Retrofitting Concurrency

Lessons from the engine room

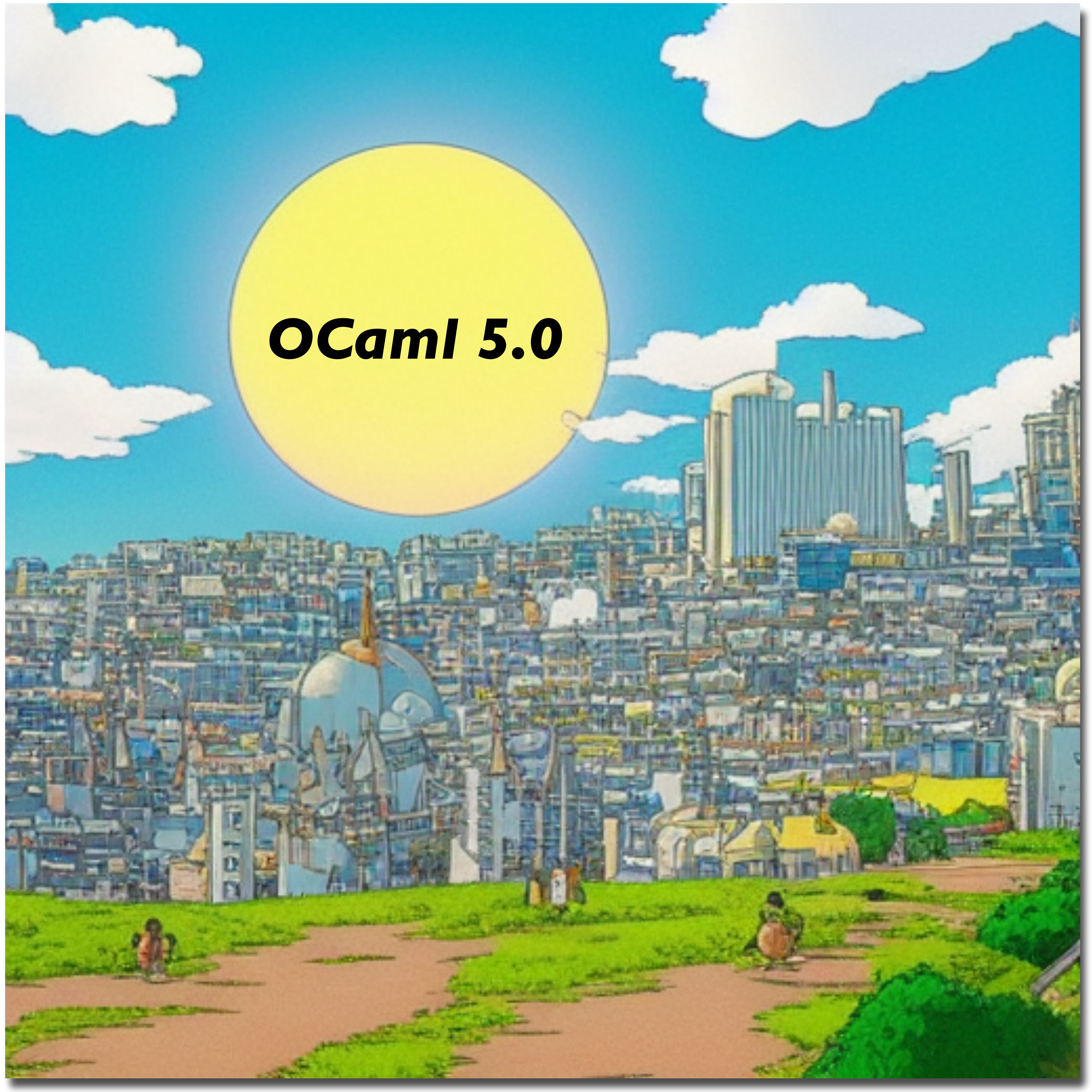
"KC" Sivaramakrishnan

IIT MADRAS

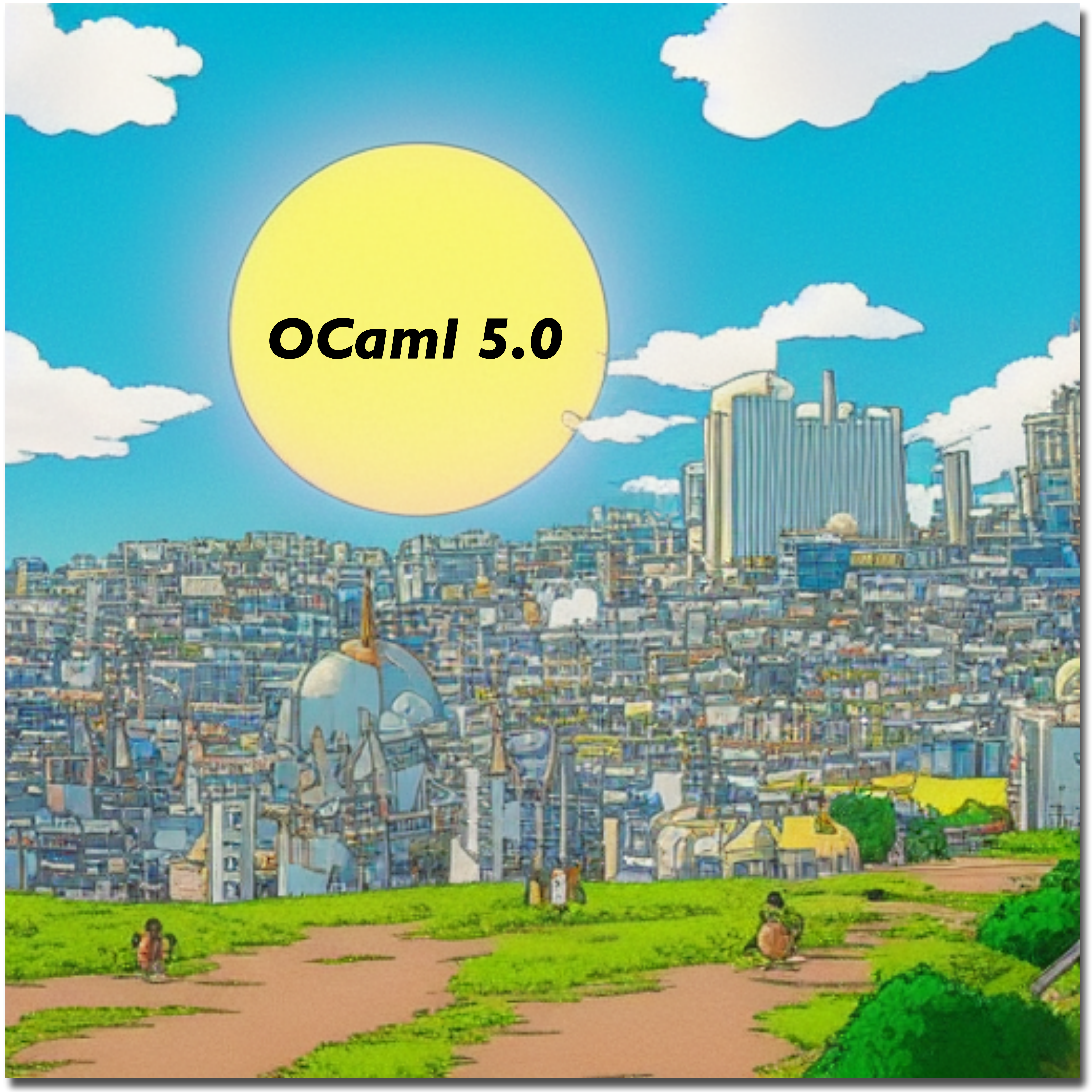
Tarides

Images made with Stable Diffusion

In Sep 2022…



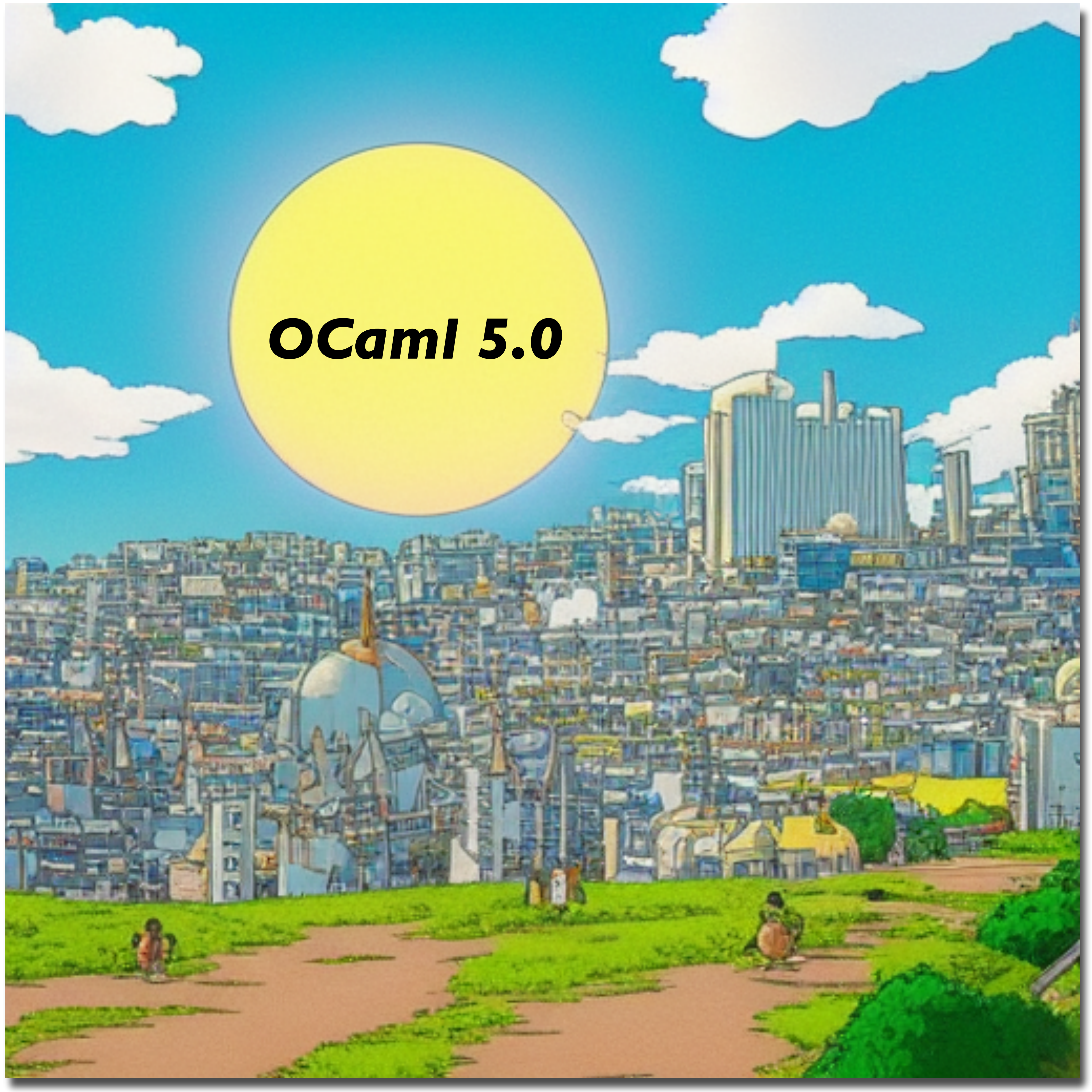
In Sep 2022…



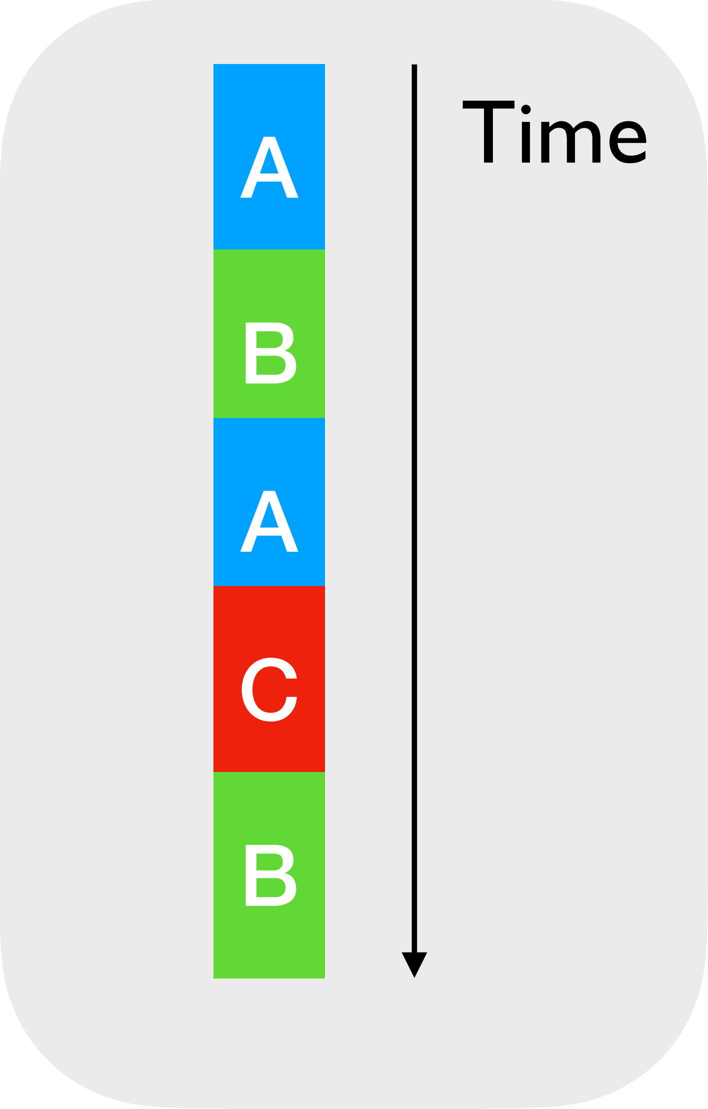
Concurrency

Parallelism

In Sep 2022…



Concurrency

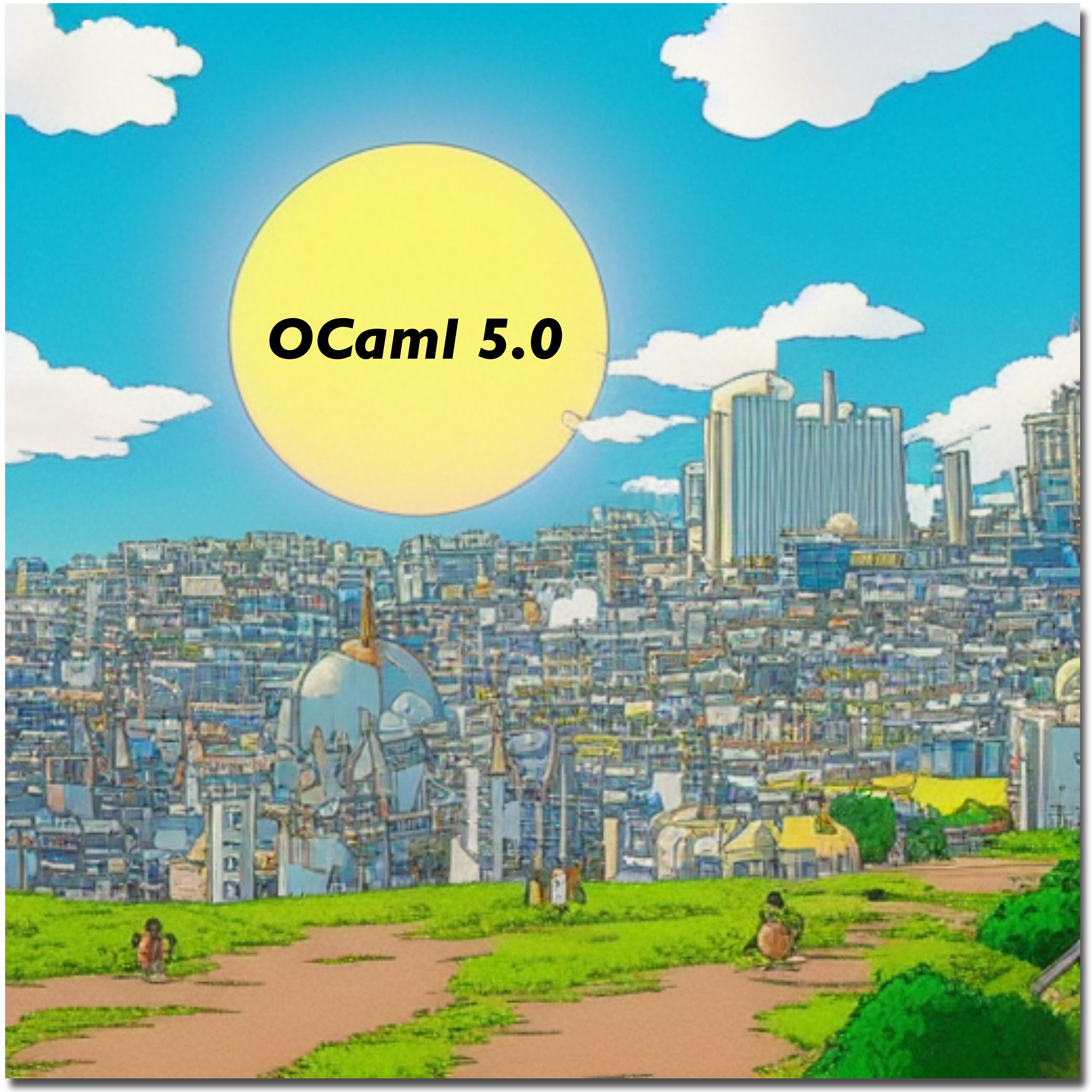


Overlapped execution

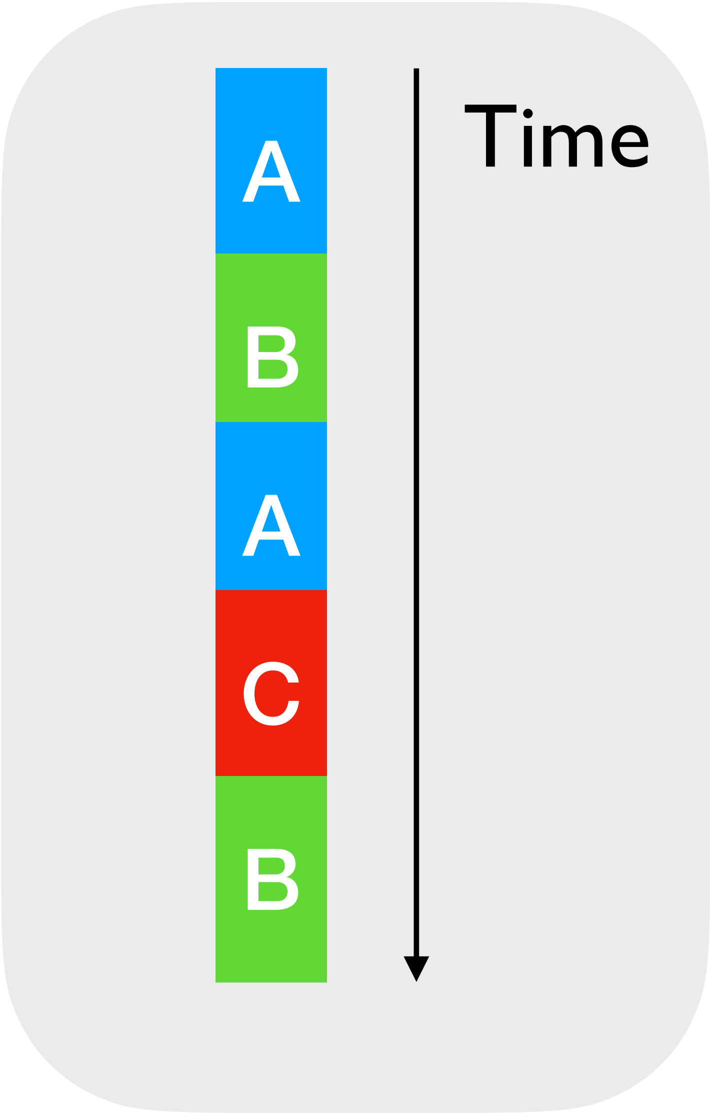
