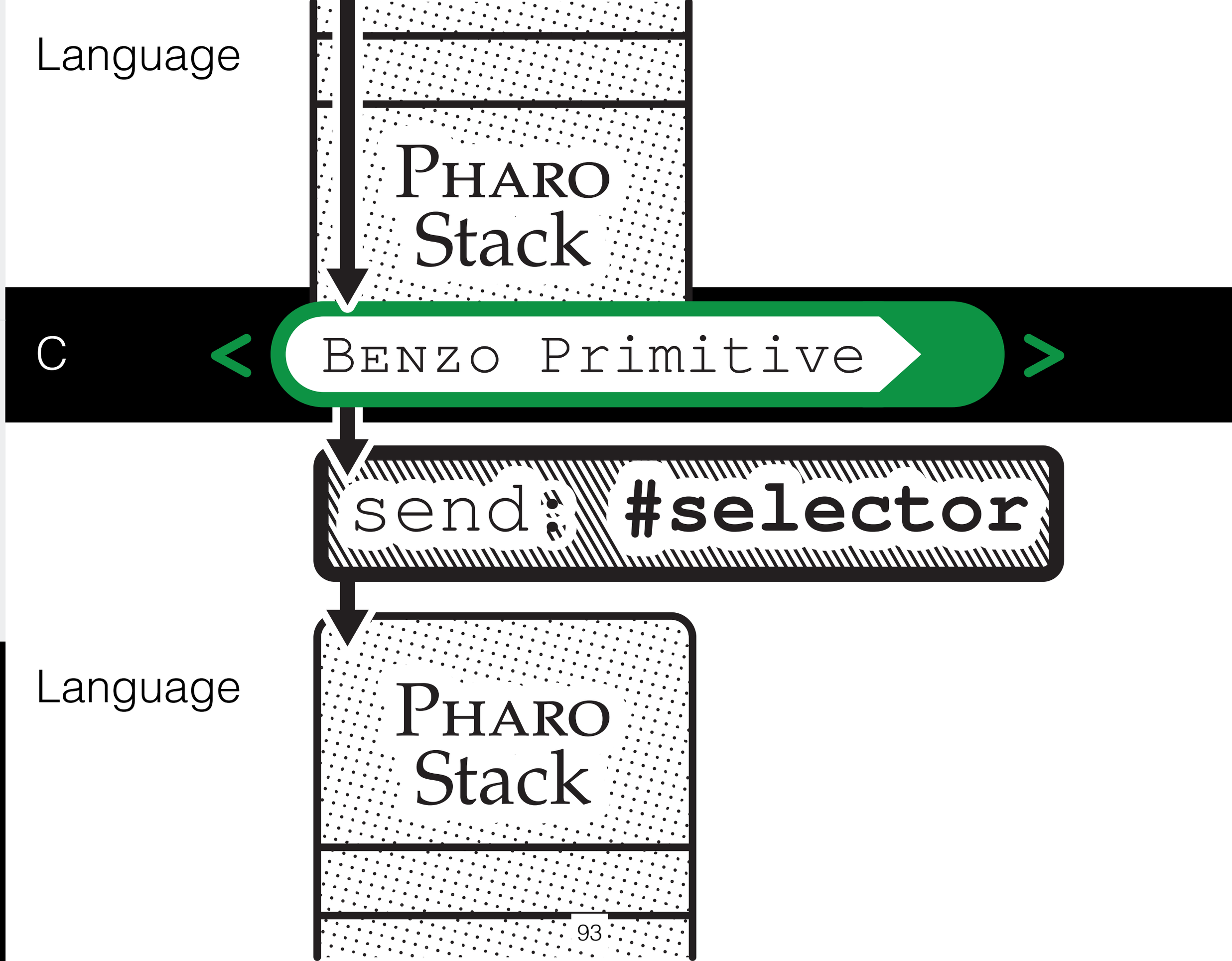



Language-side:
vm-side:
Reflectiveness:
a)
b)
c)
d)
e)
f)

