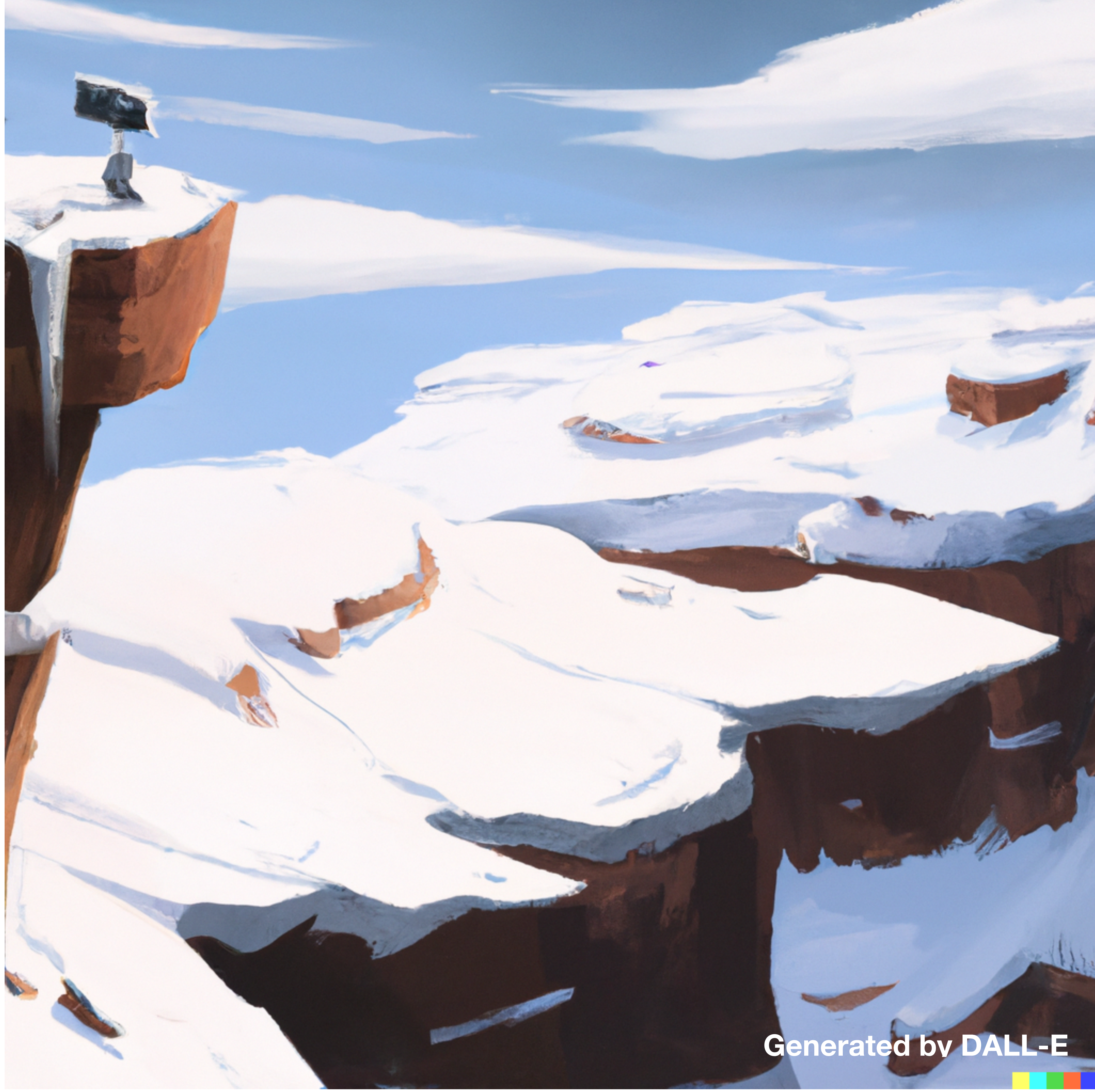
Instrumentation & the Pitfalls of Abstraction

ESUG - 2023 - Lyon

guillermo.polito@inria.fr
@guillep



* Supported by AlaMVic Action Exploratoire INRIA

First: About Me

guillermo.polito@inria.fr
@guillep

- **Keywords:** compilers, testing, test generation
- **Ph.D.:** Reflection, debloating, dynamic updates
- **Interests:** tooling, benchmarking, 日本語, board games, concurrency

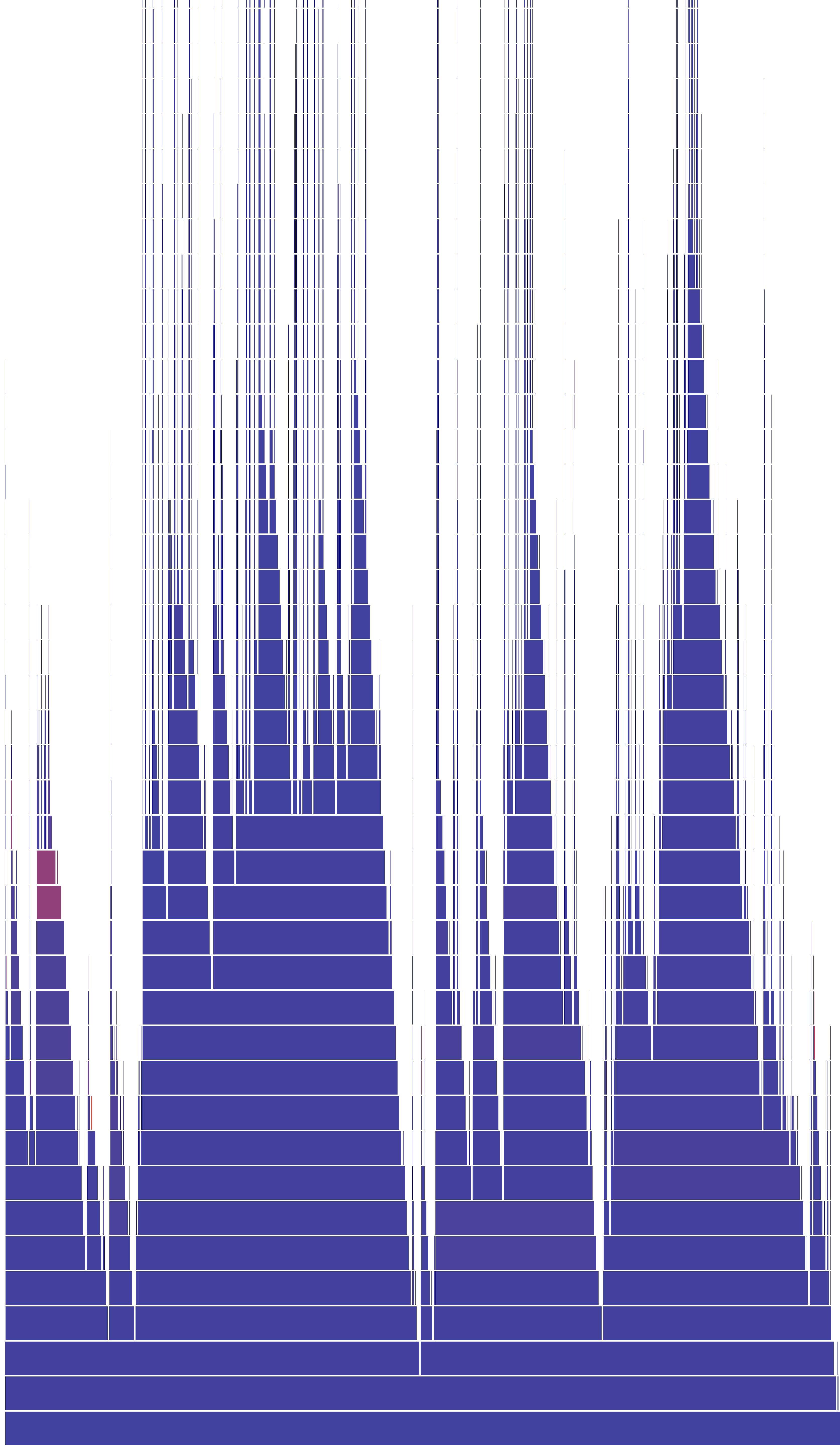
Talk to me!

Or: guillermo.polito@inria.fr

Evref

Building Dynamic Analyses

- Dynamic call graphs
- Code coverage
- Profilers
  - Time
  - Number of calls



Method Wrappers, Objects as Methods

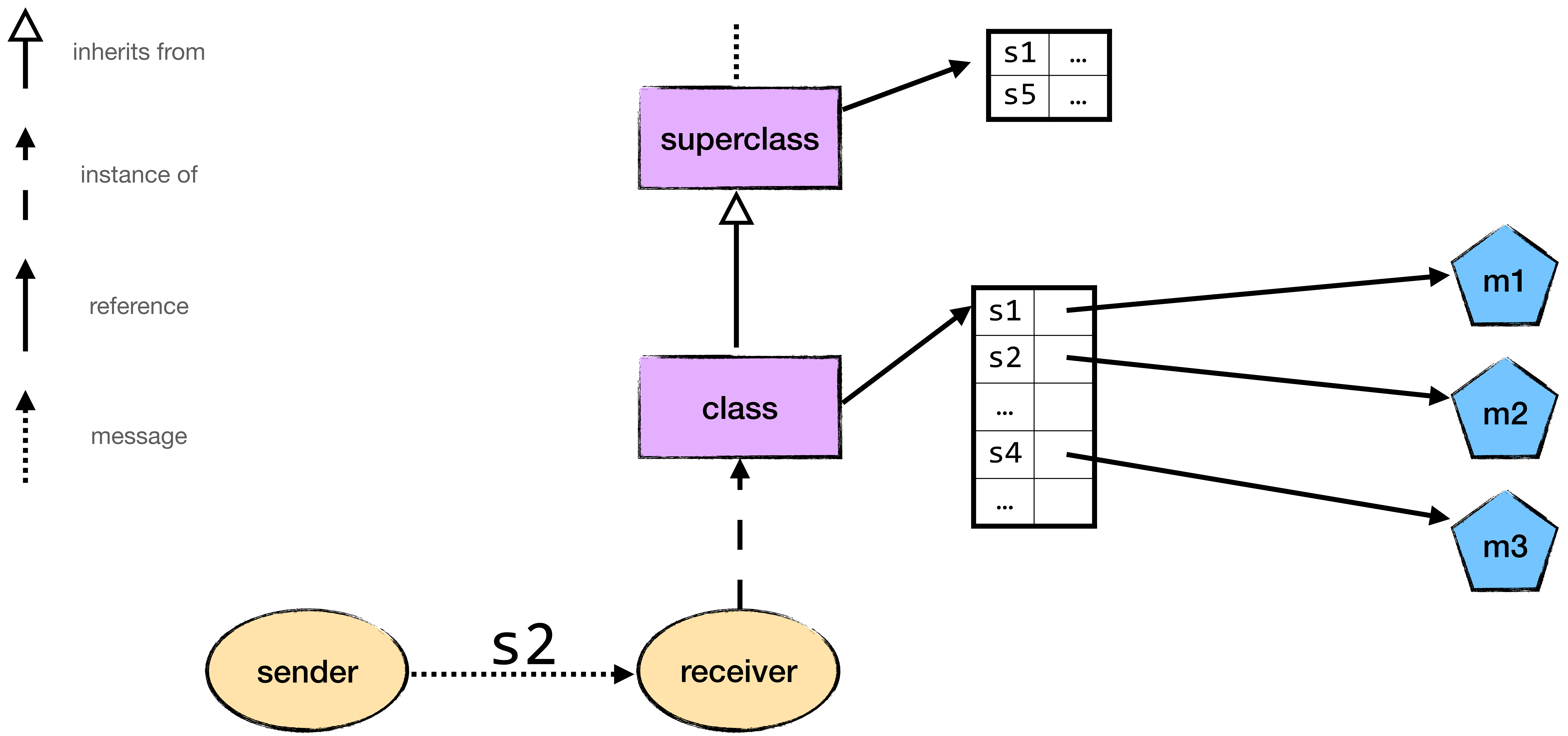
Wrappers to the Rescue

John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts

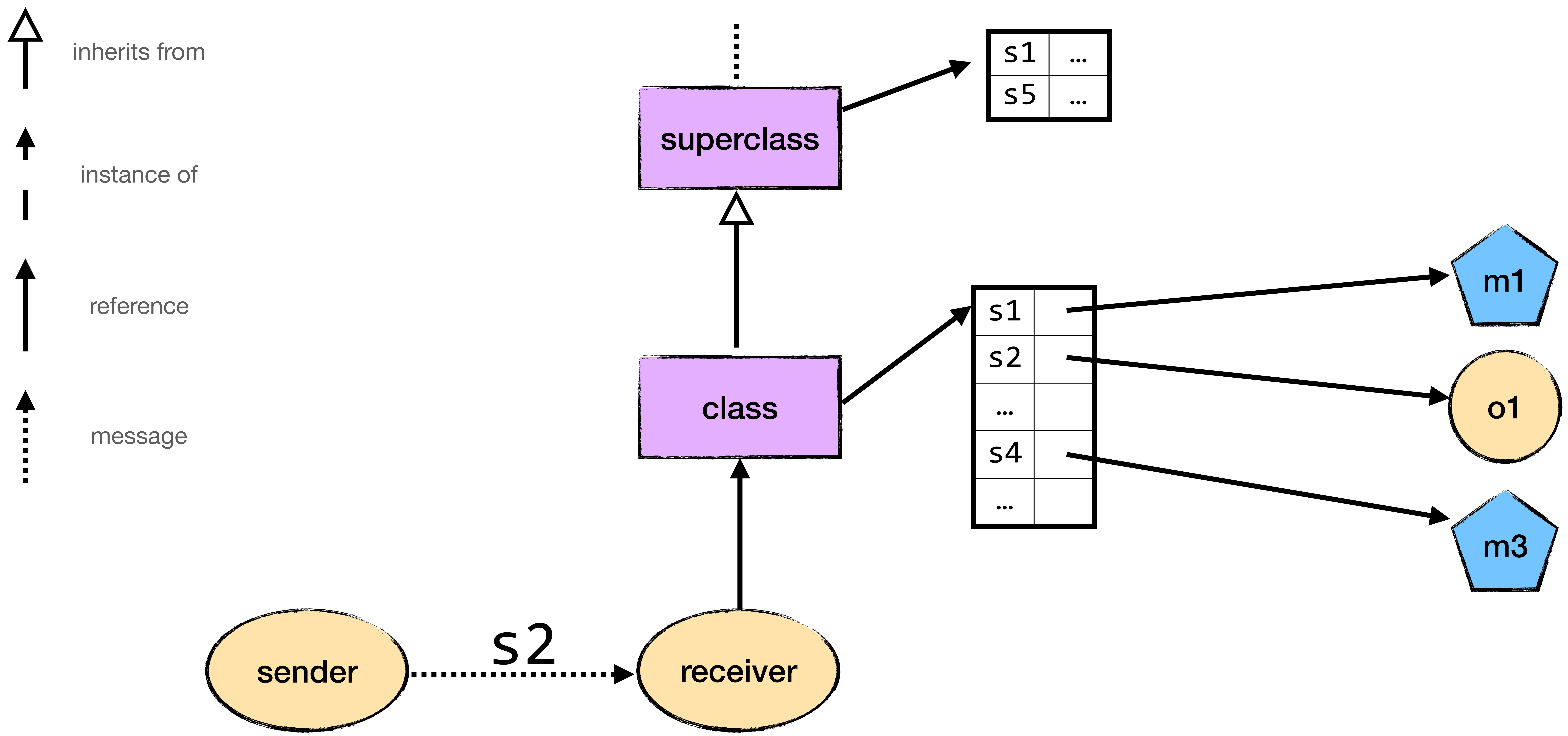
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{brant, foote, johnson, droberts}@cs.uiuc.edu