Effect Handlers

Parallelism

In Sep 2022…



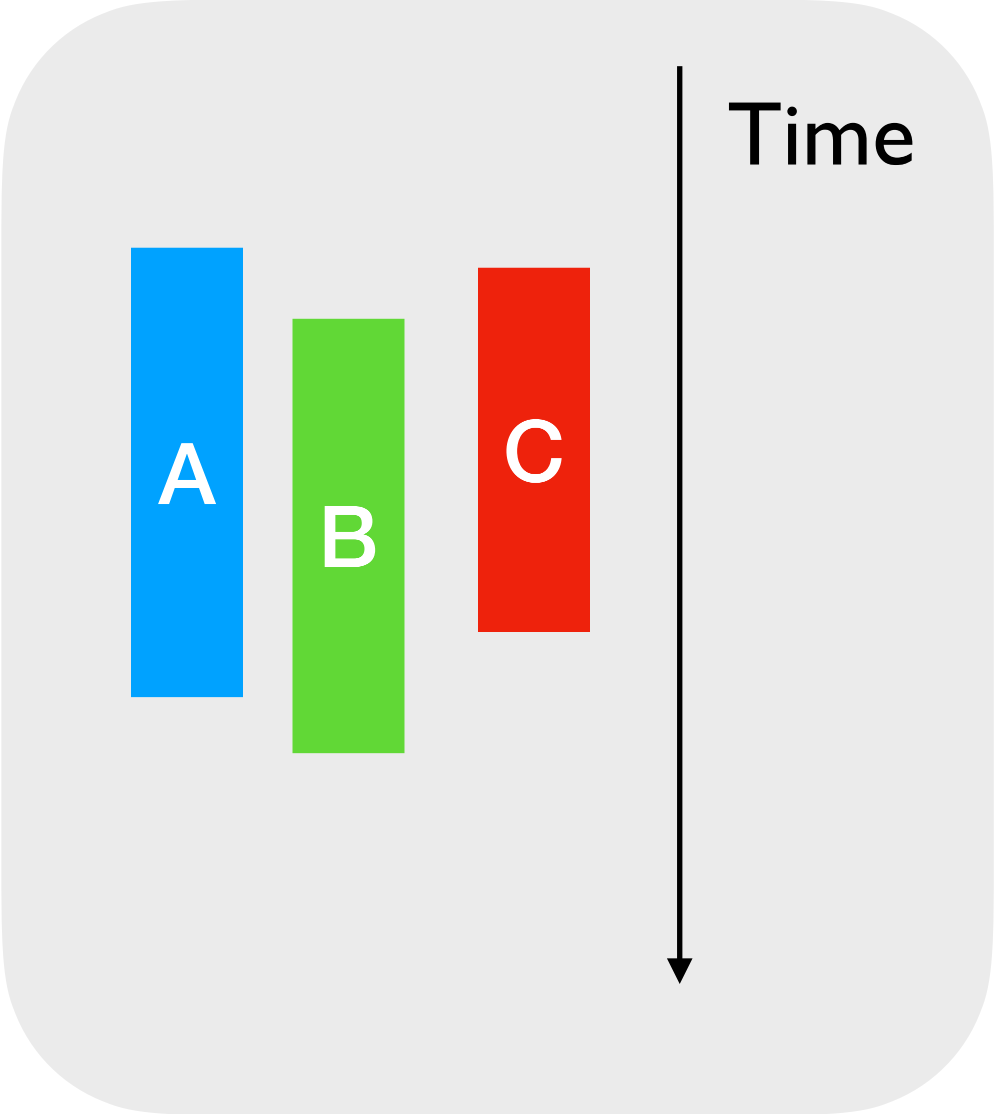
Concurrency



Overlapped execution

Effect Handlers

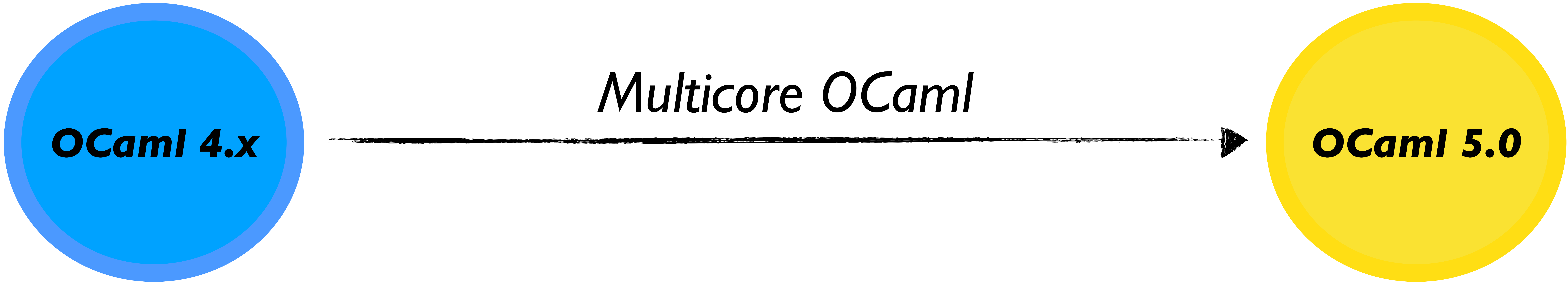
Parallelism



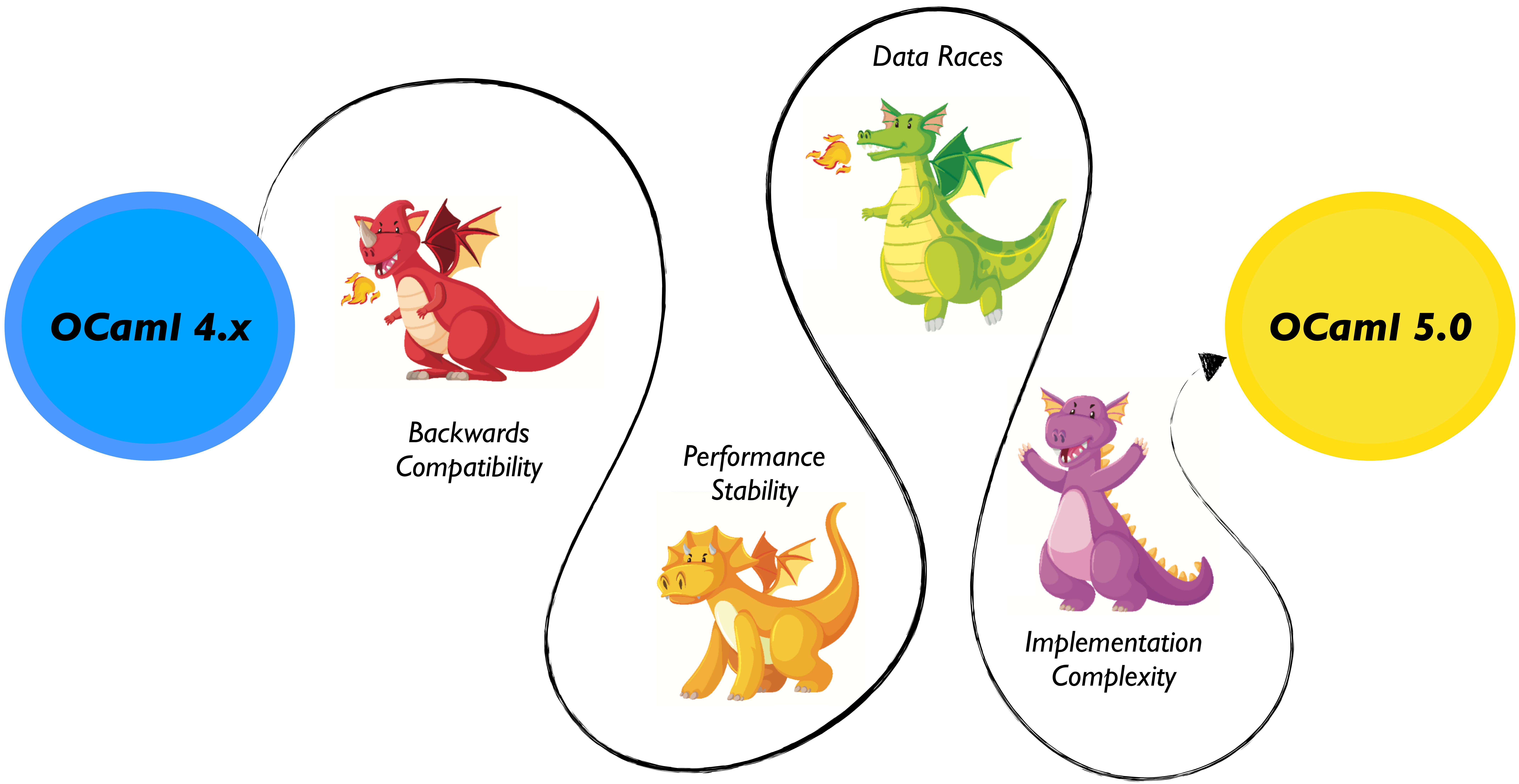
Simultaneous execution

Domains

In this talk…



In this talk...


OCaml 4.x
Backwards Compatibility
Performance Stability
Data Races
Implementation Complexity
OCaml 5.0

Journey

Takeaways

In the year 2014…

18 year-old, industrial-strength, pragmatic, functional programming language