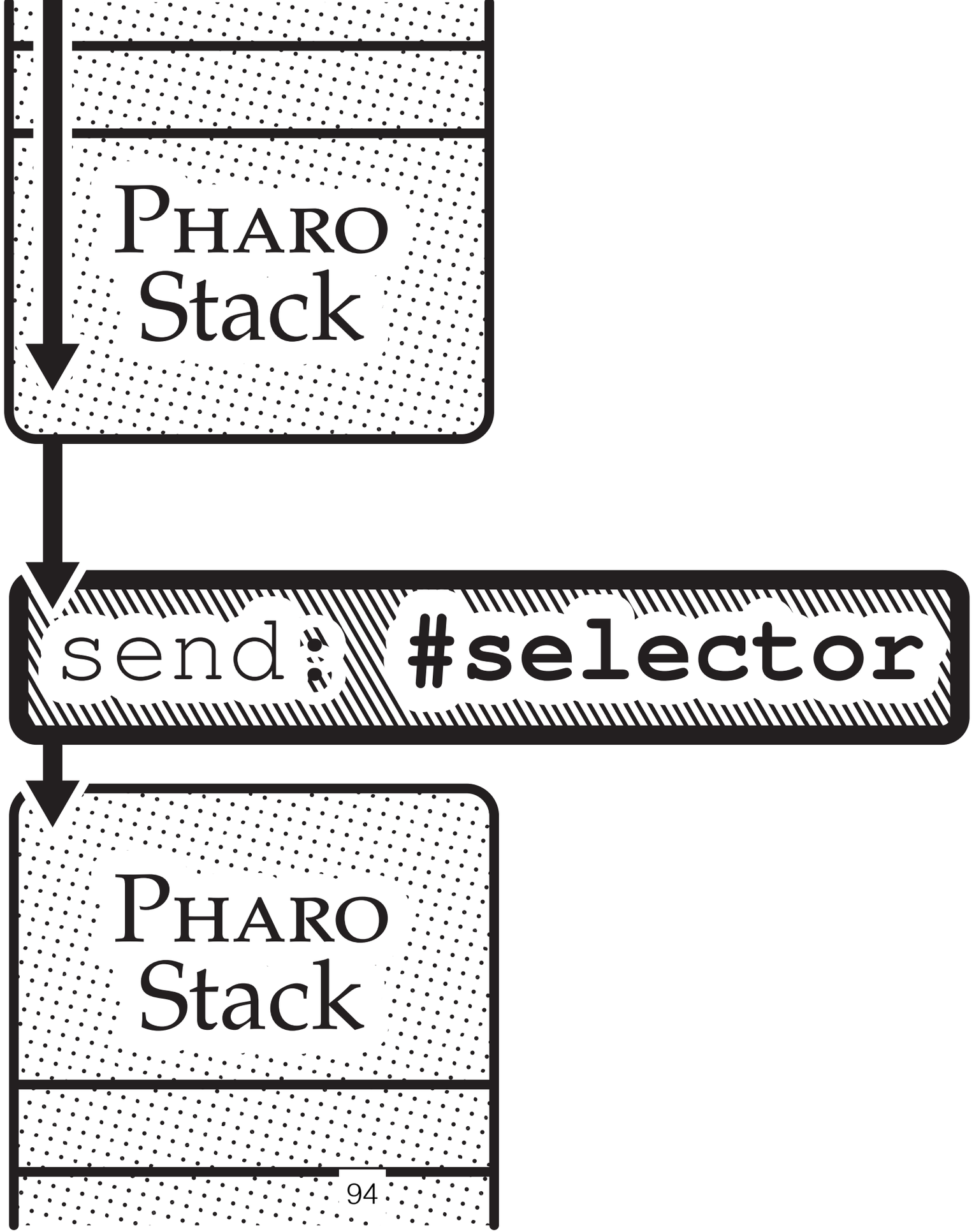
Mind the Context!