Abstract. Wrappers are mechanisms for introducing new behavior that is executed before and/or after, and perhaps even in lieu of, an existing method. This paper examines several ways to implement wrappers in Smalltalk, and compares their performance. Smalltalk programmers often use Smalltalk's lookup failure mechanism to customize method lookup. Our focus is different. Rather than changing the method lookup process, we modify the method objects that the lookup process returns. We call these objects *method wrappers*. We have used method wrappers to construct several program analysis tools: a coverage tool, a class collaboration tool, and an interaction diagramming tool. We also show how we used method wrappers to construct several extensions to Smalltalk: synchronized methods, assertions, and multimethods. Wrappers are relatively easy to build in Smalltalk because it was designed with reflective facilities that allow programmers to intervene in the lookup process. Other languages differ in the degree to

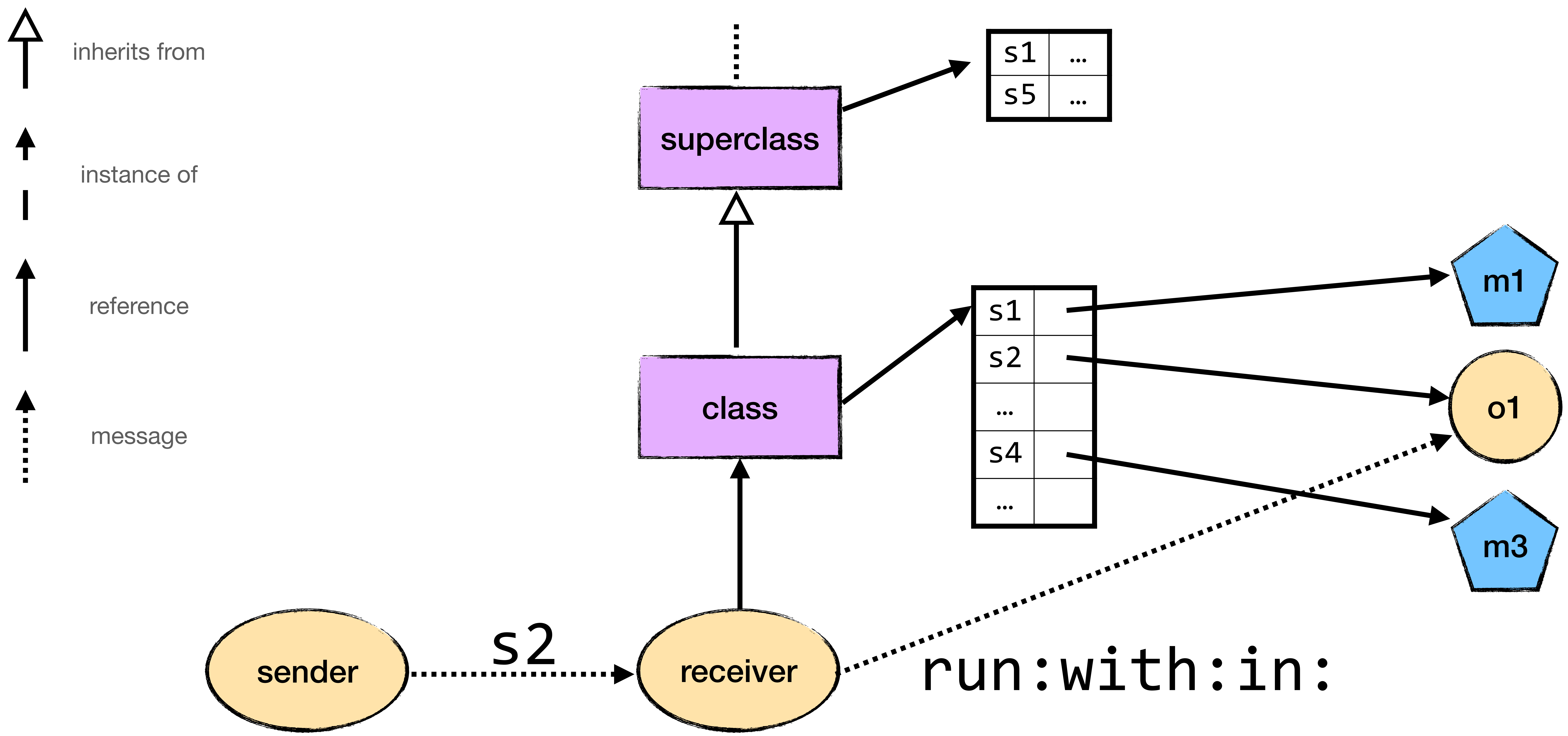
Remember Method Lookup



Objects as methods



Objects as methods + `run:with:in:`

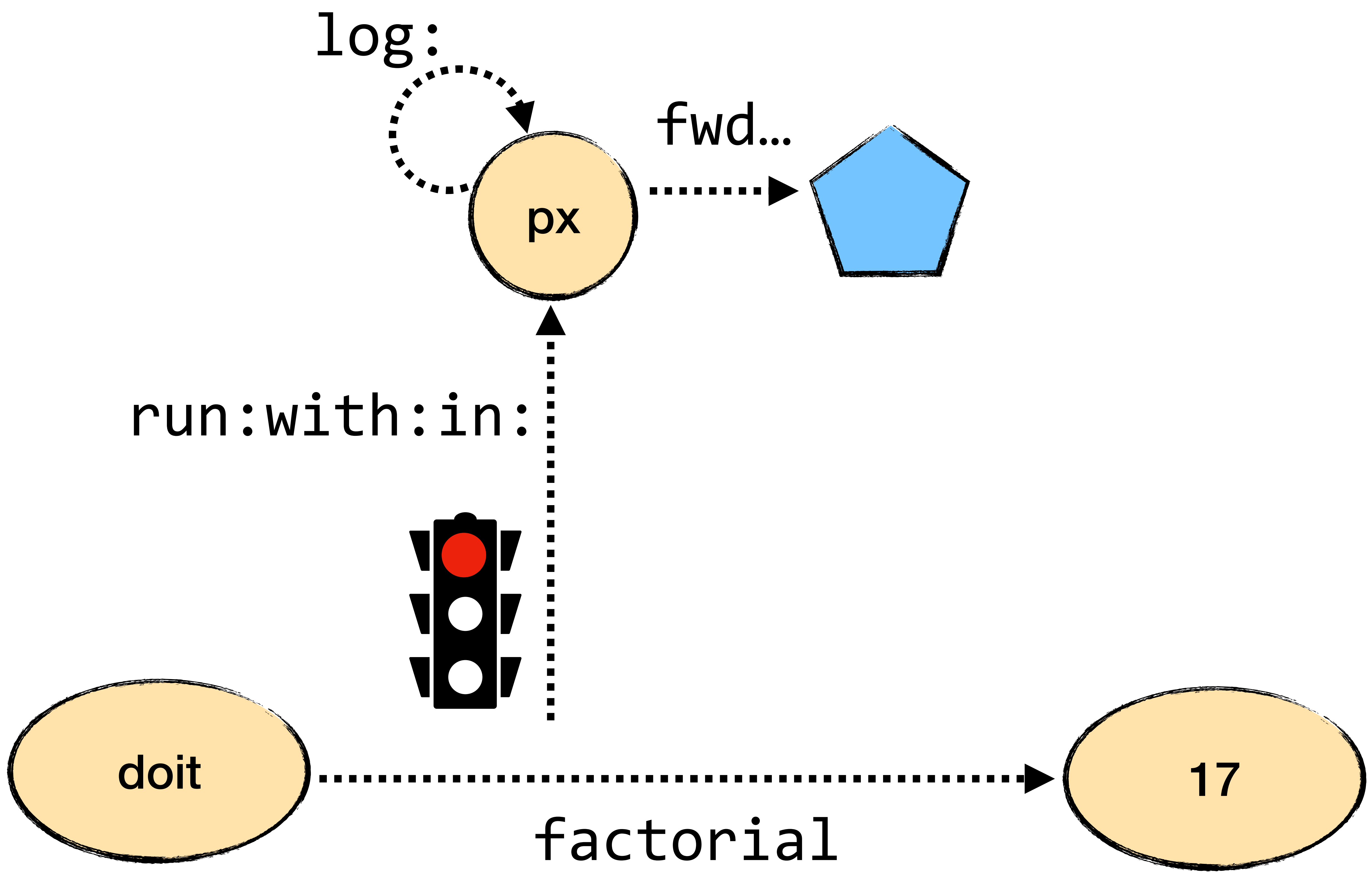


How far can we get with `run:with:in:` ?

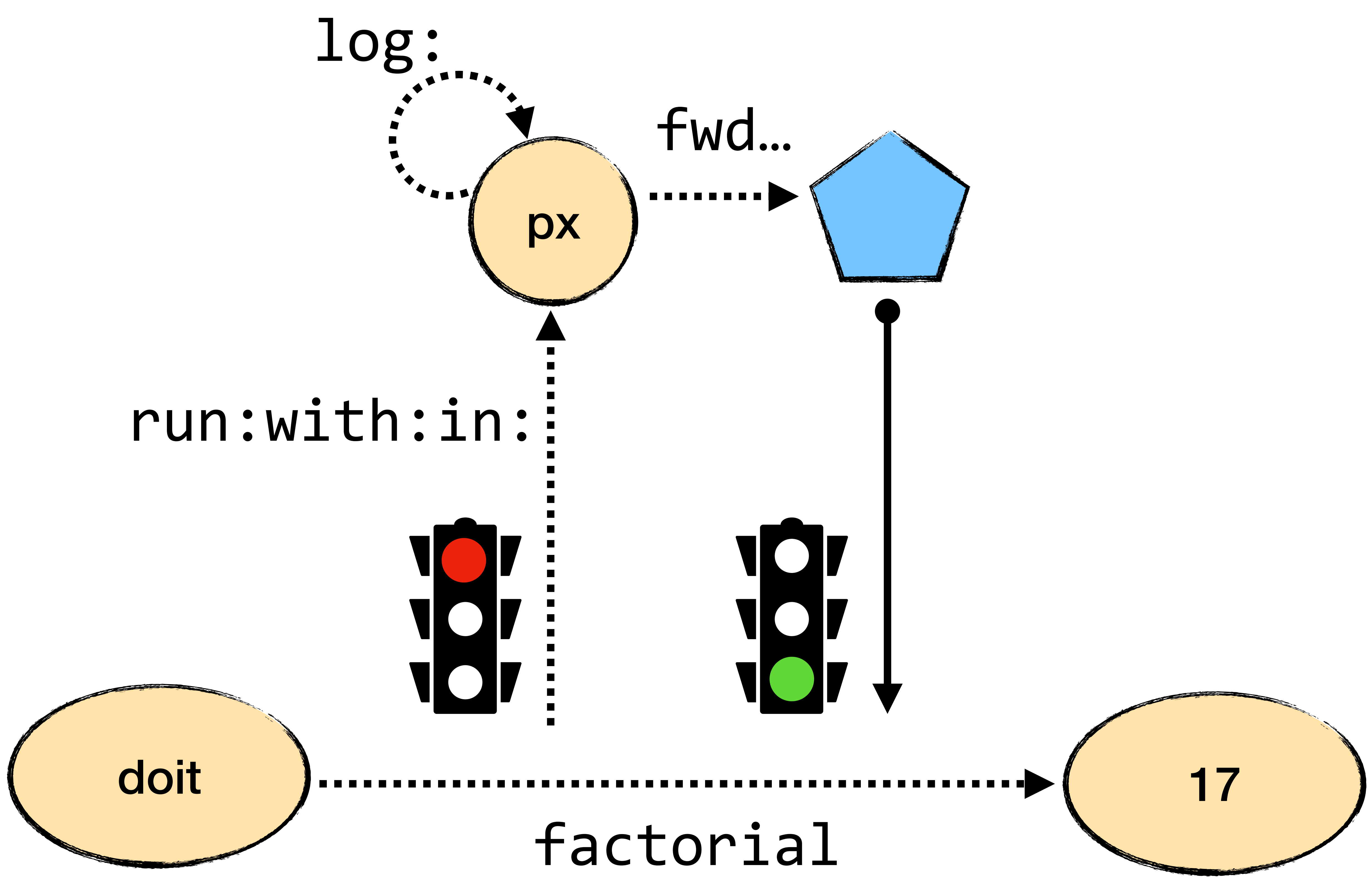
A First Method Proxy

```
run: aSelector with: anArrayOfObjects in: aReceiver
    | result |

    self logBefore: aSelector.

    result := self
        forwardMethod: originalMethod
        withReceiver: aReceiver
        withArguments: anArrayOfObjects.

    self logAfter: aSelector.

    ^ result
```