In the year 2014…

OCaml

18 year-old, industrial-strength, pragmatic, functional programming language

Industry

FACEBOOK · Tarides · Bloomberg · docker · ahrefs · SimCorp · Microsoft · Tezos · Jane Street

Projects

hack · flow · COMPCERT

In the year 2014…

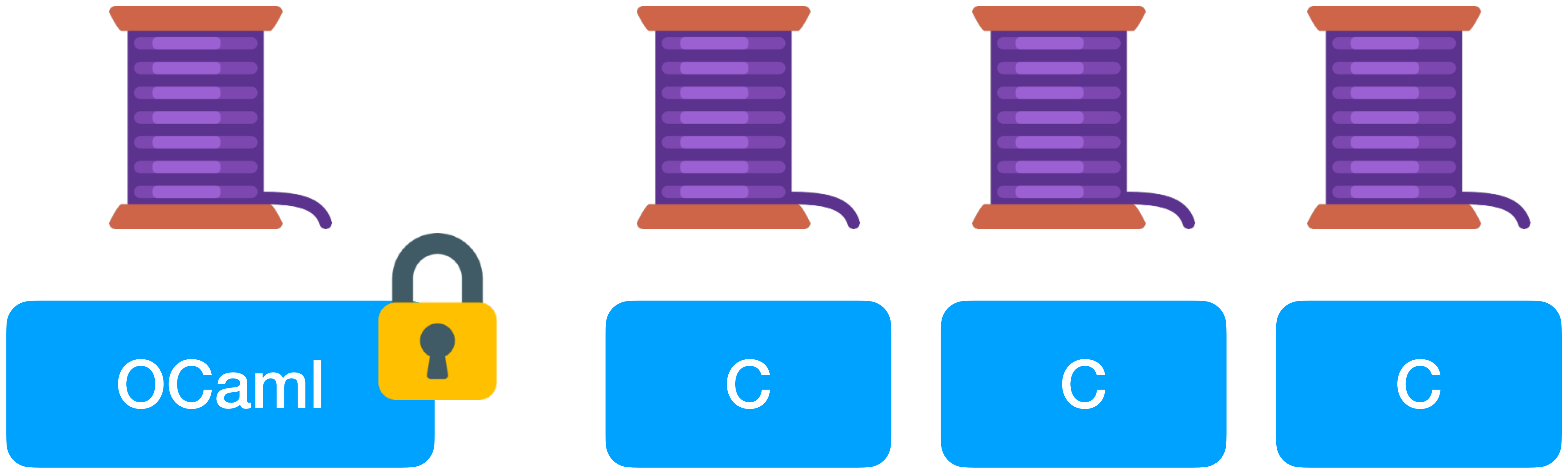
18 year-old, industrial-strength, pragmatic, functional programming language

Industry

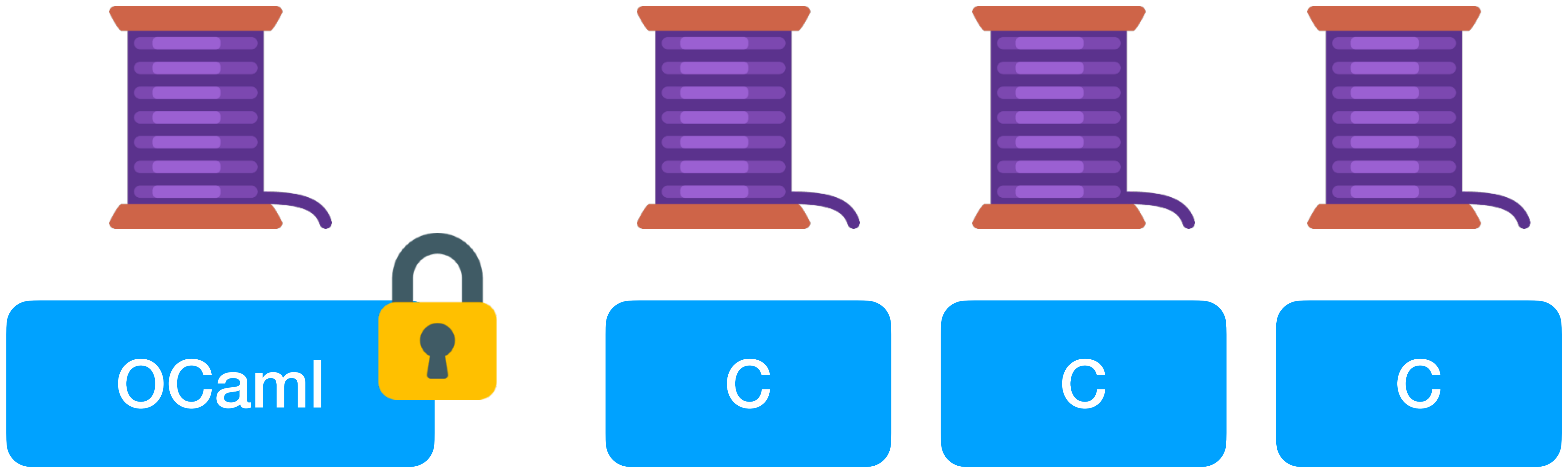
Projects

No multicore support!