PHARO
Stack
<
BENZO Primitive
>
Native
Stack


Language
PHARO
Stack
<
BENZO Primitive
>
send
#selector
Language
PHARO
Stack


Language
PHARO
Stack
C
<
BENZO Primitive
>
send
#selector
Language
PHARO
Stack

Language

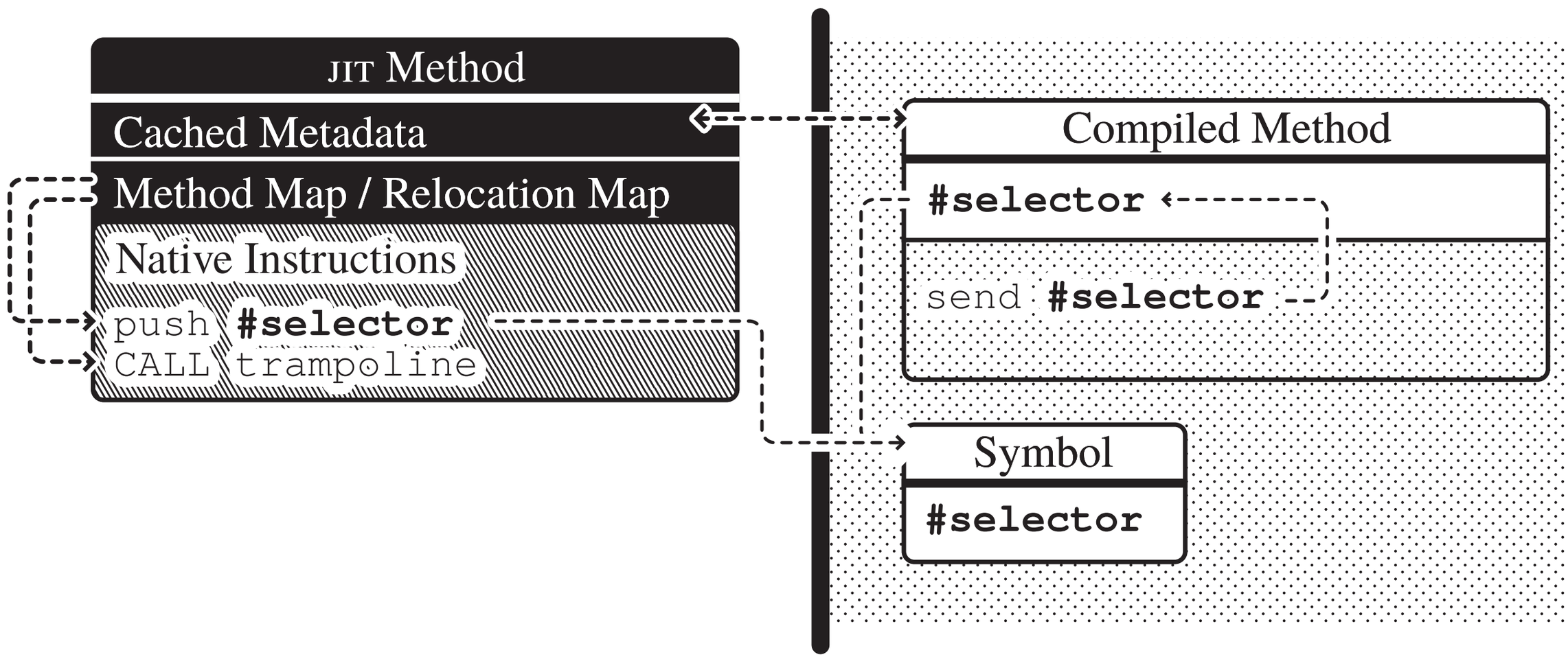


The GC is everywhere!