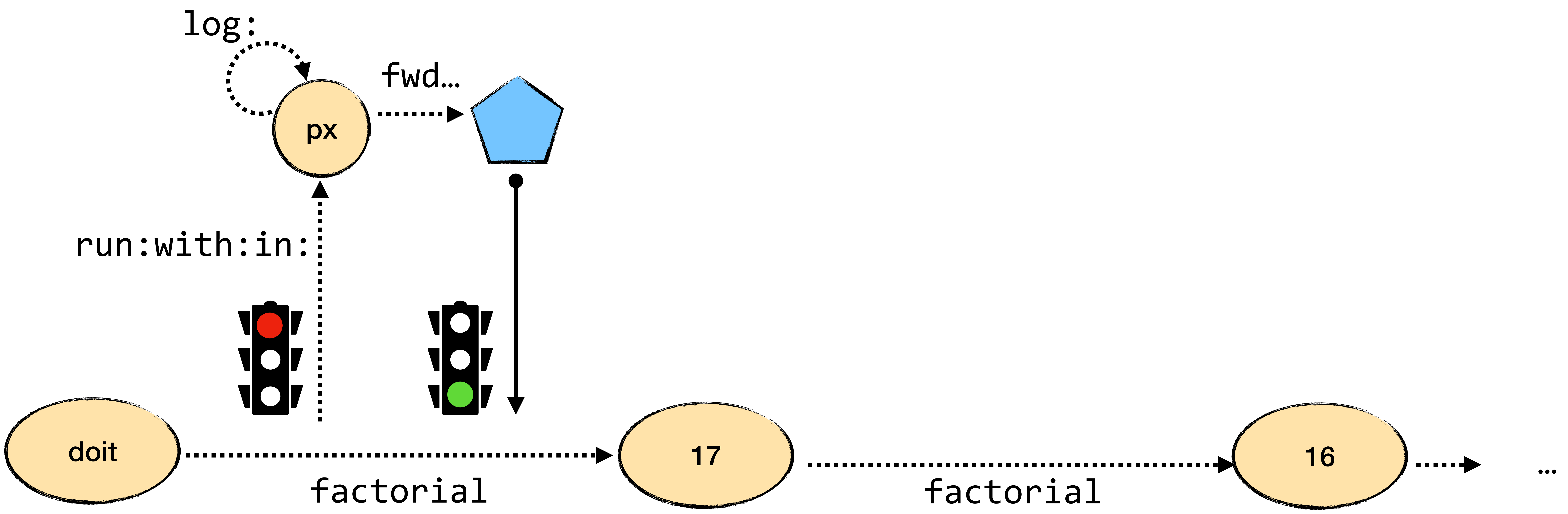
Let’s instrument factorial



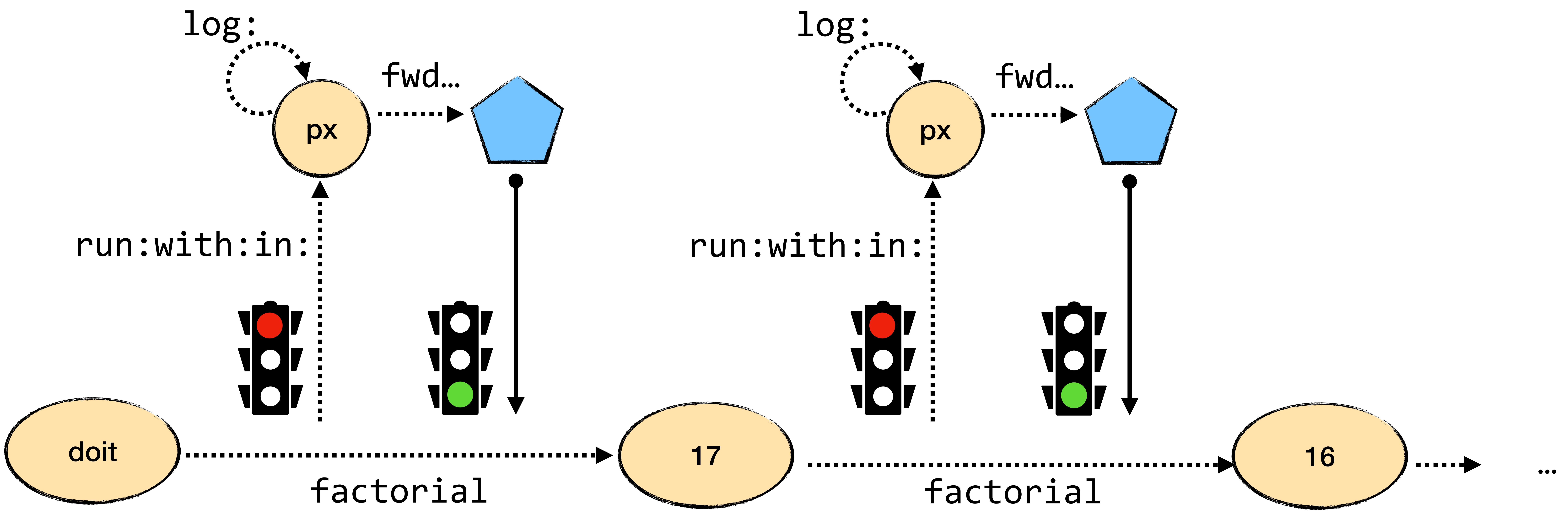
Let’s instrument factorial



Let’s instrument factorial

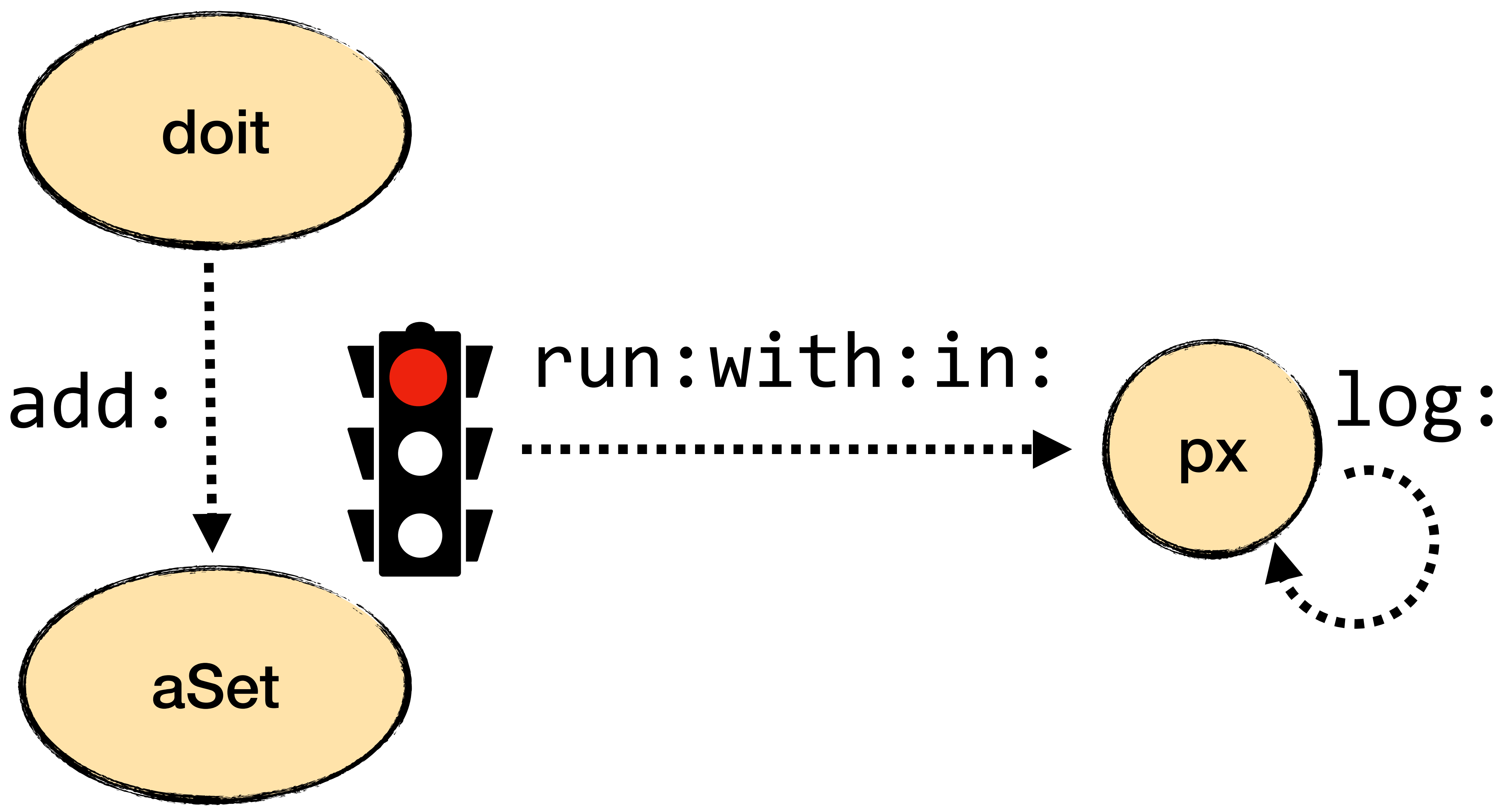


Let's instrument factorial

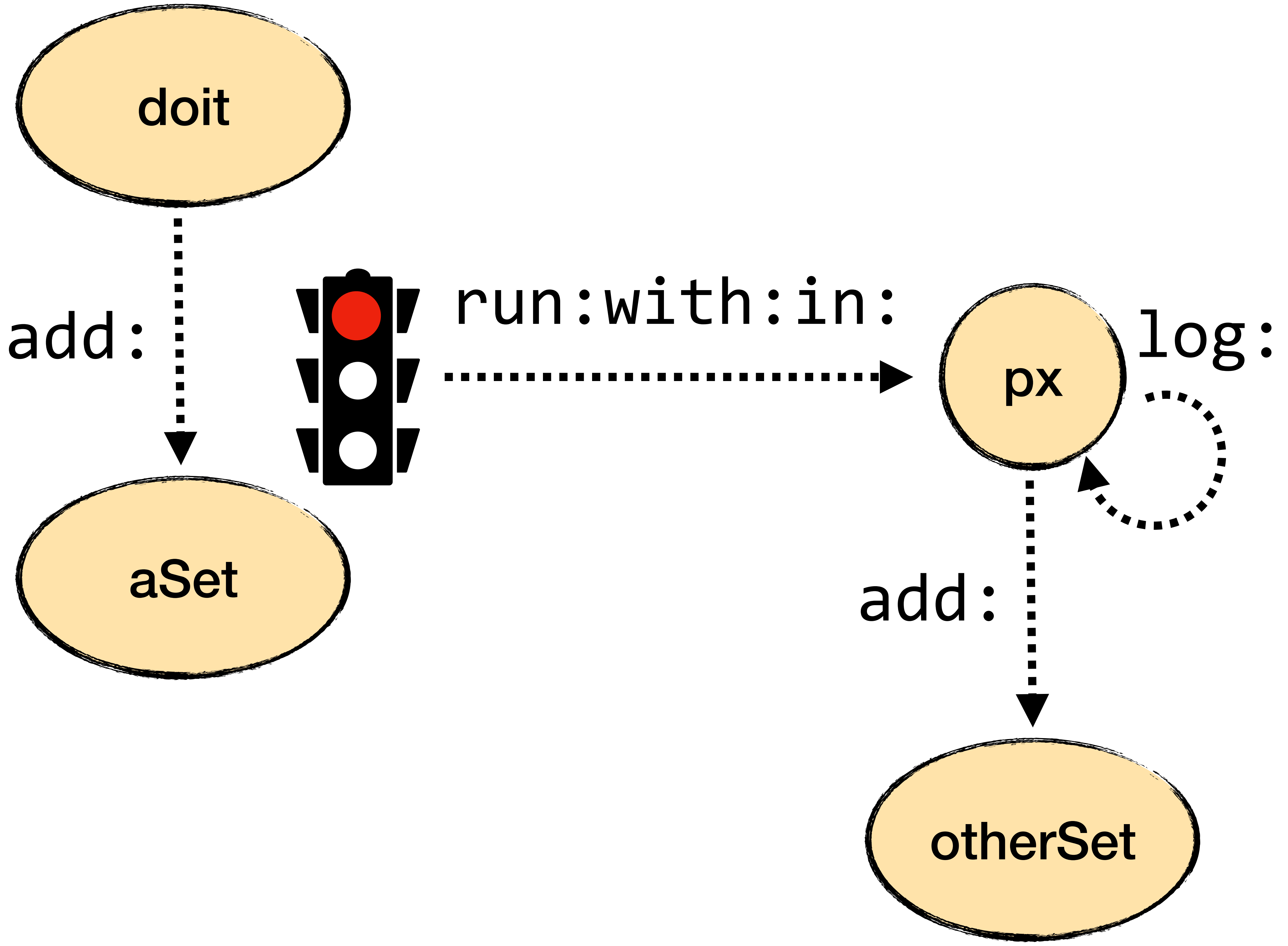


Let’s get a bit more hardcore

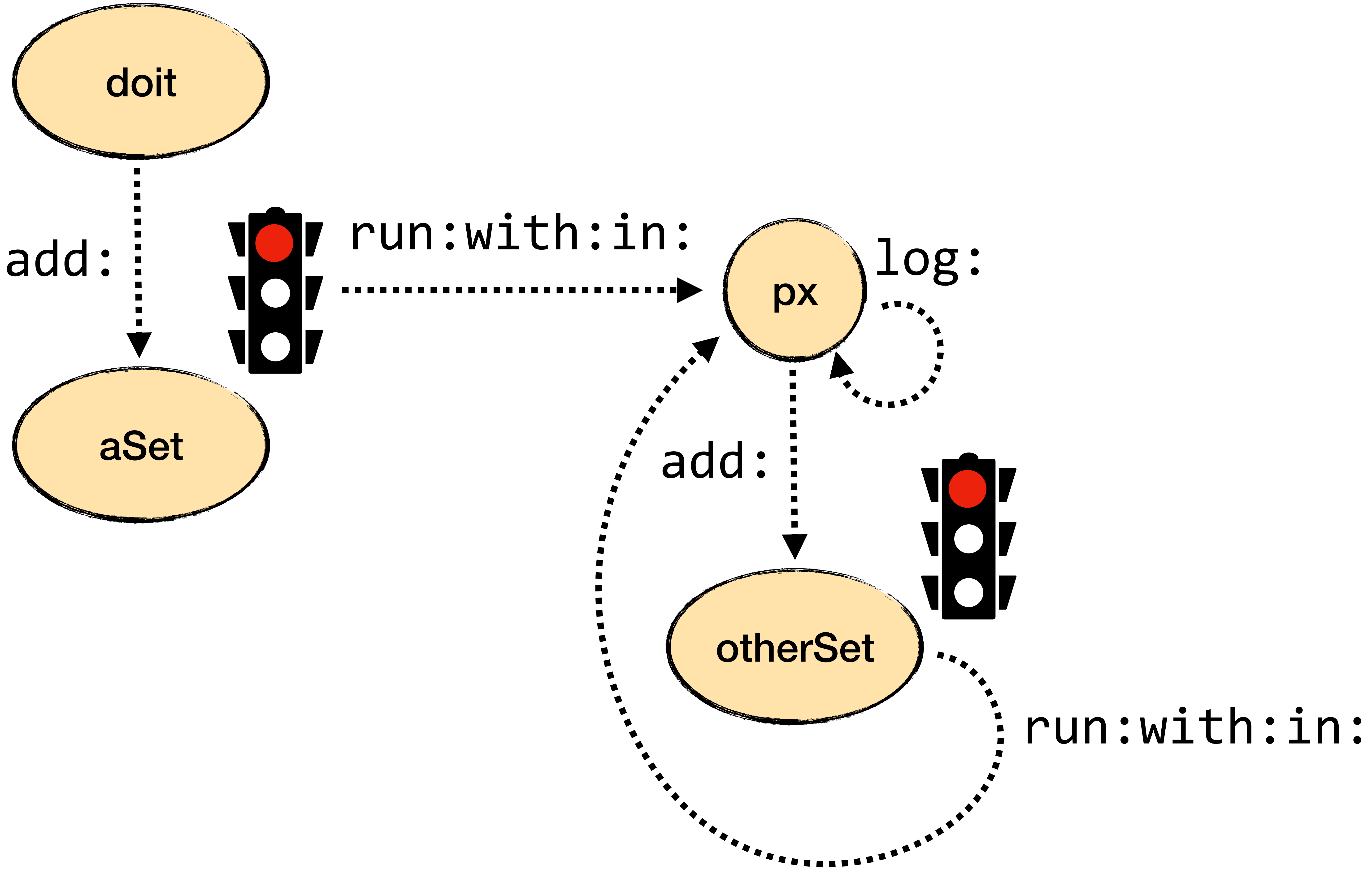