Runtime lock

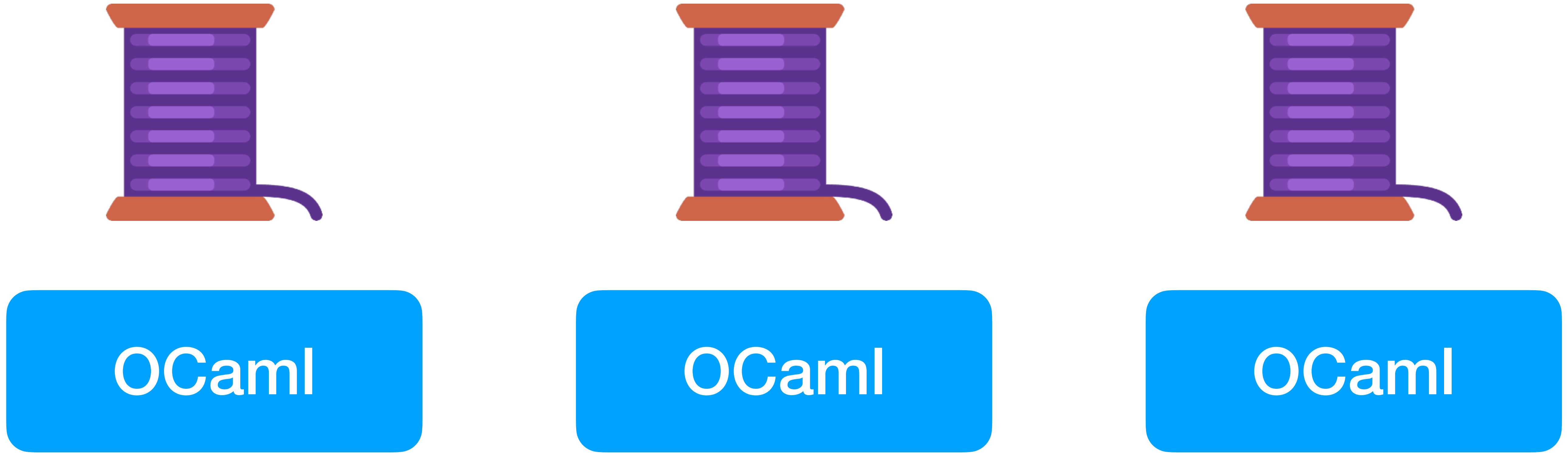


Runtime lock

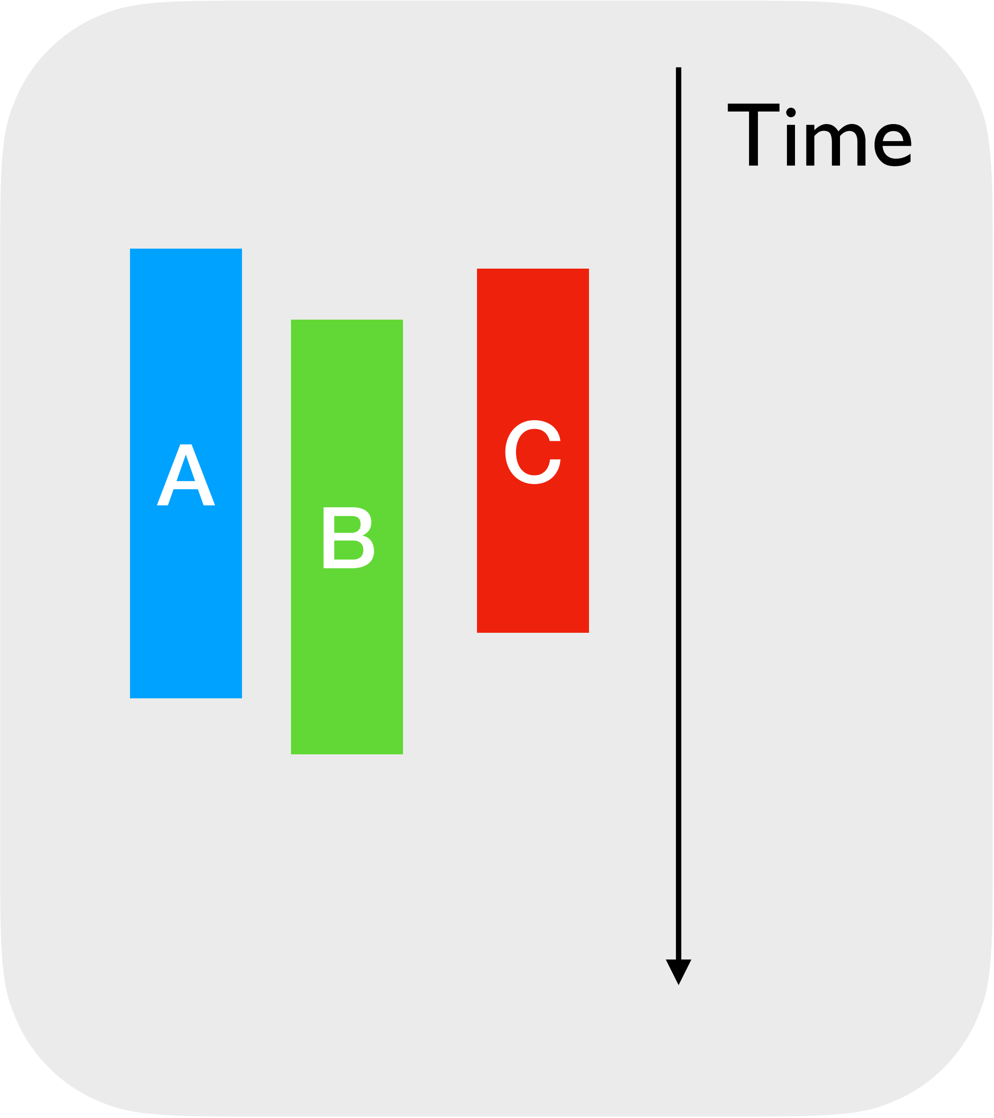


python™ GIL

Eliminate the runtime lock



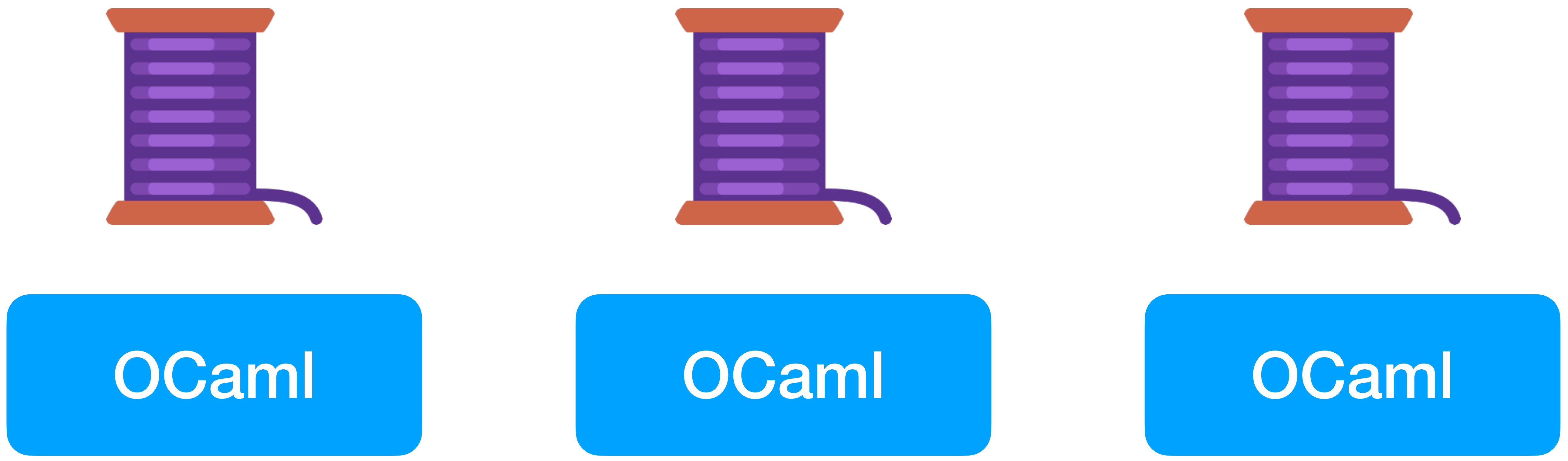
Parallelism



Simultaneous execution

Domains

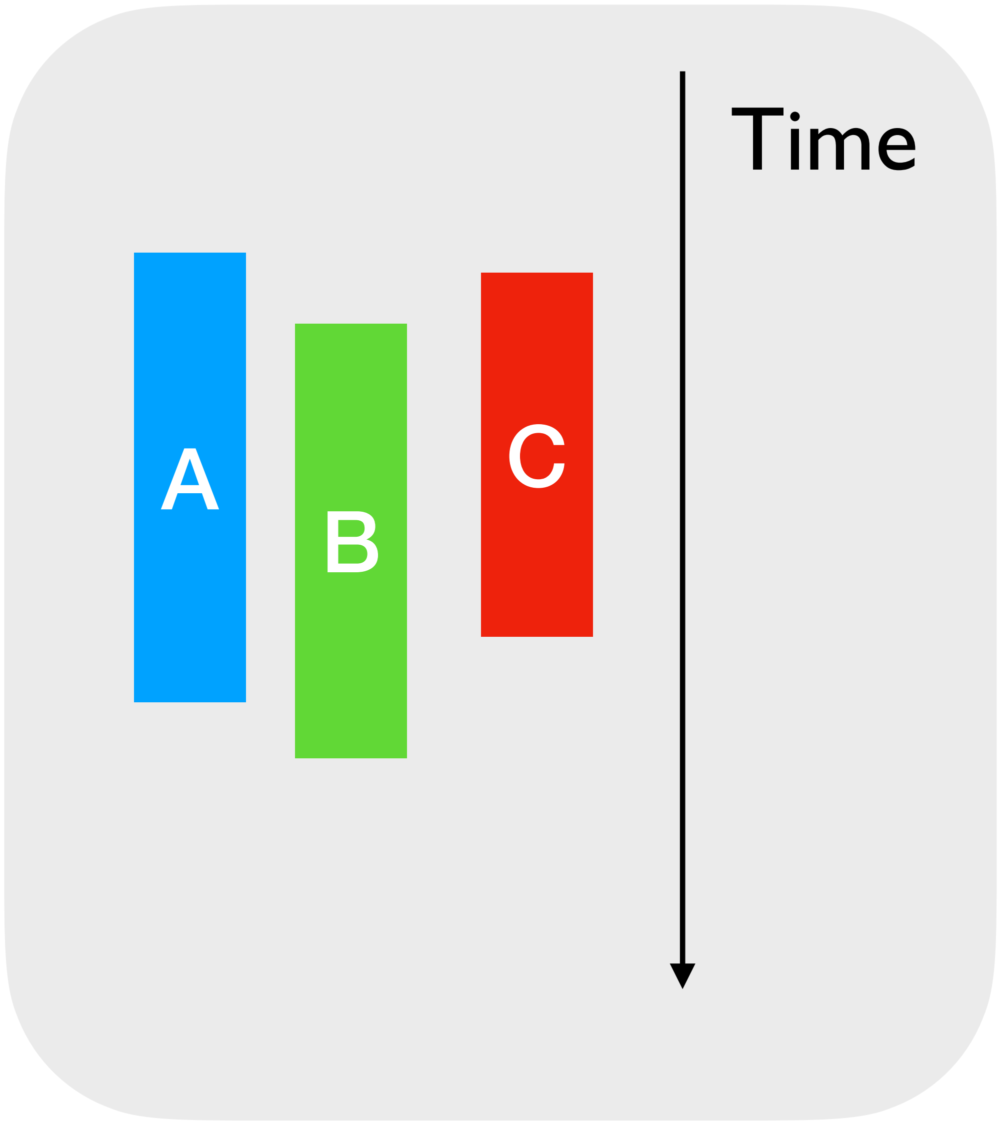
Eliminate the runtime lock



python™ GIL

Sam Gross, Meta, "Multithreaded Python without the GIL"