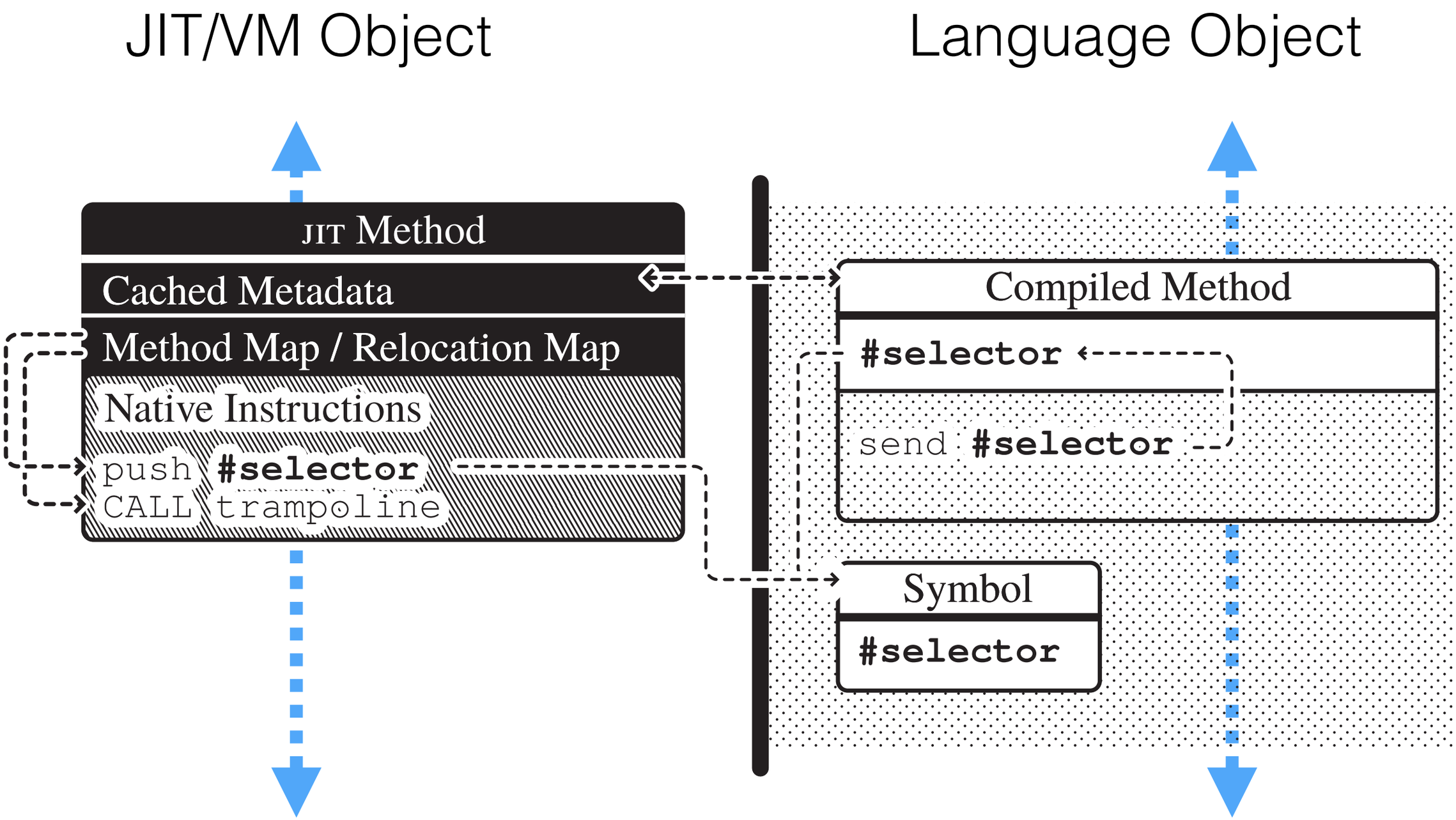
JIT Objects

JIT/VM Object

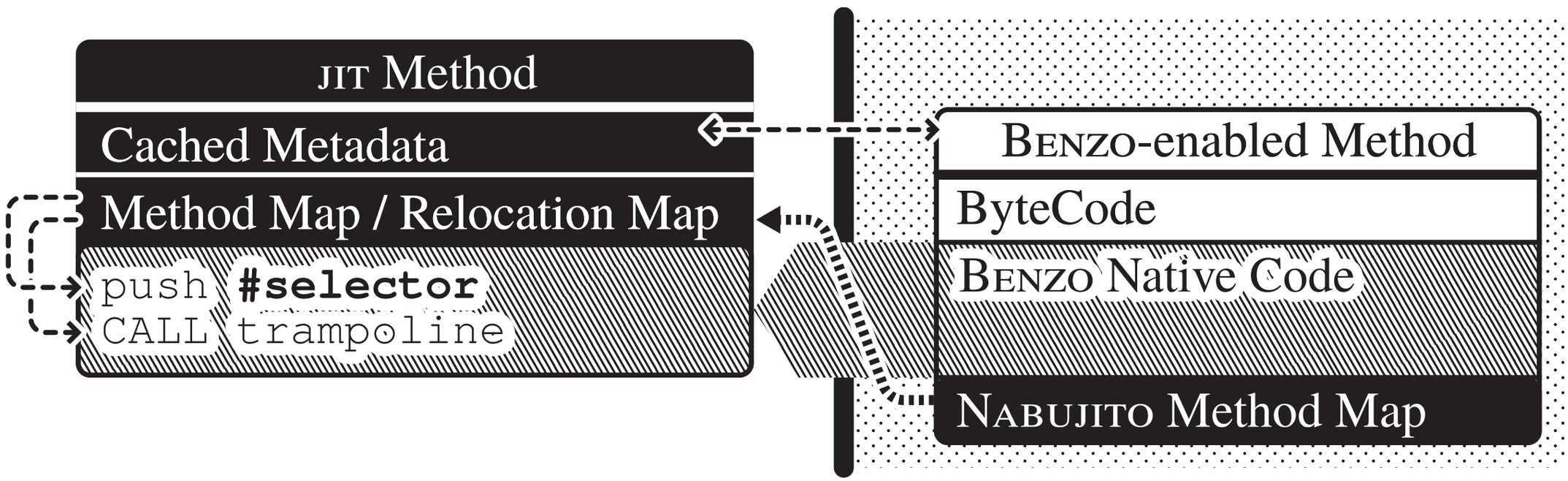
Language Object



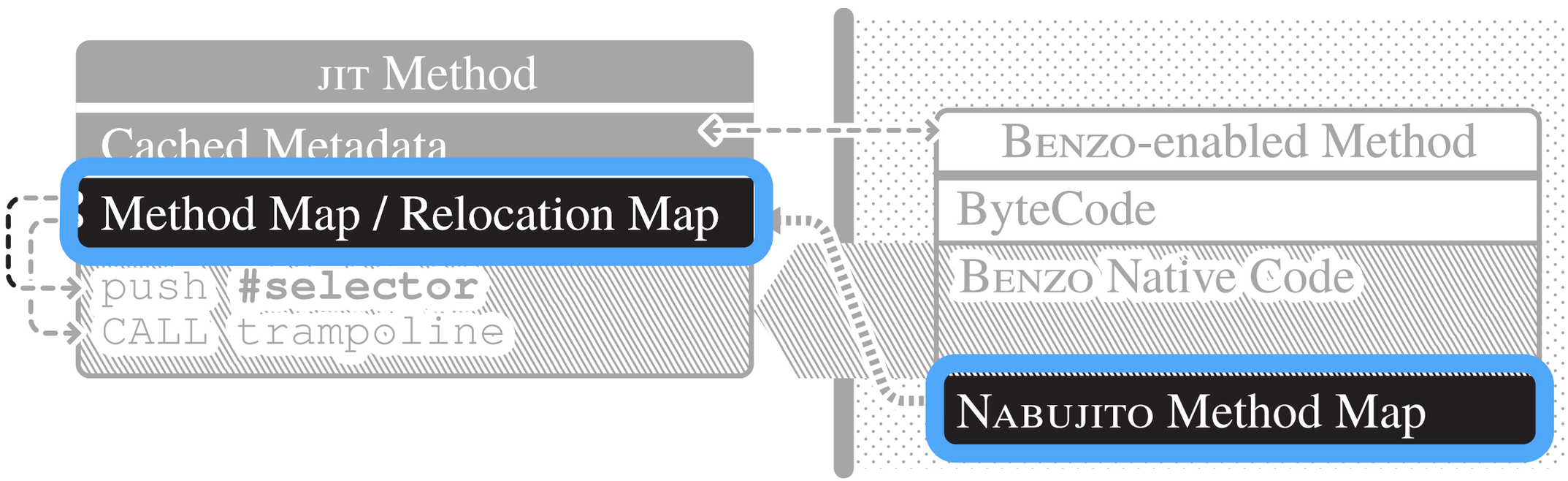
JIT Objects


JIT/VM Object
Language Object
JIT Method
Cached Metadata
Method Map / Relocation Map
Native Instructions
push #selector
CALL trampoline
Compiled Method
#selector
send #selector
Symbol
#selector