Instrumenting Set>>#add:



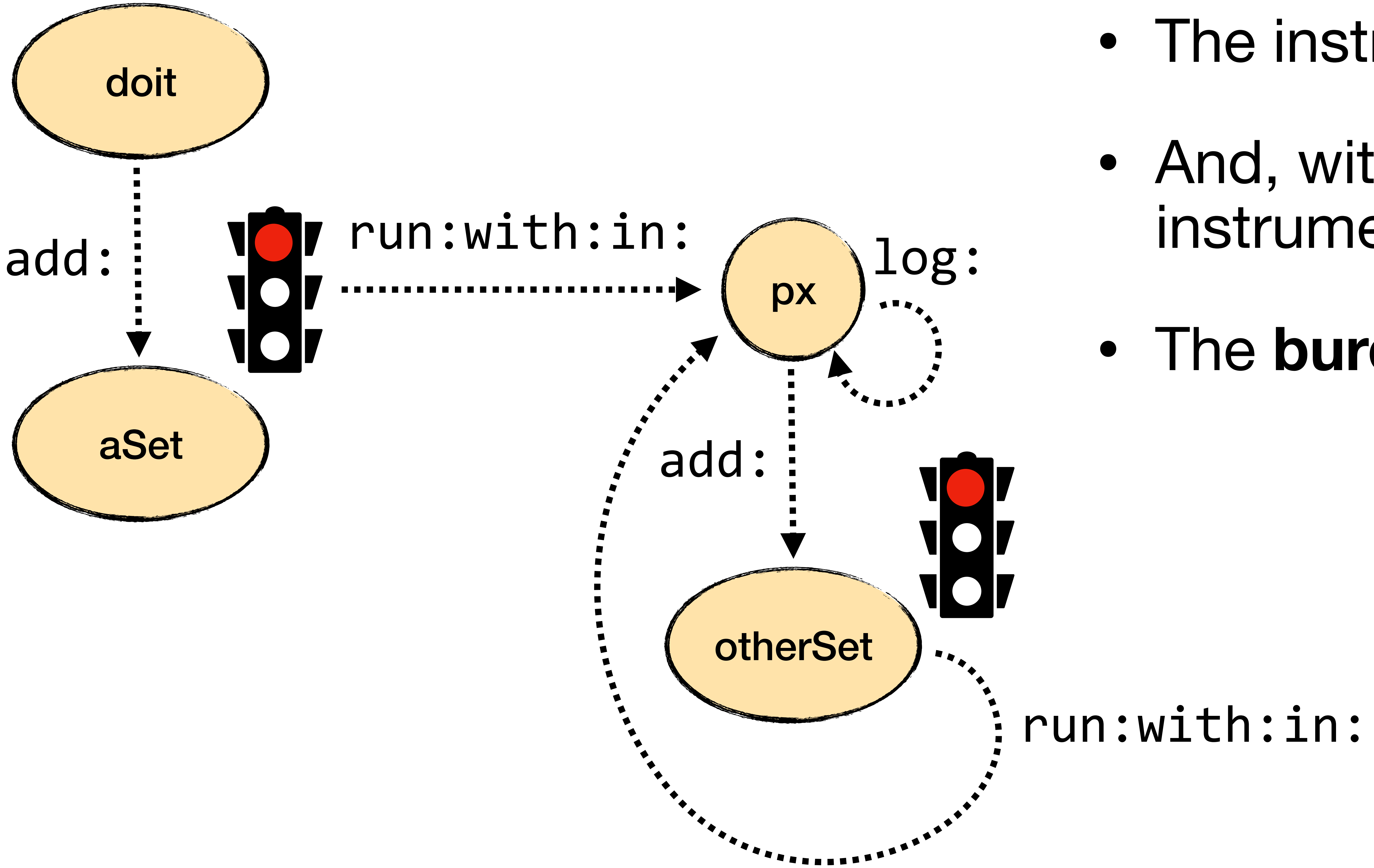
Instrumenting Set>>#add:



Instrumenting Set>>#add:

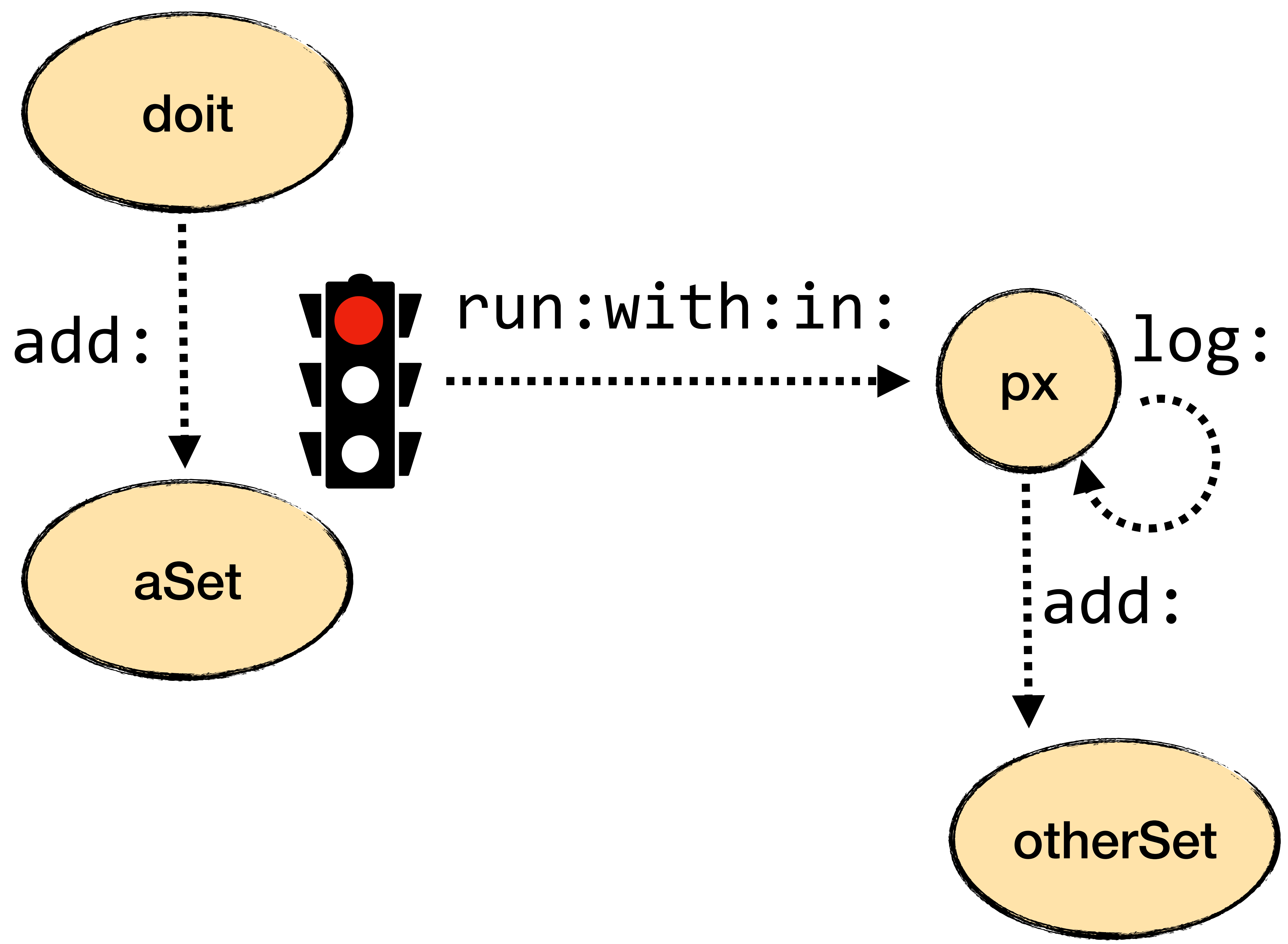


Meta-Recursions

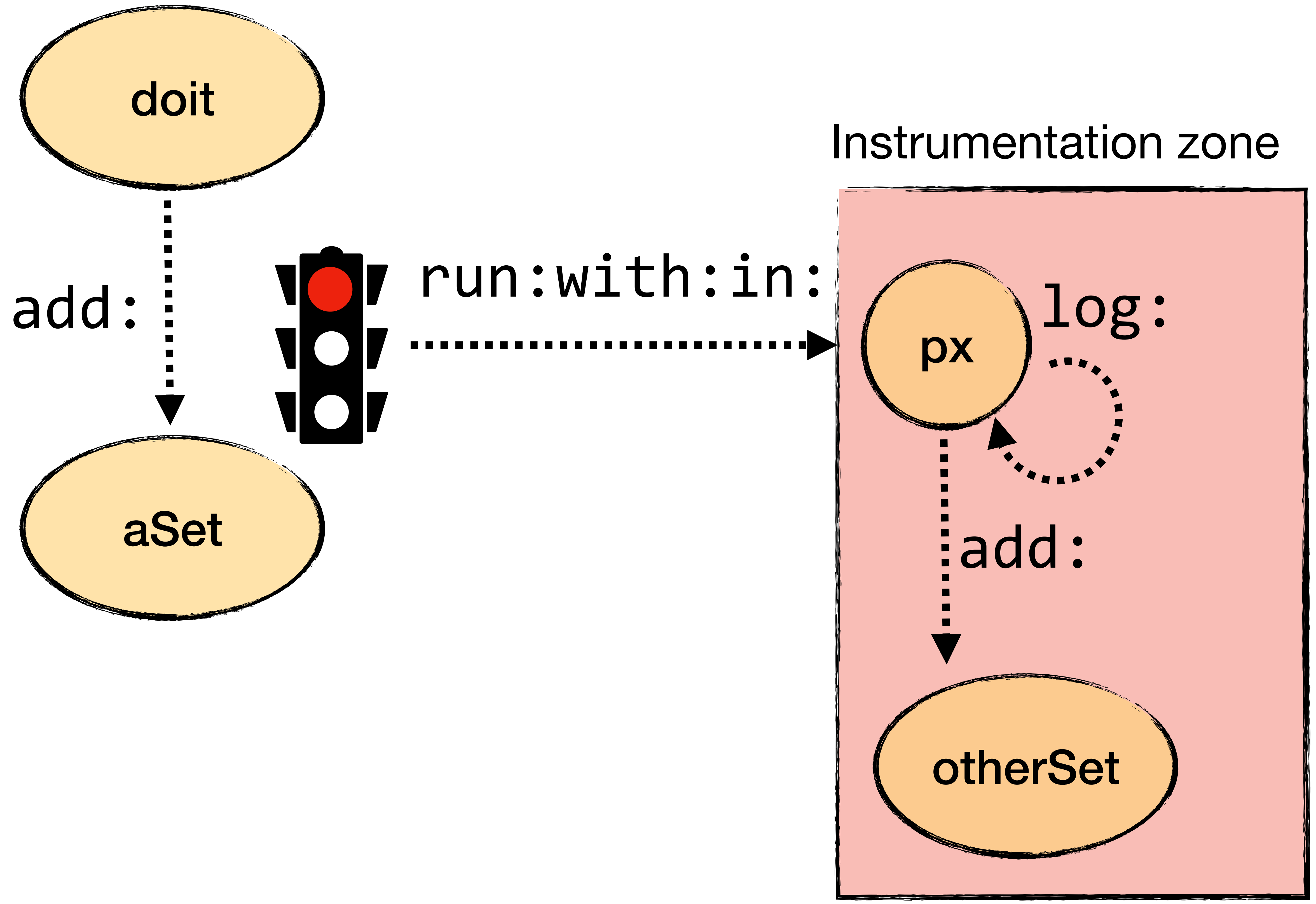


- The instrumentation gets instrumented!
- And, with more complex instrumentation, more difficult to debug
- The **burden:** on the developer