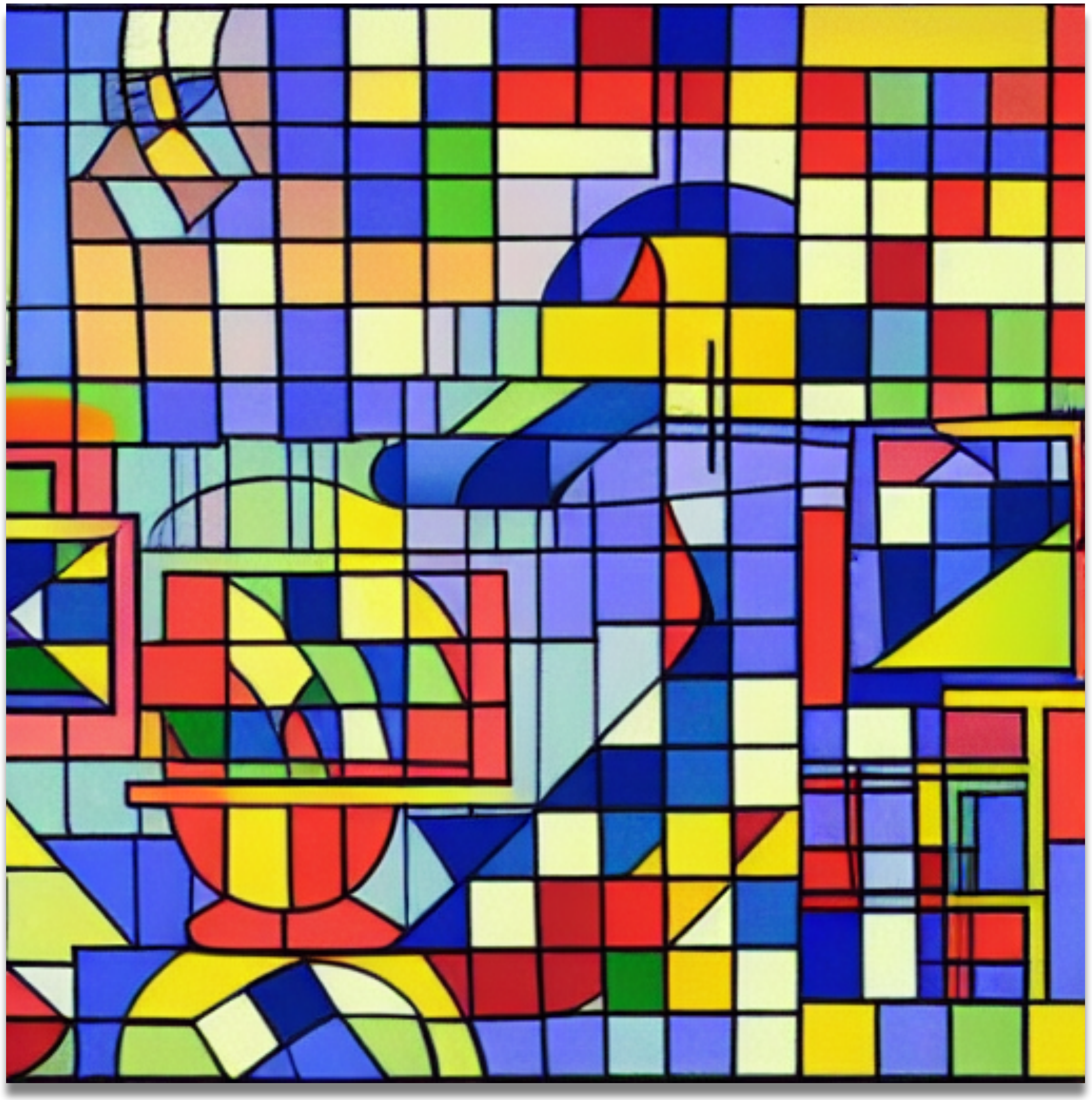
Parallelism



Simultaneous execution

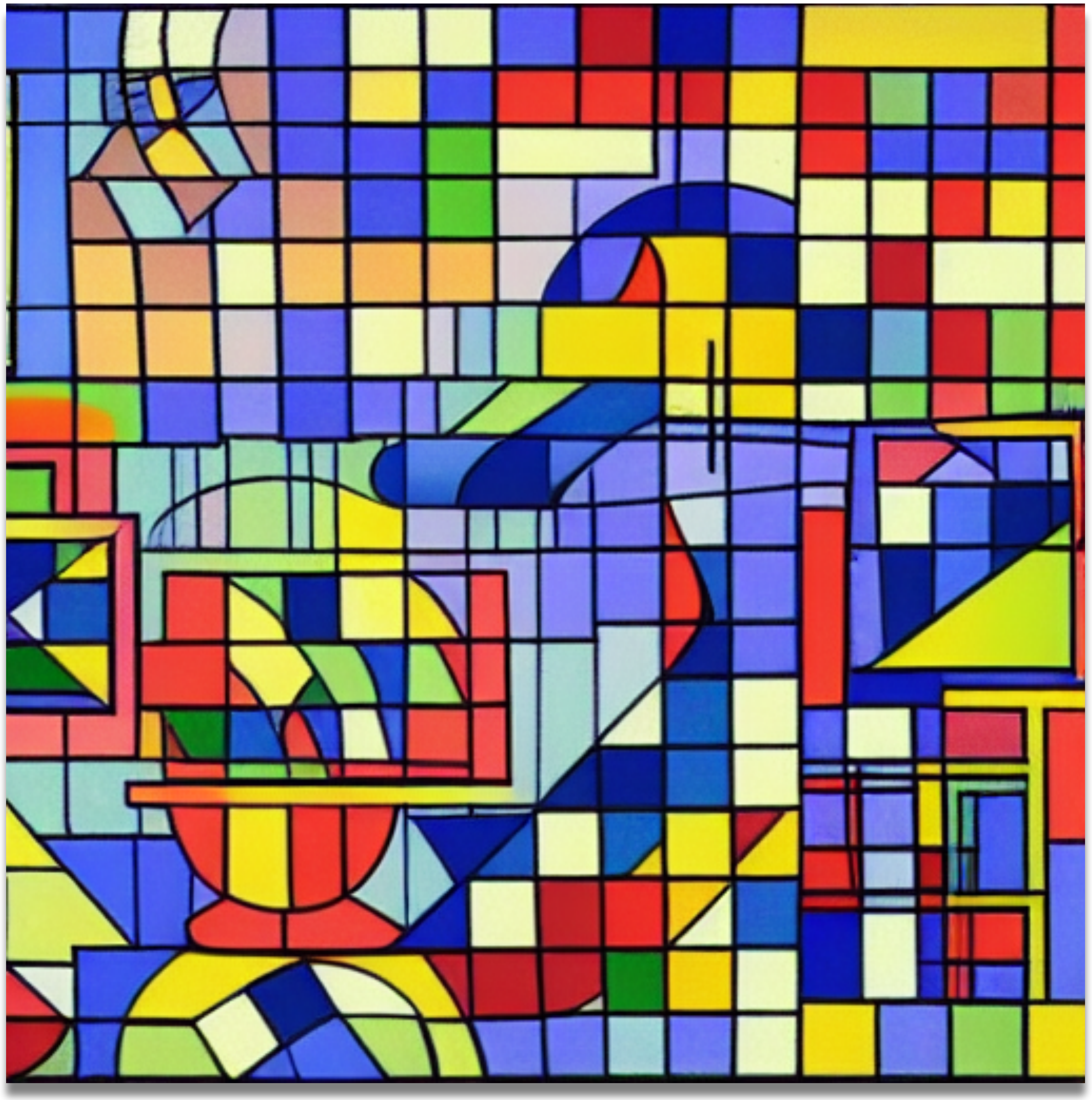
Domains

Retrofitting Challenges ~> Approach



- **Millions** of lines of legacy software
  - ✦ Most code likely to remain sequential even with multicore
- Cost of refactoring is prohibitive

Retrofitting Challenges ~> Approach



- **Millions** of lines of legacy software
  - ✦ Most code likely to remain sequential even with multicore
- Cost of refactoring is prohibitive

Do not break existing code!

Retrofitting Challenges ~> Approach

- *Low latency* and *predictable* performance
  - ✦ Great for ~10ms tolerance

Retrofitting Challenges ~> Approach

- *Low latency* and *predictable* performance
  - ✦ Great for ~10ms tolerance

Optimise for GC latency before scalability

Retrofitting Challenges ~> Approach

- OCaml core team is composed of *volunteers*
  - ✦ Aim to reduce *complexity* and *maintenance* burden

Retrofitting Challenges ~> Approach

- OCaml core team is composed of *volunteers*
  - ✦ Aim to reduce *complexity* and *maintenance* burden

No separate sequential and parallel runtimes

Unlike –threaded runtime

Retrofitting Challenges ~> Approach

- OCaml core team is composed of *volunteers*
  - ✦ Aim to reduce *complexity* and *maintenance* burden