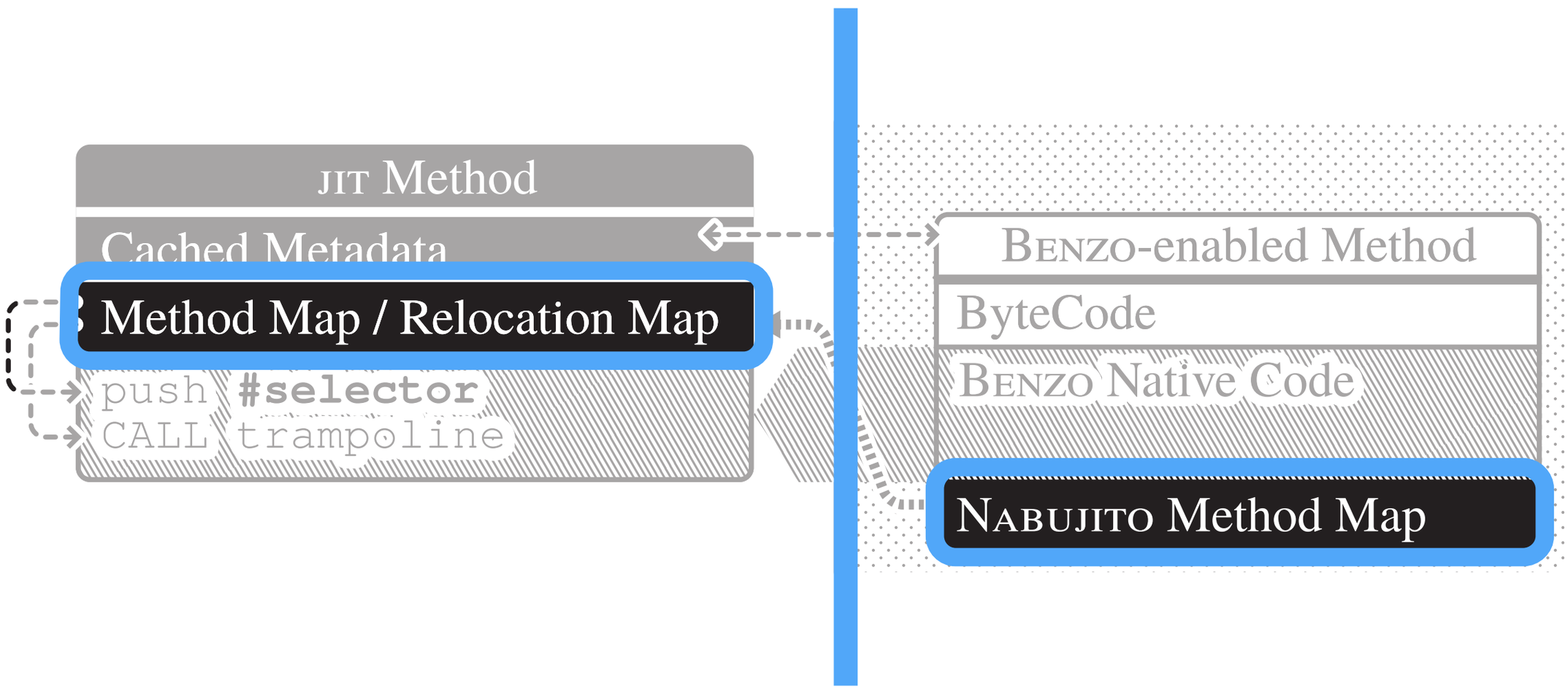
JIT GC–Information



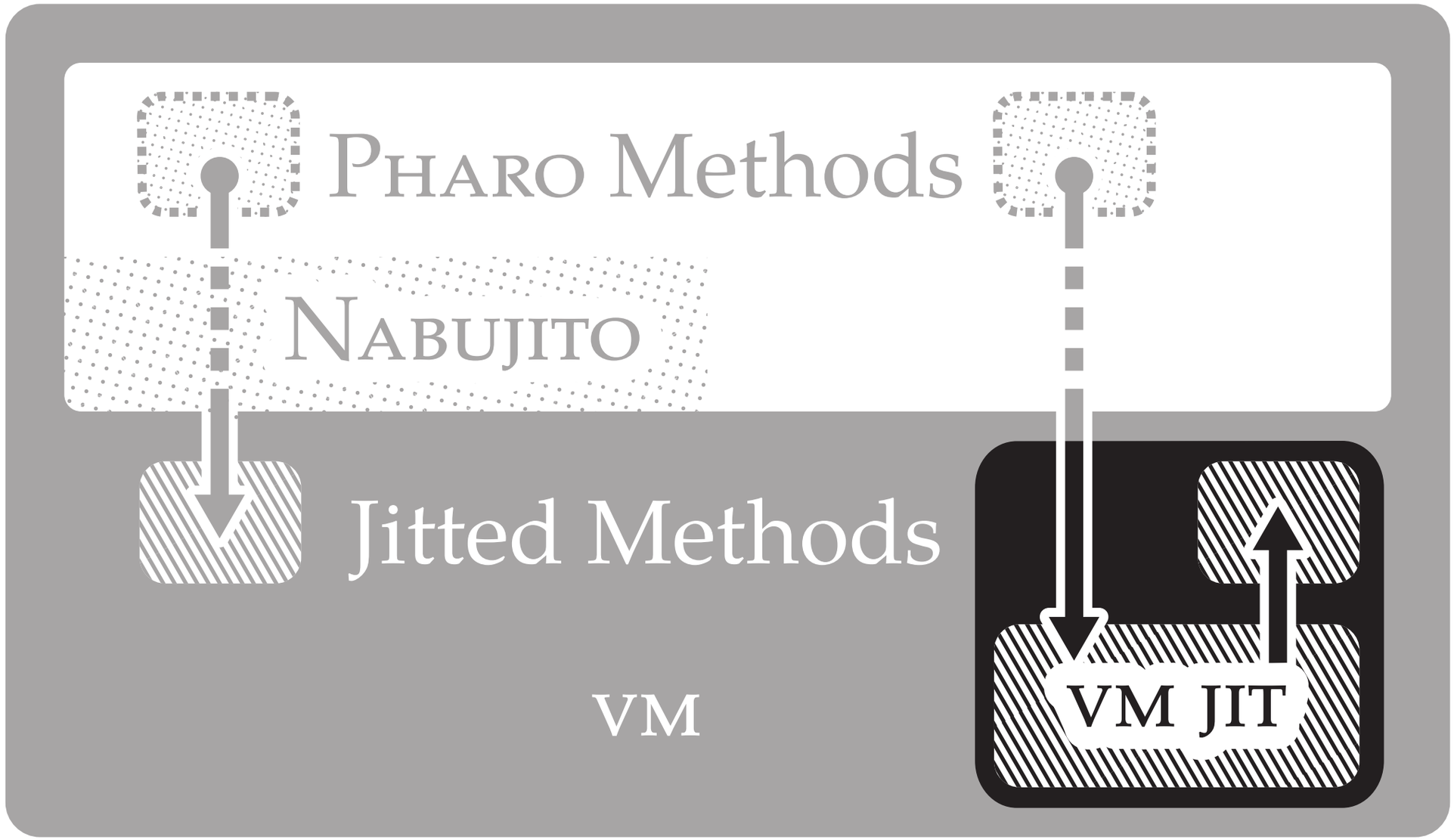
JIT GC–Information



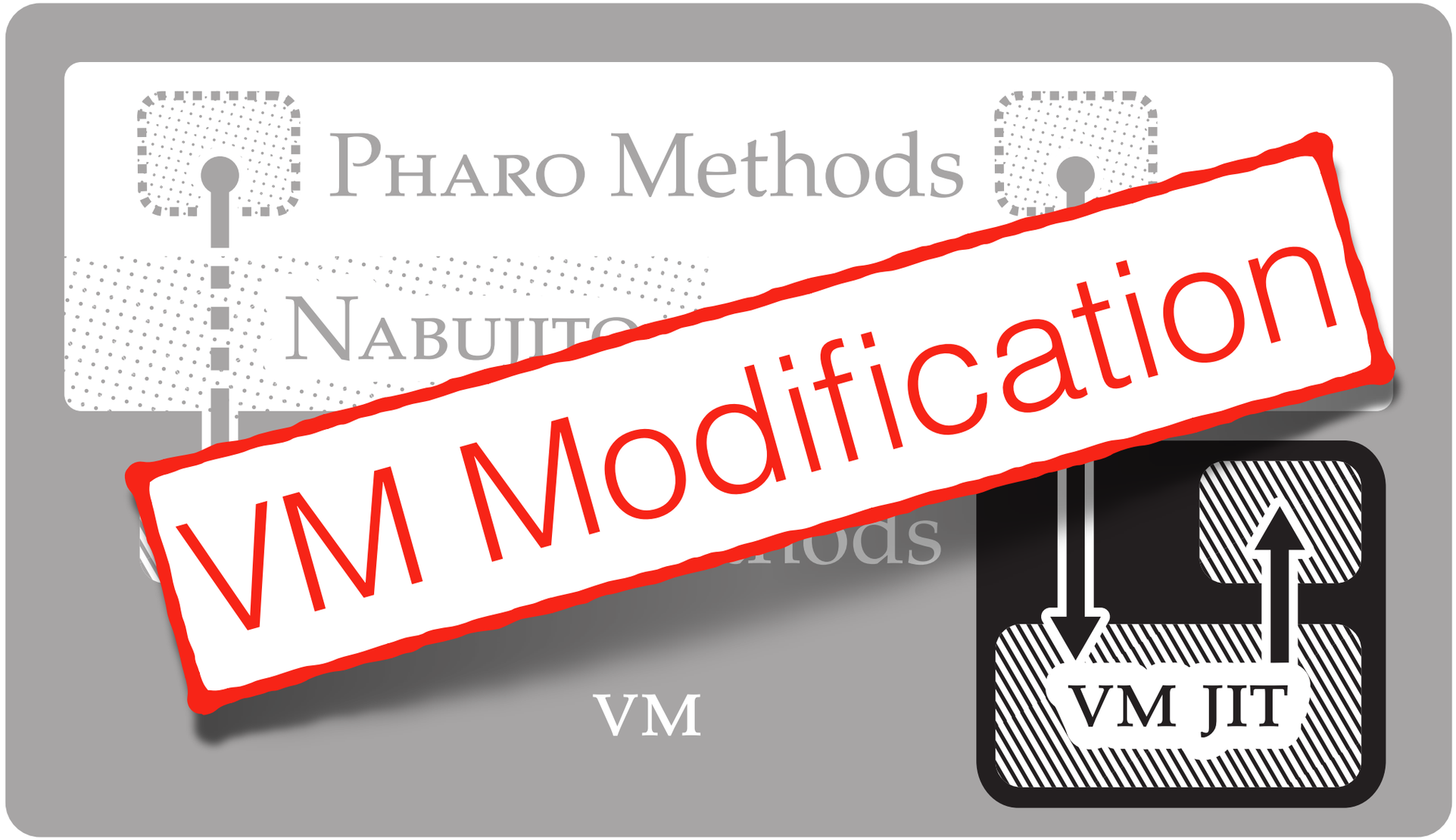
JIT GC–Information



Required JIT API



Required JIT API



Limitations

Missing JIT Interface to the VM

Missing Location Independent Code

Missing Access to VM Internal Objects

Summary

	Incremental	Dynamic Native Code	VM–level Intercession
HL LL Programming	+	–	–
Pinocchio	–	+	~
Klein	–	+	~
Benzo	**+**	**+**	**~**

Dissemination of Results

Publications

T. Verwaest, **C. Bruni,** M. Lungu and O. Nierstrasz