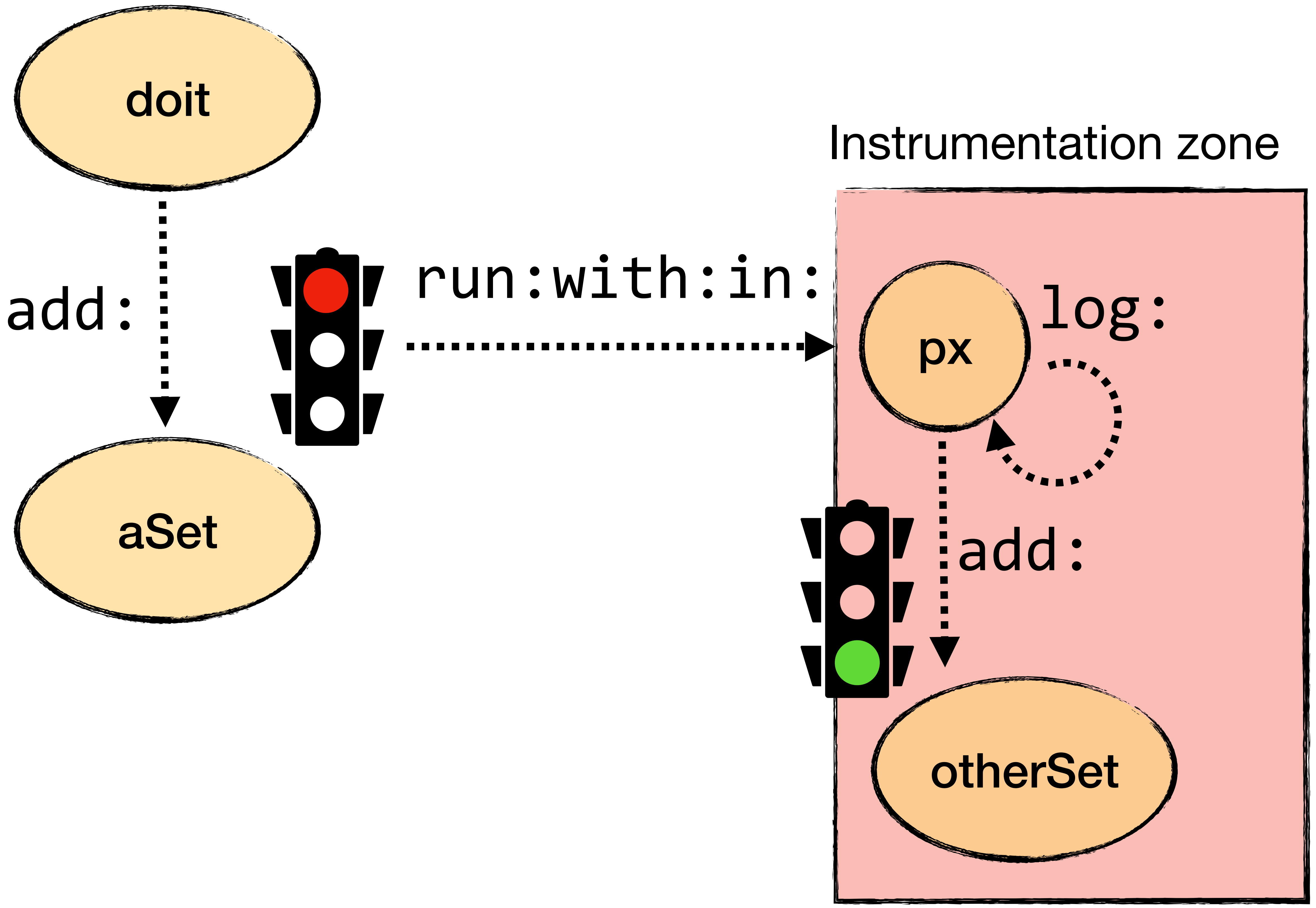
Solving Meta-Recursions



Solving Meta-Recursions



Solving Meta-Recursions



And That’s not All

- Stack unwind (non-local returns, exceptions) pass around the `logAfter:`
- Concurrent access to our instrumentation zone?
  - lose logs
  - break the instrumentation
- Maybe we can do some concessions: e.g., do not proxy the proxy…

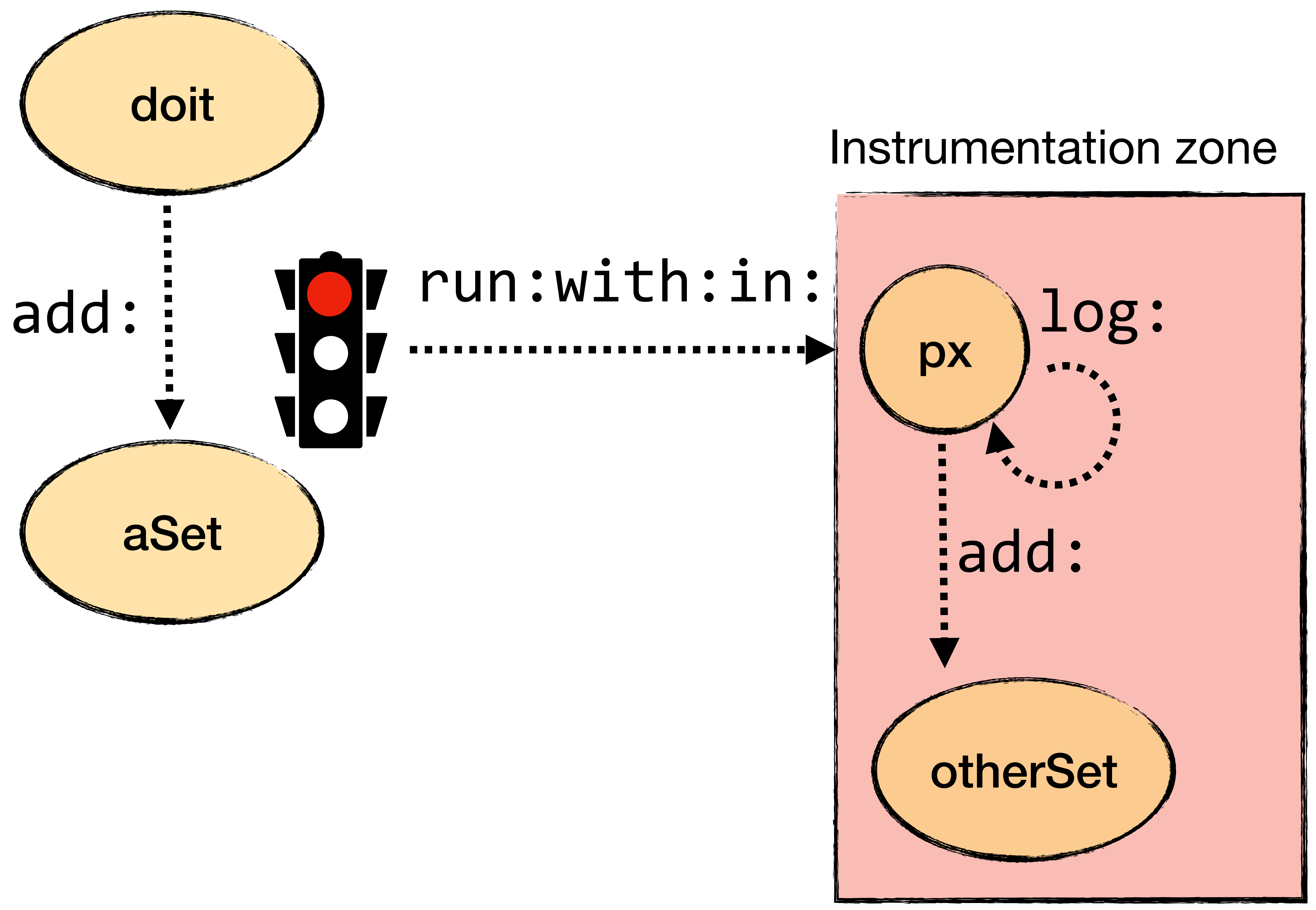
This burden, is on the developer

The *Cost* of Missing Abstraction

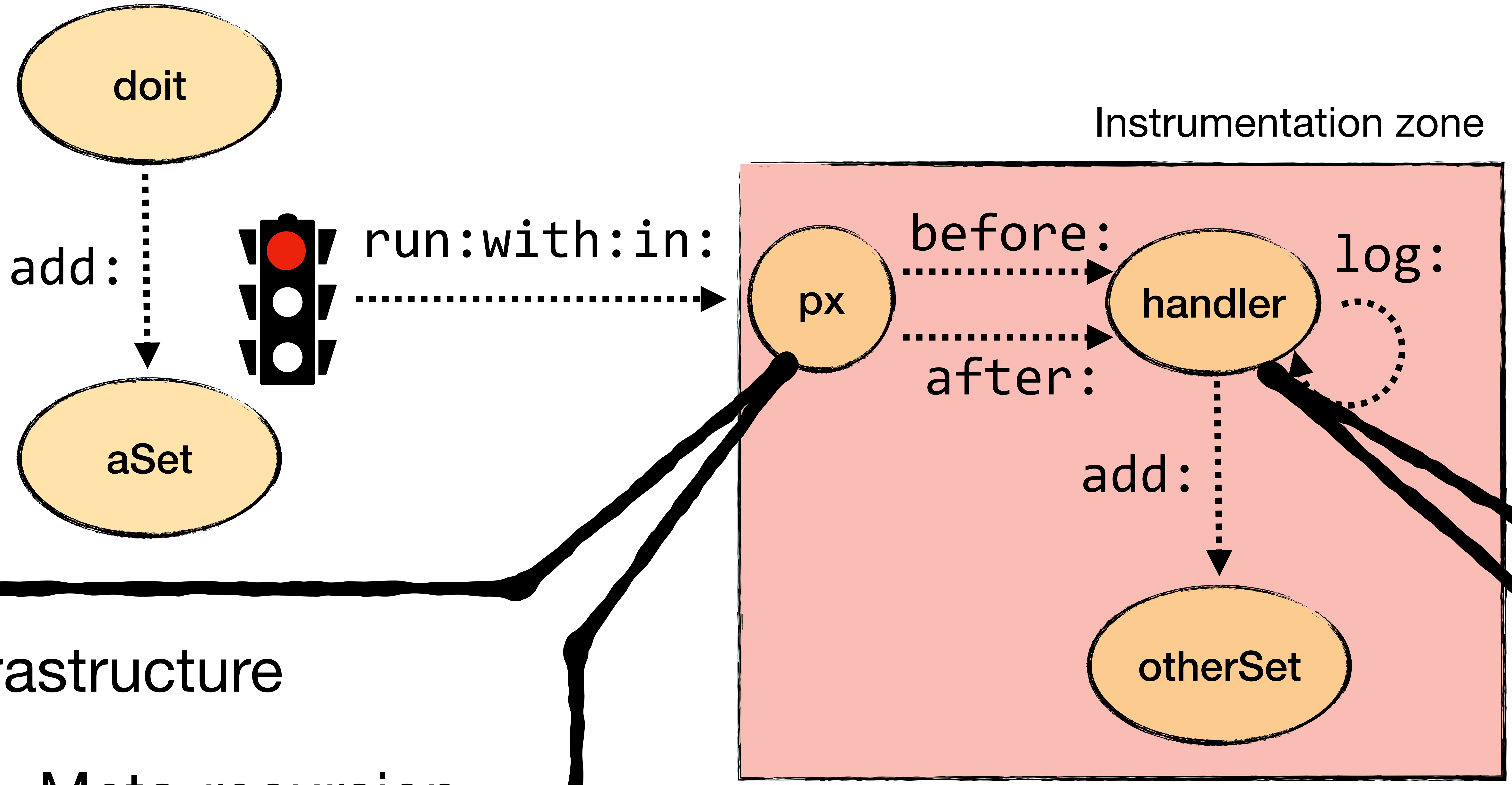
- The language gives us only **low-level instrumentation** hooks
  - `#run:with:in:`
  - `#doesNotUnderstand:`
  - `#cannotInterpret:`
- i.e., they are at the *wrong level of abstraction* for proper instrumentation

Covering the GAP, is on the developer

The Proxy We Have



The *Stratified* Proxy We Want



Infrastructure

- Meta-recursion
- Concurrency

User concern

- logging?
- analysis?

Stratified Proxies

Proxies: Design Principle
Object-oriented Interce

Tom Van Cutsem *

Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
tvcutsem@vub.ac.be

Abstract

Proxies are a powerful approach to implement meta-objects in object-oriented languages without having to resort to metacircular interpretation. We introduce such a meta-level API based on proxies for Javascript. We simultaneously introduce a set of design principles that characterize such APIs in general, and compare similar APIs of other languages in terms of these principles. We highlight how principled proxy-based APIs improve code robustness by avoiding interference between base and meta-level code that occur in more common reflective intercession mechanisms.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented languages

trol, ent v **Virtual** out th addre (emu jects futur

The c of a prox tion 4), ples that metapro

Efficient Proxies in Smalltalk

Mariano Martinez Peck[1 2] Noury Bouraqadi[2] Marcus Denker[1]
Stéphane Ducasse[1] Luc Fabresse[2]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1
[2]Université Lille Nord de France, Ecole des Mines de Douai

marianopeck@gmail.com, {stephane.ducasse,marcus.denker}@inria.fr,
{noury.bouraqadi,luc.fabresse}@mines-douai.fr

Abstract

A proxy object is a surrogate or placeholder that controls access to another target object. Proxy objects are a widely used solution for different scenarios such as remote method invocation, future objects, behavioral reflection, object databases, inter-languages communications and bindings, access control, lazy or parallel evaluation, security, among others.

Most proxy implementations support proxies for regular objects but they are unable to create proxies for classes or methods. Proxies can be complex to install, have a significant overhead, be limited to certain type of classes, etc. Moreover, most proxy implementations are not stratified at all and there is no separation between proxies and handlers.

systems [3, 20], future objects [23], behavioral reflection [10, 15, 29], aspect-oriented programming [16], wrappers [6], object databases [7, 19], inter-languages communications and bindings, access control and read-only execution [1], lazy or parallel evaluation, middlewares like CORBA [13, 17, 28], encapsulators [22], security [27], among others.

Most proxy implementations support proxies for regular objects (instances of common classes) only. Some of them, *e.g.*, Java Dynamic Proxies [11, 14] even requires that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages.

Creating uniform proxies for not only regular objects, but also for classes and methods has not been considered.