No separate sequential and parallel runtimes

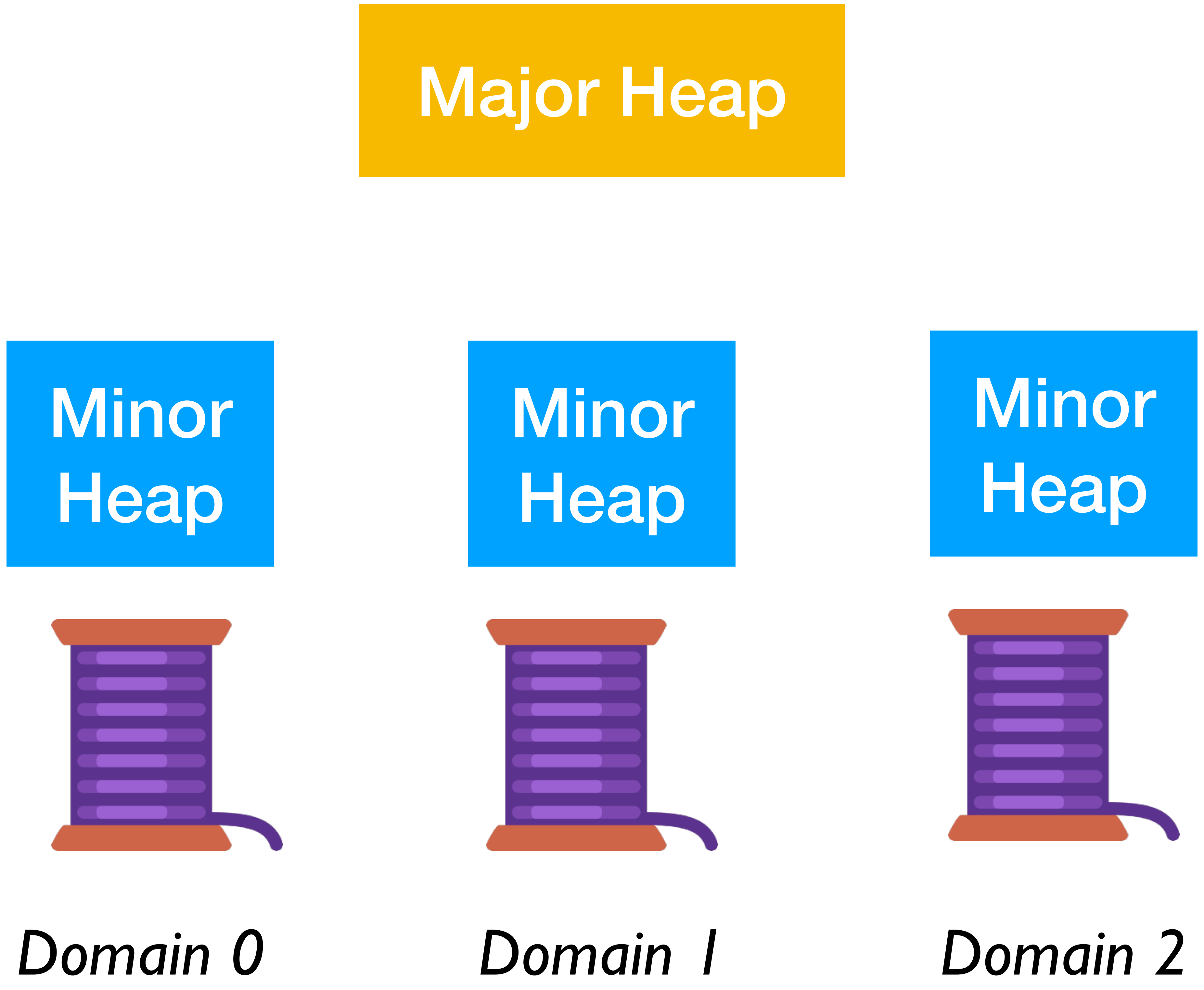
⇒

Existing sequential programs run just as fast using just as much memory

Unlike -threaded runtime

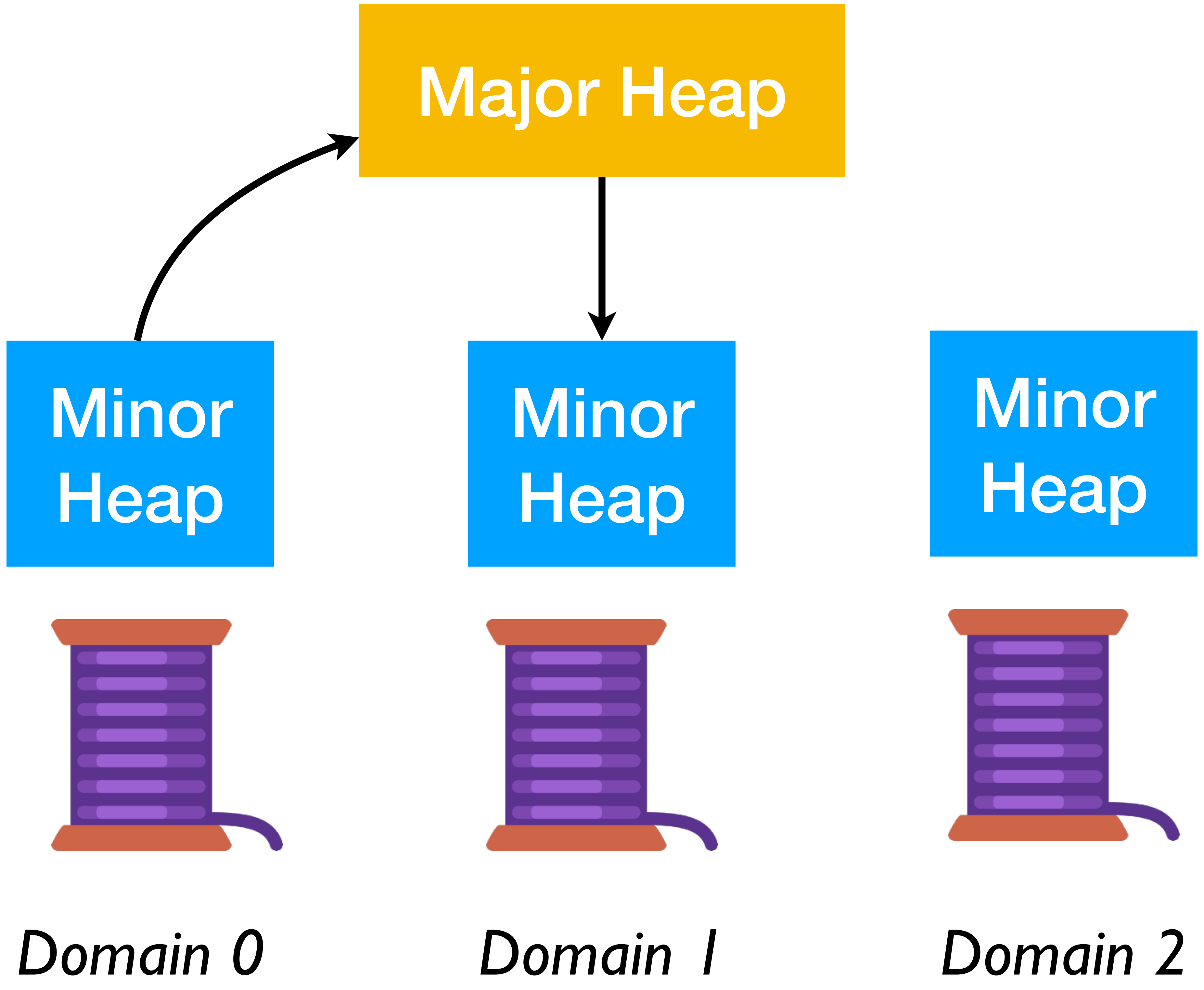
Parallel Allocator & GC

Medieval garbage truck according to Stable Diffusion