Flexible object layouts: enabling lightweight language extensions by intercepting slot access
In Proceedings of OOPSLA ’11

C. Bruni, S. Ducasse, I. Stasenko and L. Fabresse
Language-side Foreign Function Interfaces with NativeBoost
In Proceedings of IWST ’13

C. Bruni, G. Chari, S. Ducasse and I. Stasenko
BENZO: Reflective Low-level Programming
In Progress

G. Chari, **C. Bruni,** D.Garbervetsky, M. Denker and S. Ducasse
WATERFALL: On the Fly Primitive Generation
In Progress

Community Contributions

FFI in Pharo 2.0

First–class Layouts and Slots in Pharo 3.0

Vision
Related Work Analysis
Solution
Validation
Conclusion & Future Work

Results

BENZO Approach is feasible for a language like PHARO.

Missing low-level reification puts limits.

BENZO is a valid replacement for most VM-plugins.

Future Work

Language

High-level Interaction

Low-level Interaction

VM

High-level Interaction

Error Handling

Debug Mode

ASM Abstraction and Platform Independence

Low-level Interaction

VM Component Access

Low-level Reification

Summary

We validated high-level low-level programming in a dynamic context with BENZO.

BENZO is a step towards a self–aware VM.

BENZO replaces most VM–plugins.

Missing VM–level reification limits applications.