Safe Method Proxies + Exact Method Profiler

- **Method Proxies:** https://github.com/pharo-contributions/MethodProxies
- **Method Profiler:** https://github.com/pharo-contributions/MethodProfiler

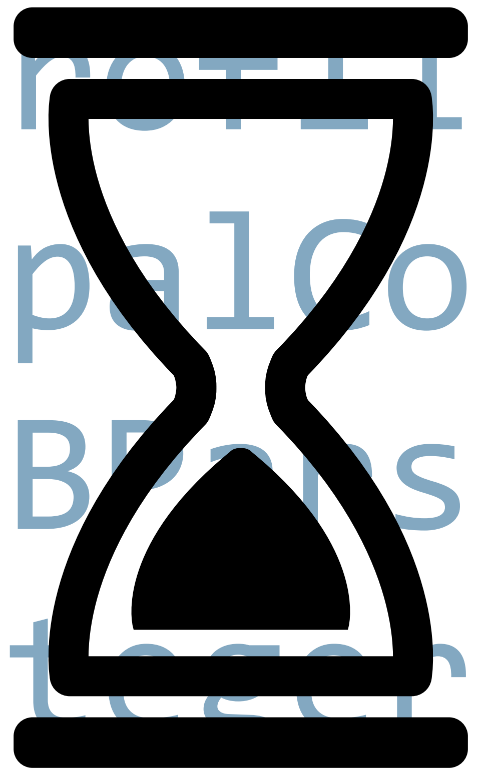
- + common instrumentation layer between proxies and meta-links !

Let’s get a bit more hardcore ***again***

Let’s Instrument the Compiler

```
prf := PrfMethodProfiler new.
prf addPackage: OpalCompiler package.
prf addPackage: RBParser package.
prf profile: [ Integer recompile ].
```

Let’s Instrument the Compiler



```
prf := PrfMethodProfiler new.
prf addPackage: OpalCompiler package.
prf addPackage: RBParser package.
prf profile: [ Integer recompile ].
```

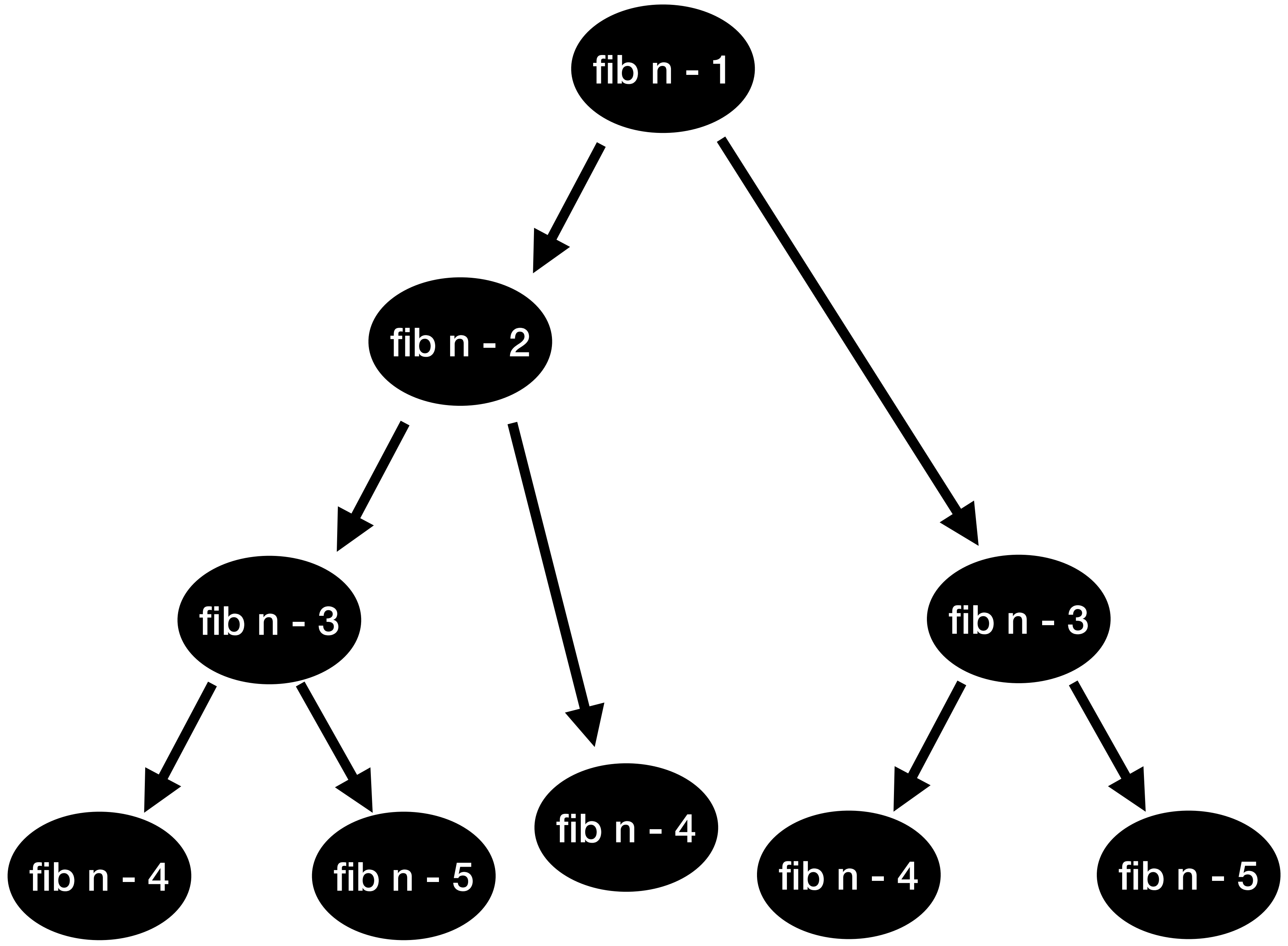
Part 2: The **Cost** of Abstraction

Let’s Profile Fibonacci

```
>> benchFib

  ^ self < 2
     ifTrue: [1]
     ifFalse: [
      (self-1) benchFib + (self-2) benchFib + 1]
```

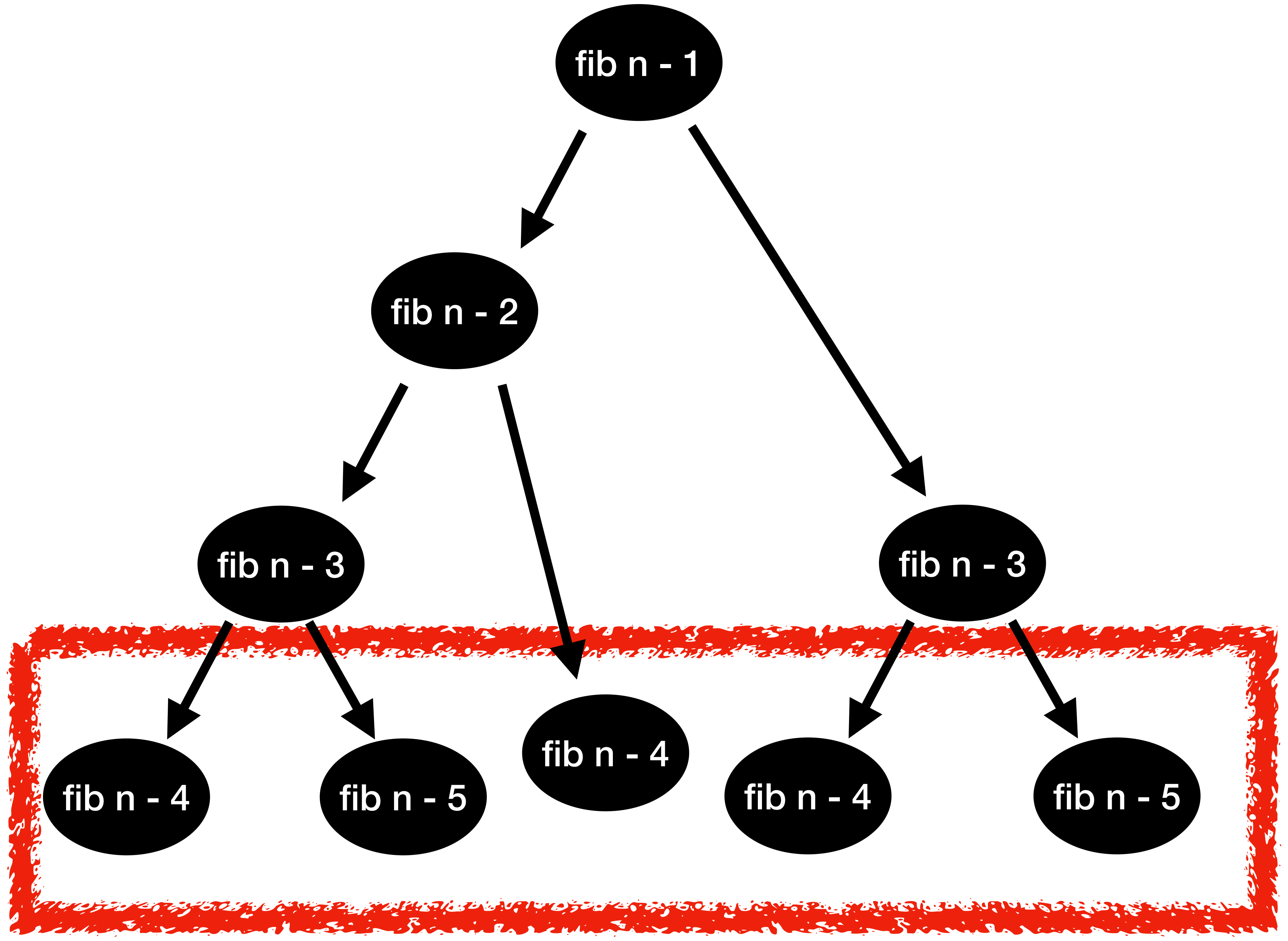
Let's *Benchmark* with Fibonacci

- **Best case** for proxy infrastructure
  - no exceptions
  - no non-local return
  - no meta-recursion
  - no concurrent usages by default



Let's *Benchmark* with Fibonacci (II)

- Good case to measure profiler/proxy overhead
  - Simulate a **big call-tree**
  - Leaves are fast paths (early exits)
    - => **high overhead *expected***
  - **`fib(n)` ~~ number of messages**

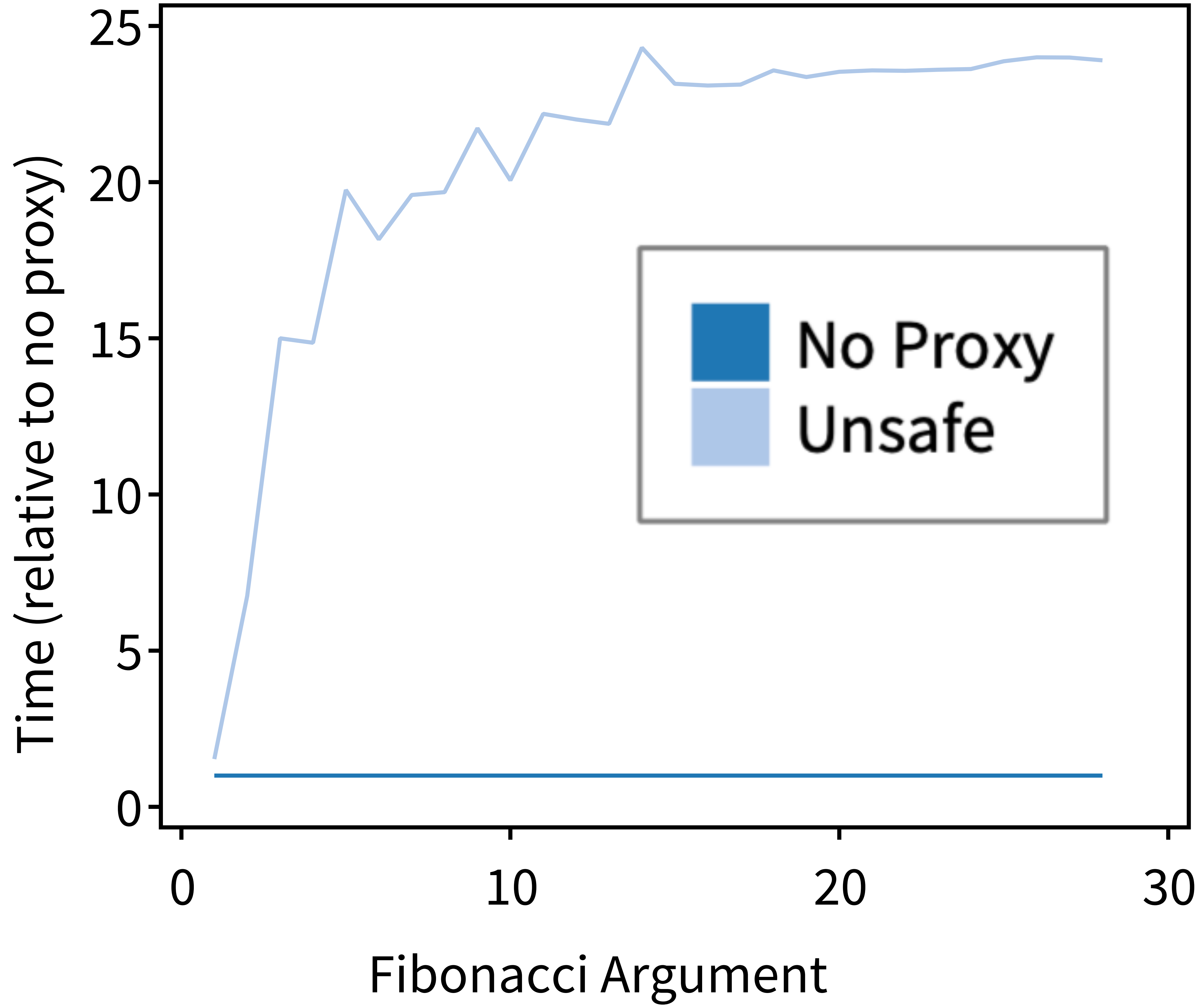


Our Lower Bound is `run:with:in:`

```
run: aSelector with: anArrayOfObjects in: aReceiver

    ^ self
        forwardMethod: originalMethod
        withReceiver: aReceiver
        withArguments: anArrayOfObjects
```

`run:with:in:` Performance vs fib(x)

- **~25x** slower !
- *Seems* faster for lower args
- ***Noise*** due to μs measures?