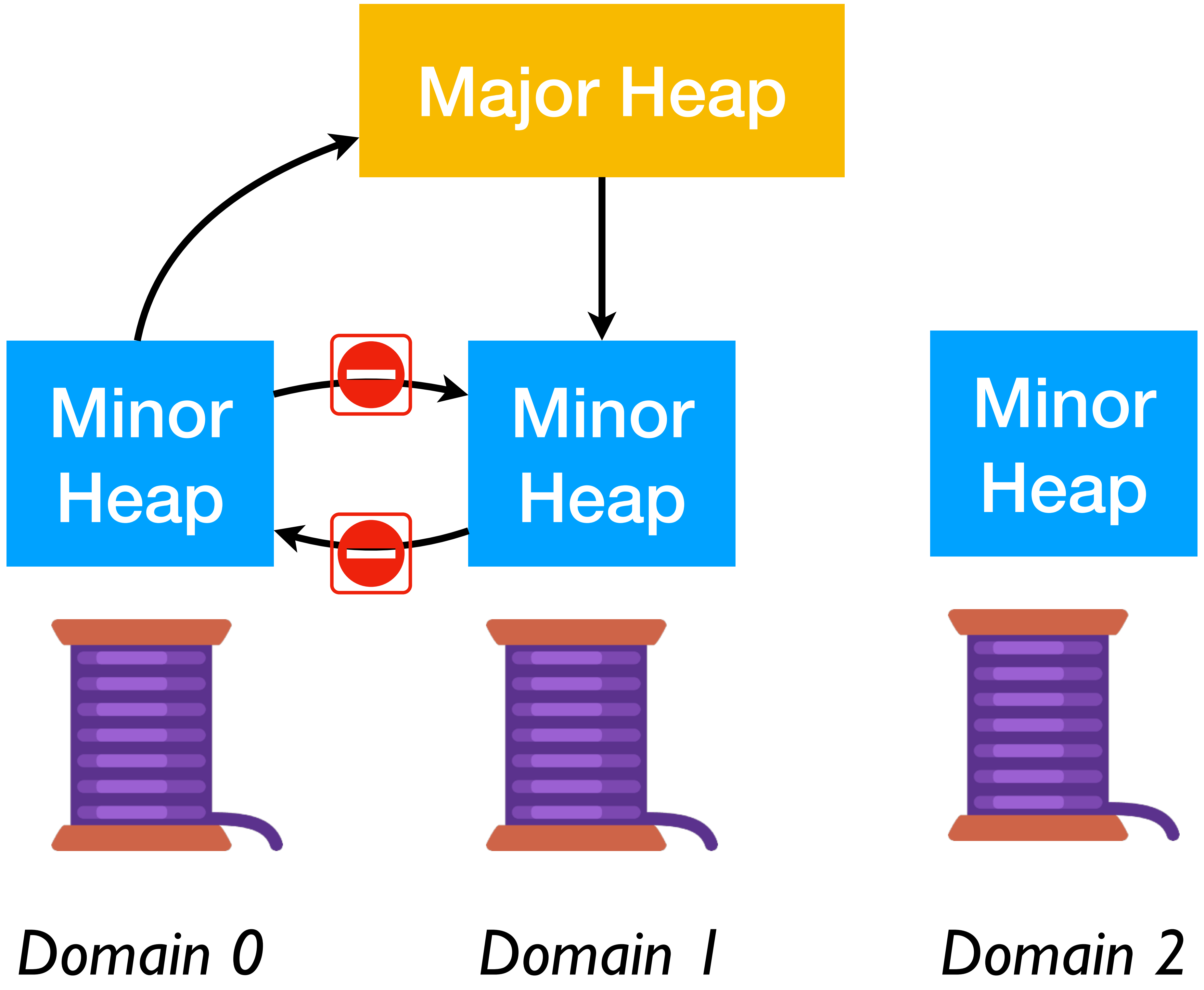
Parallel Allocator & GC

Medieval garbage truck according to Stable Diffusion

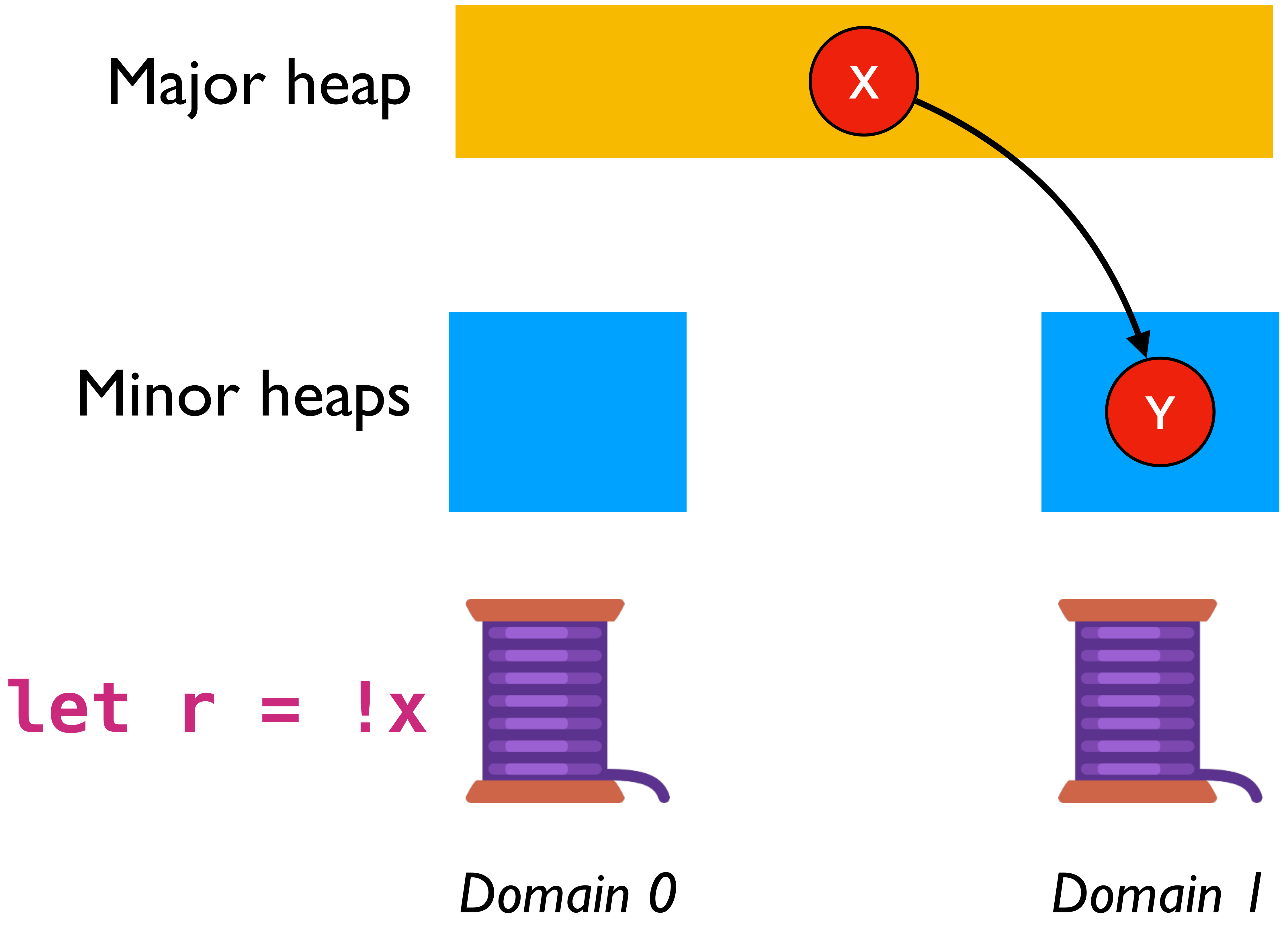


Parallel Allocator & GC

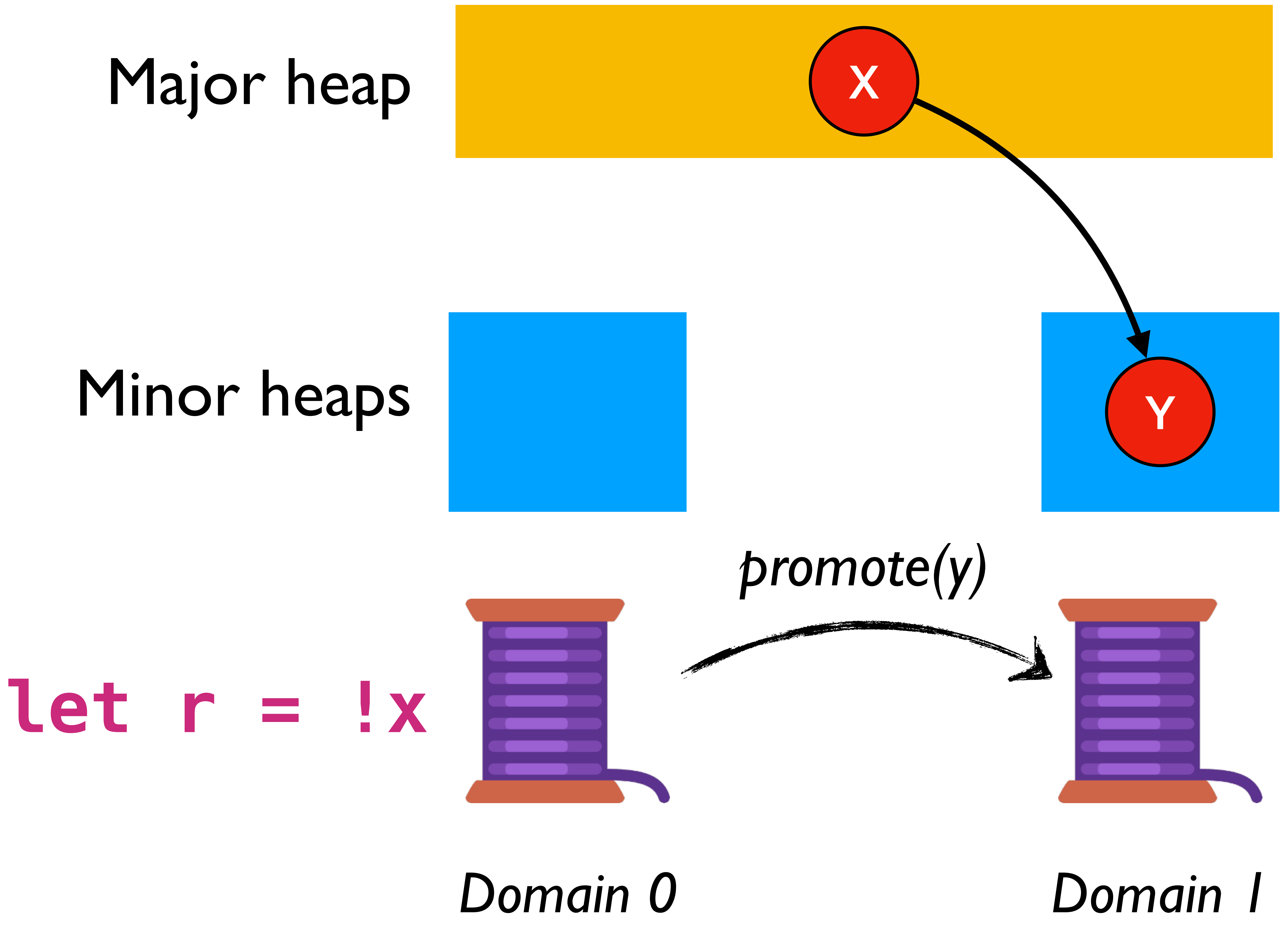
Medieval garbage truck according to Stable Diffusion