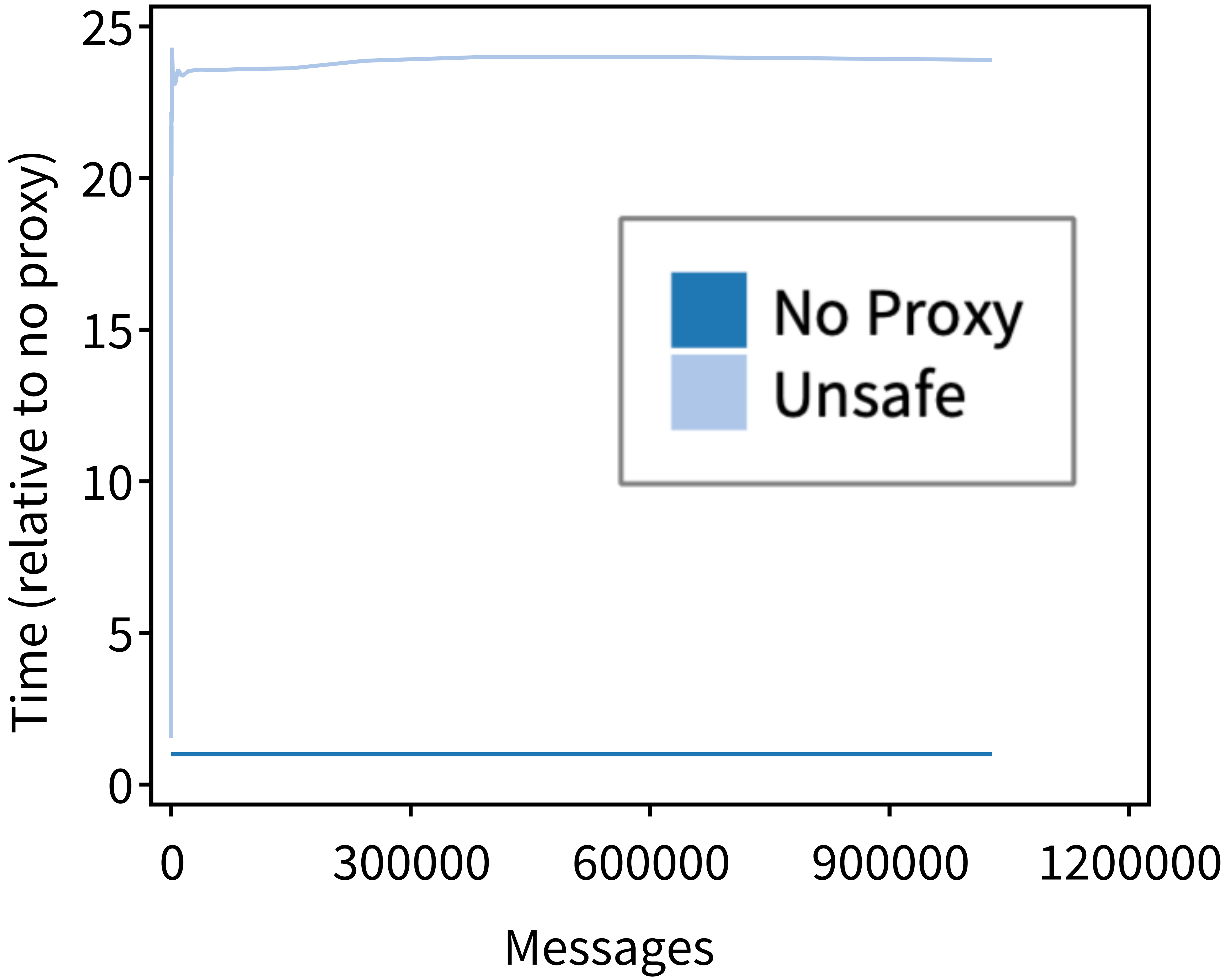


* Averages of 100 runs in μs. X = 1 to: 28

`run:with:in:` Performance vs *Messages*

- *Consistent* **~25x** overhead

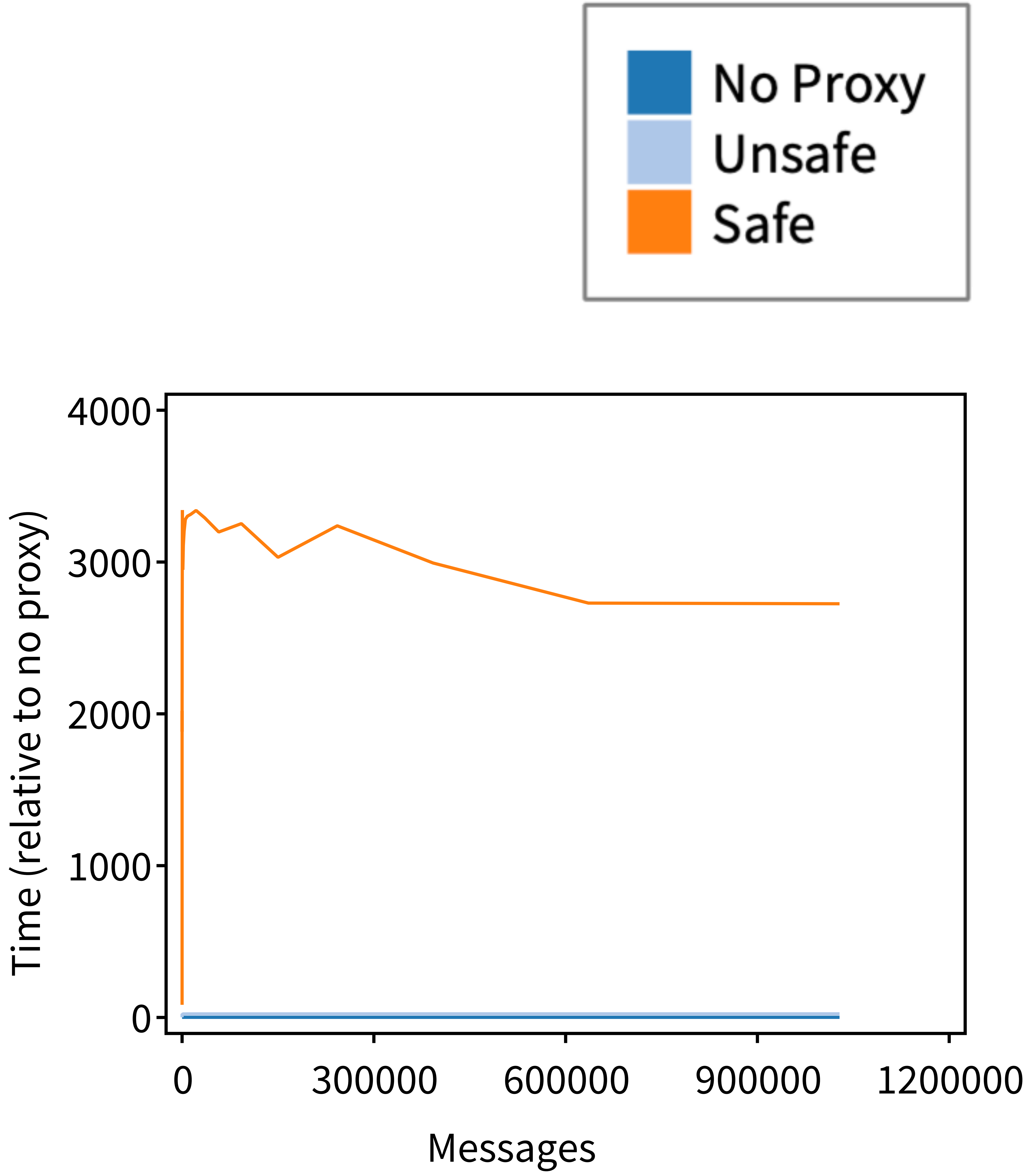
- Cries for *language implementation improvement* (!!)



* Averages of 100 runs in μs.

The Cost of Safety

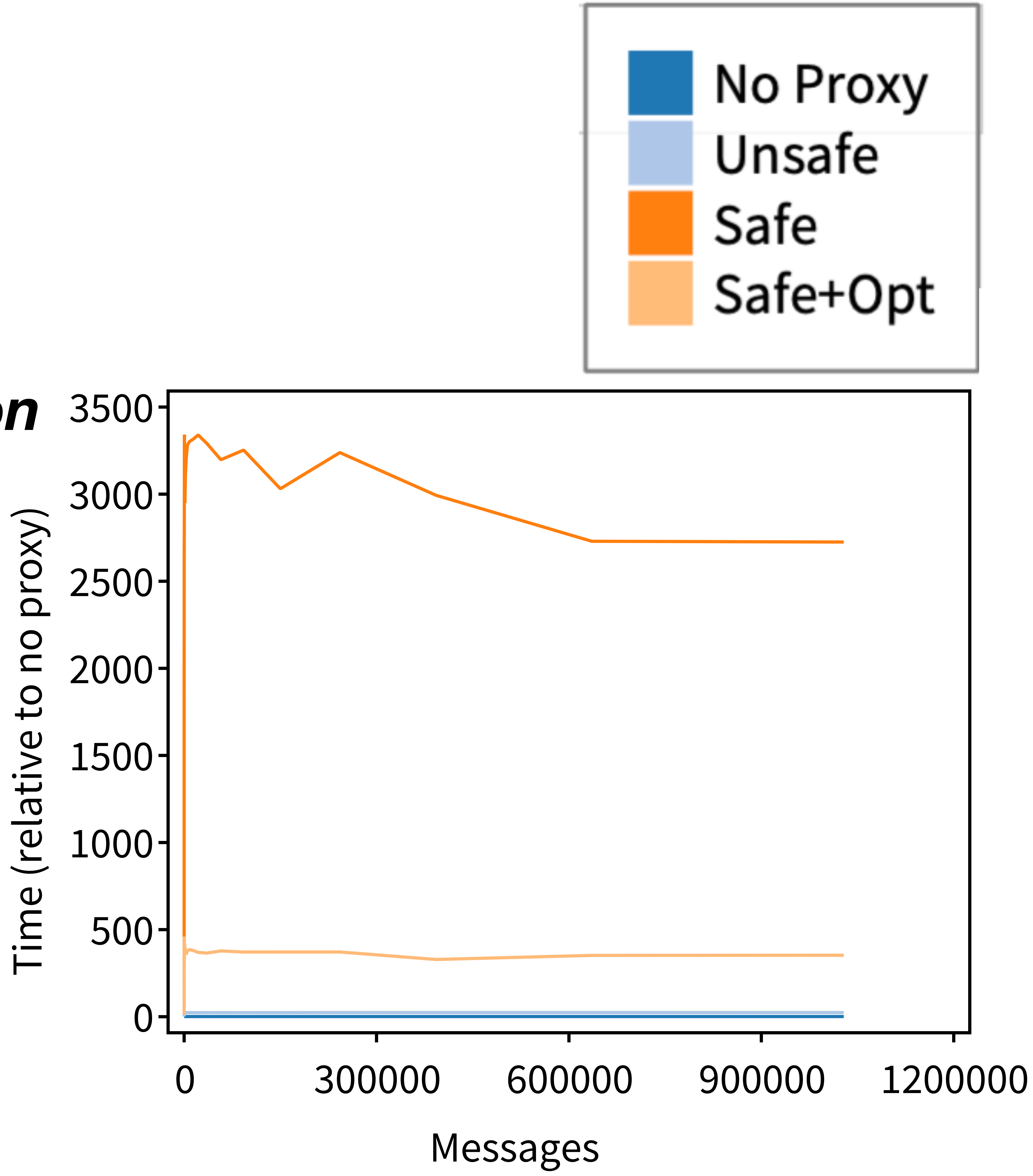
- Safe method proxies are **~3000x worse**
    - Non-clean closures
        - allocation
        - *thisContext* reification
    - More messages (!)
        - `#ensure:`
        - *meta-recursion control*
        - `#before`, `#after` hooks



* Averages of 100 runs in μs.

Can we get better?

- Down to **~400x just *removing abstraction***
  - Inlinings (!!)
    - to remove messages
    - to avoid blocks
    - differentiate fast vs slow path
      - concurrent, meta-recursive



* Averages of 100 runs in μs.

Overhead of Call-Tree Construction

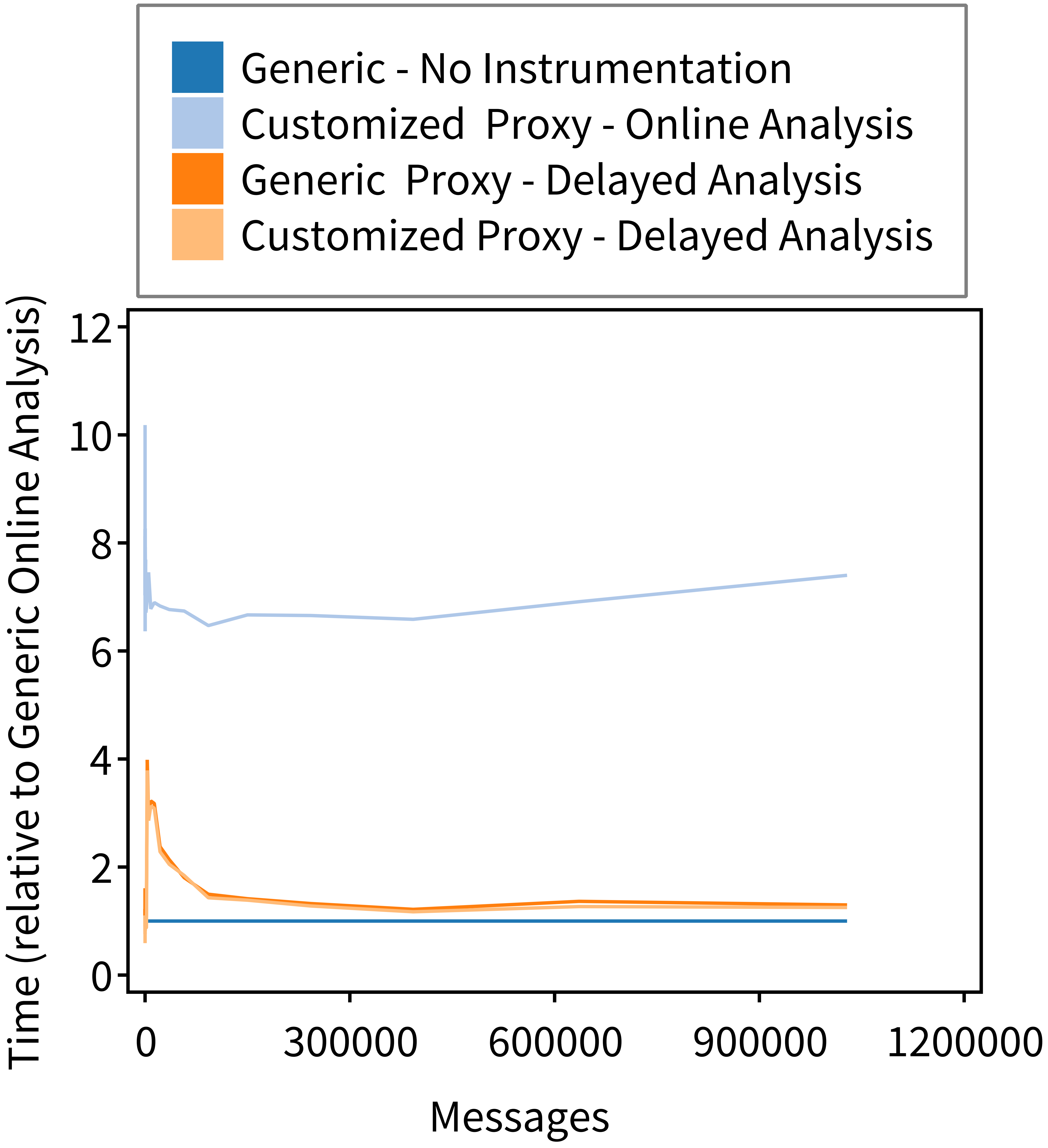
- **2 proxy variants**
  - Generic: handler object
  - Customized: *inlined* handler
- **2 instrumentation variants**
  - Online: build the call tree while executing
  - Delayed: *trace* the minimum to build it in a post-process
- **Comparison Baseline:** safe+*opt* proxy with no instrumentation

* Averages of 100 runs in μs.

Call-tree construction

- Generic + Online was off the charts :)
  - => off the presentation too

- **Delaying** the analysis is the best
- Customization gets only *slightly better*
  - removes 4 messages per call

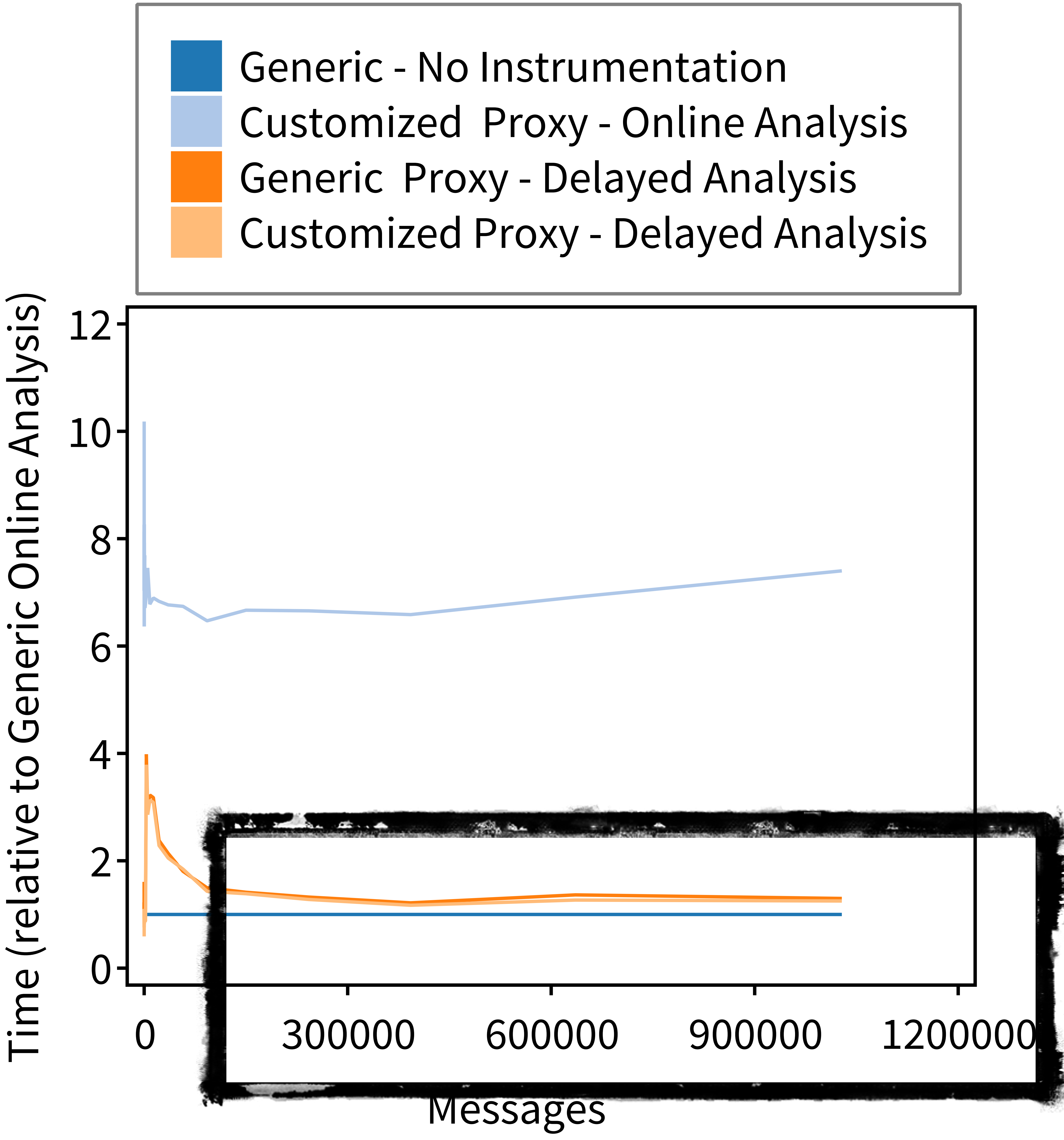


* Averages of 100 runs in μs.

Call-tree construction

- Generic + Online was off the charts :)
    - => off the presentation too

- **Delaying** the analysis is the best
- Customization gets only *slightly better*
    - removes 4 messages per call



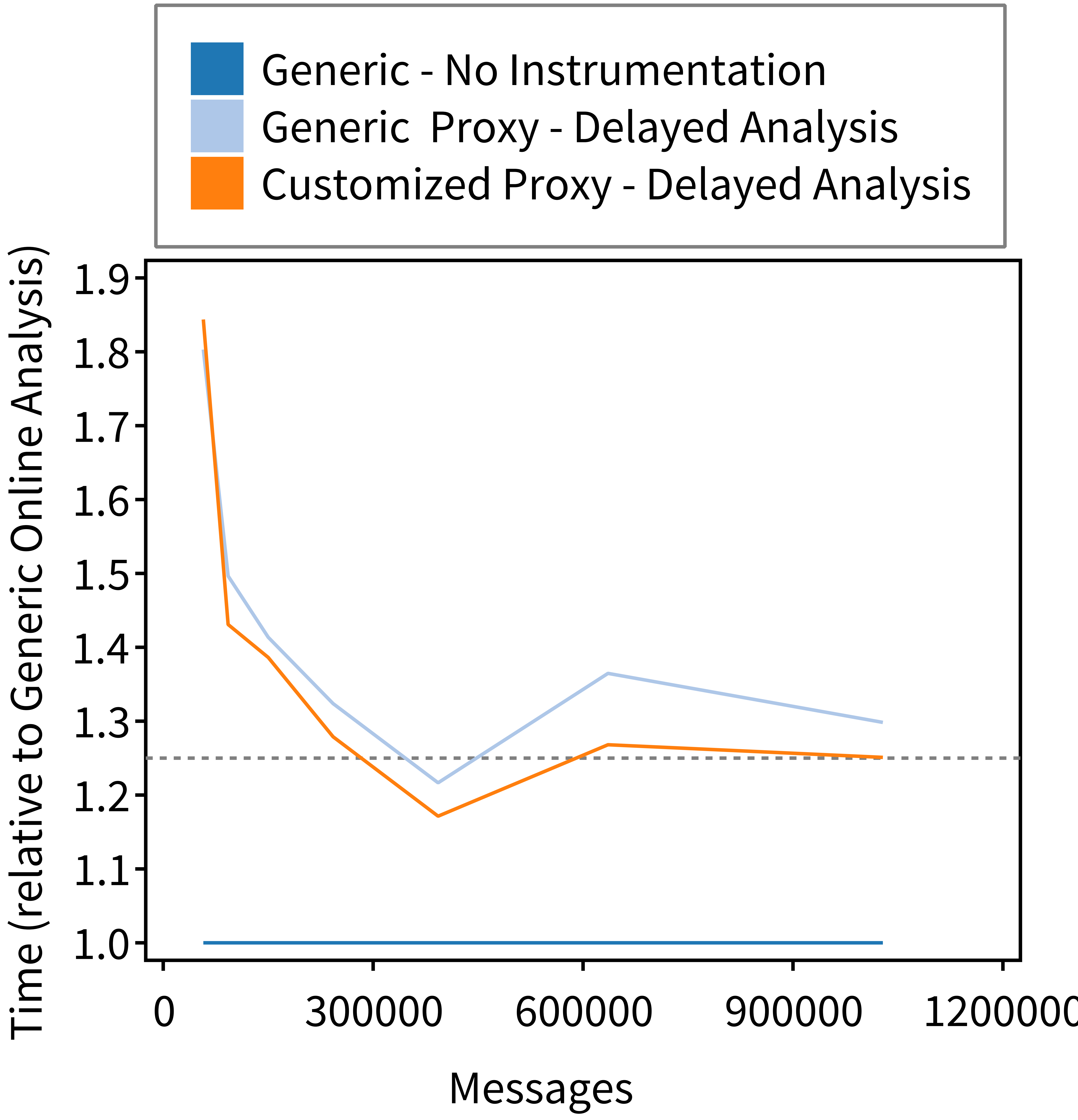
* Averages of 100 runs in μs.

Zooming in

- Delayed is **~1.25x** proxy alone

~1.25 * 400x (safety) ~= *~500x overhead*

(over no instrumentation)

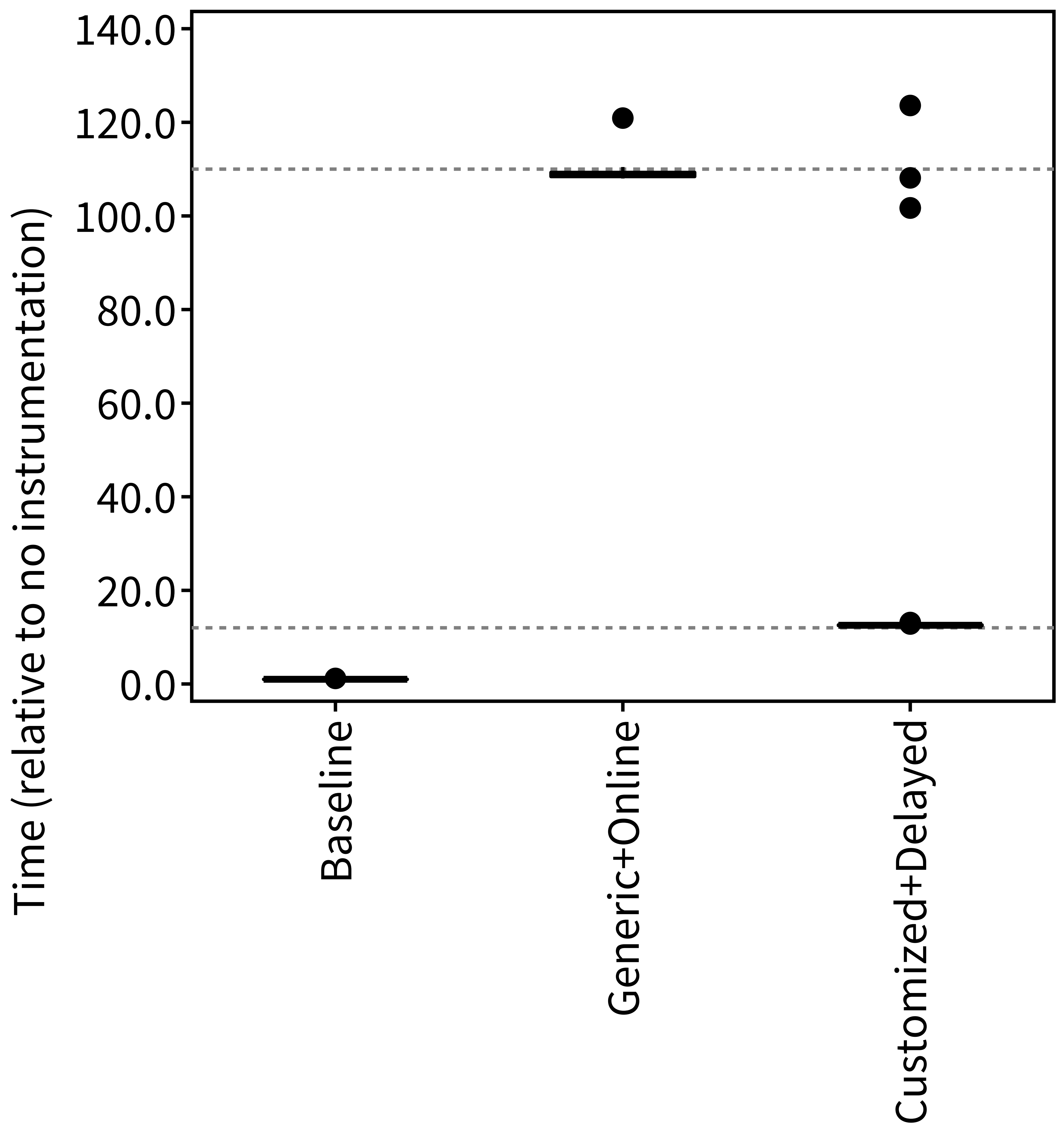


* Averages of 100 runs in μs.

Profiling the Compiler — *again*

- Partial Instrumentation
- Down from ~110x to ~12x

```
prf := PrfMethodProfiler new.
prf addPackage: OpalCompiler package.
prf addPackage: RBParser package.
prf profile: [ Integer recompile ].
```



Takeaways

https://github.com/pharo-contributions/MethodProxies
https://github.com/pharo-contributions/MethodProfiler

- Users need *native language support for instrumentation*
    - **Safe**, **stratified** and ****efficient****
- Low-level hooks are **not enough**: they miss abstractions
    - Think twice when writing your own proxy implementation!
        - Think concurrency, think stack unwind, thing meta-recursions

* Supported by AlaMVic Action Exploratoire INRIA