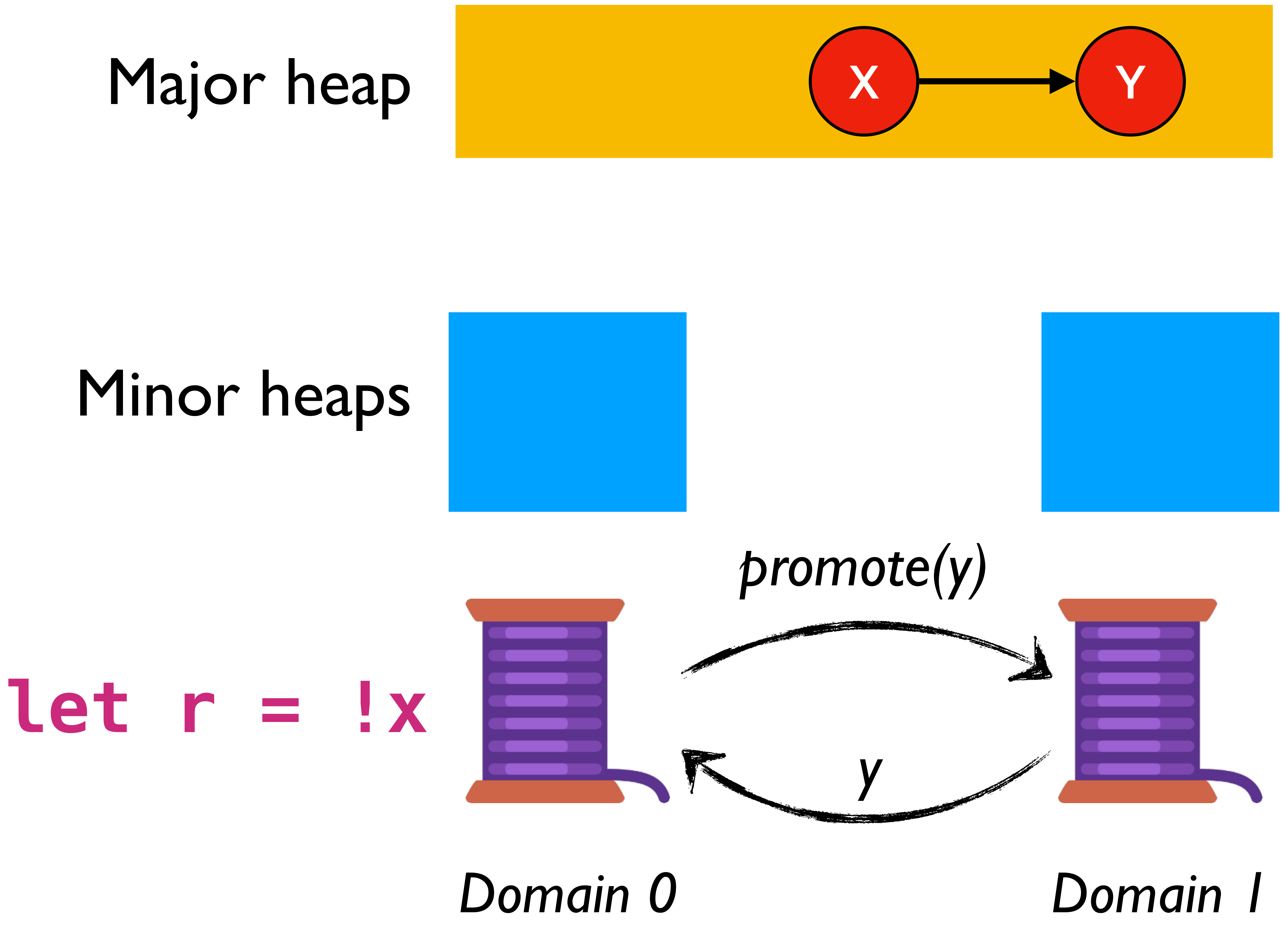
Access remote objects



Access remote objects

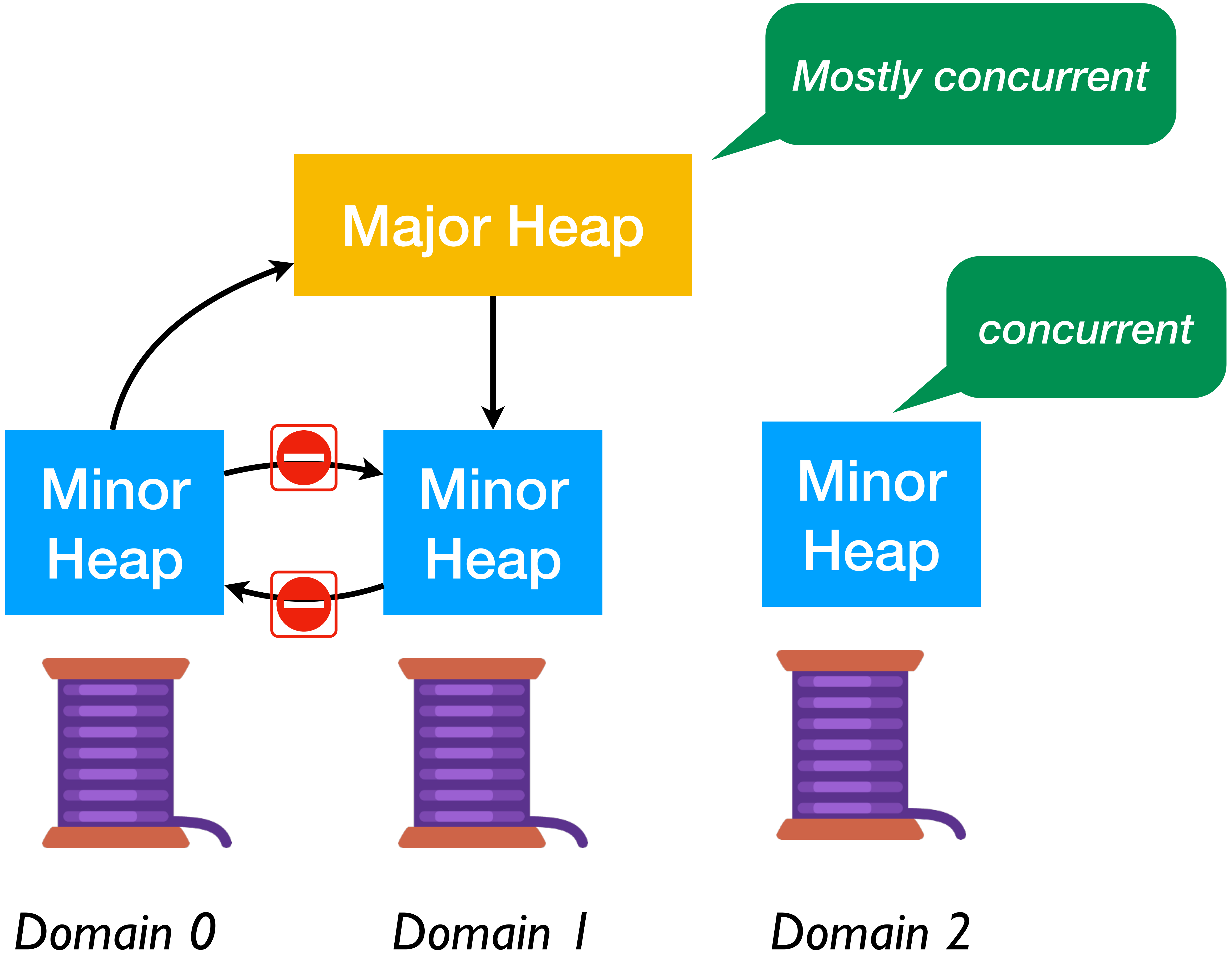


Access remote objects

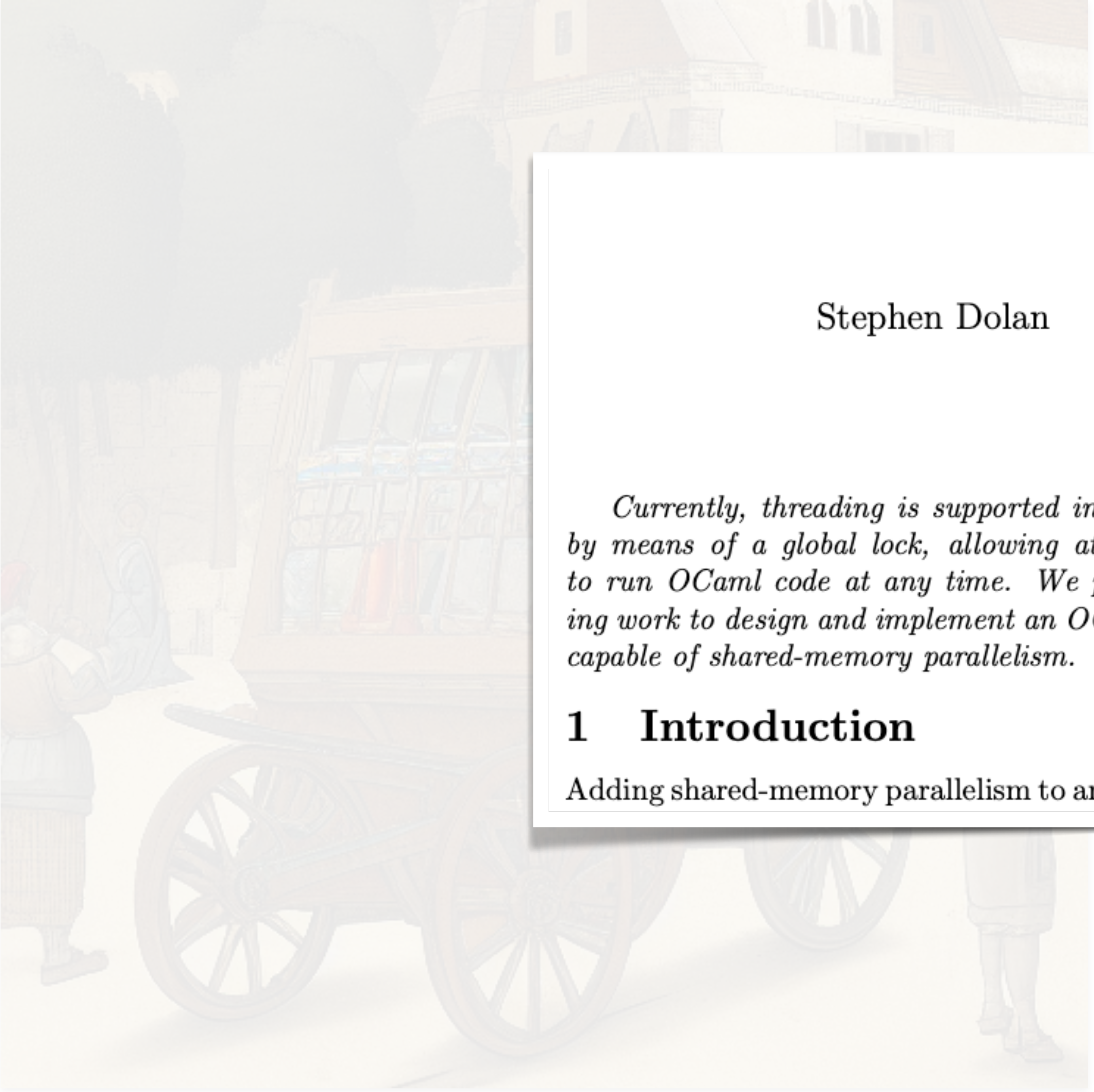


Parallel Allocator & GC

Medieval garbage truck according to Stable Diffusion



Parallel Allocator & GC



Multicore OCaml — OCaml '14

Stephen Dolan   Leo White   Anil Madhavapeddy

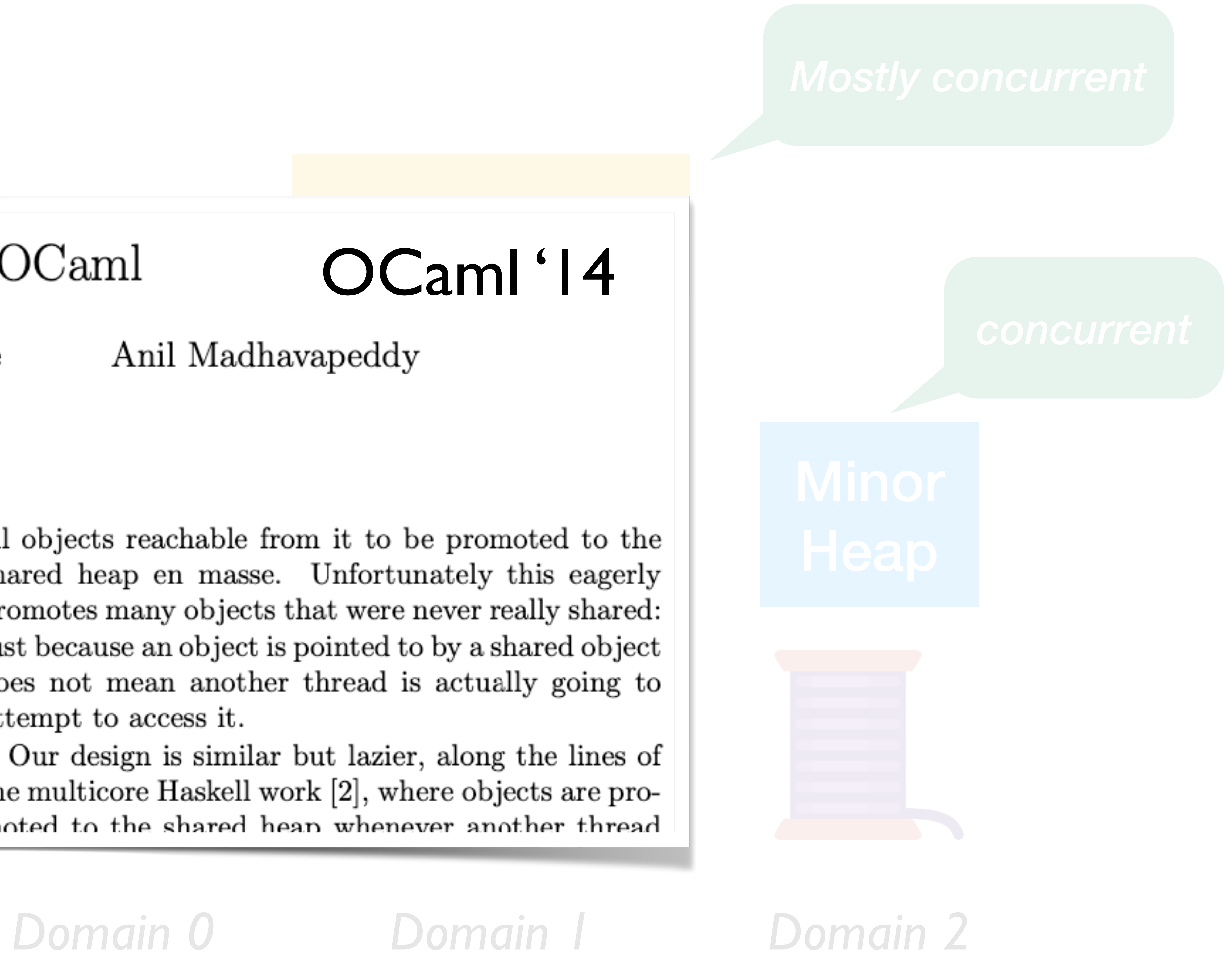
Currently, threading is supported in OCaml only by means of a global lock, allowing at most thread to run OCaml code at any time. We present ongoing work to design and implement an OCaml runtime capable of shared-memory parallelism.

1 Introduction

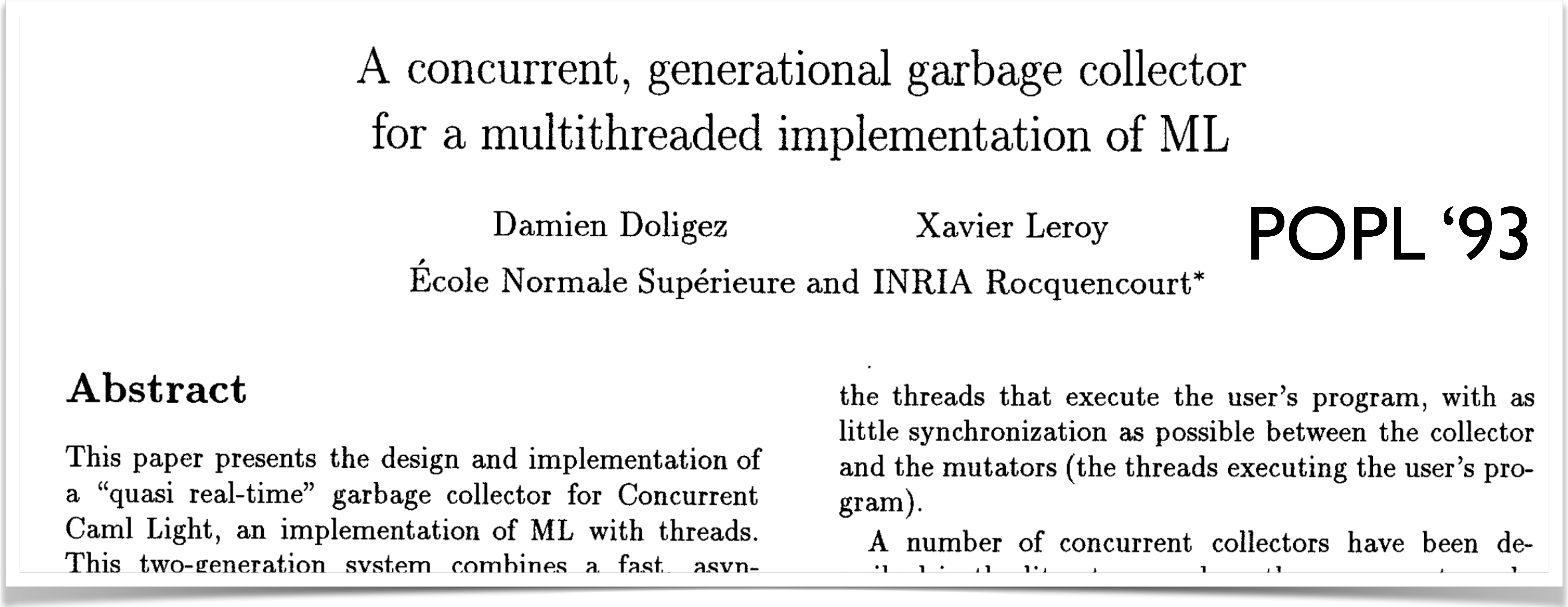
Adding shared-memory parallelism to an existing lan-

all objects reachable from it to be promoted to the shared heap en masse. Unfortunately this eagerly promotes many objects that were never really shared: just because an object is pointed to by a shared object does not mean another thread is actually going to attempt to access it.

Our design is similar but lazier, along the lines of the multicore Haskell work [2], where objects are promoted to the shared heap whenever another thread



Parallel Allocator & GC



A concurrent, generational garbage collector
for a multithreaded implementation of ML

Damien Doligez   Xavier Leroy

École Normale Supérieure and INRIA Rocquencourt*

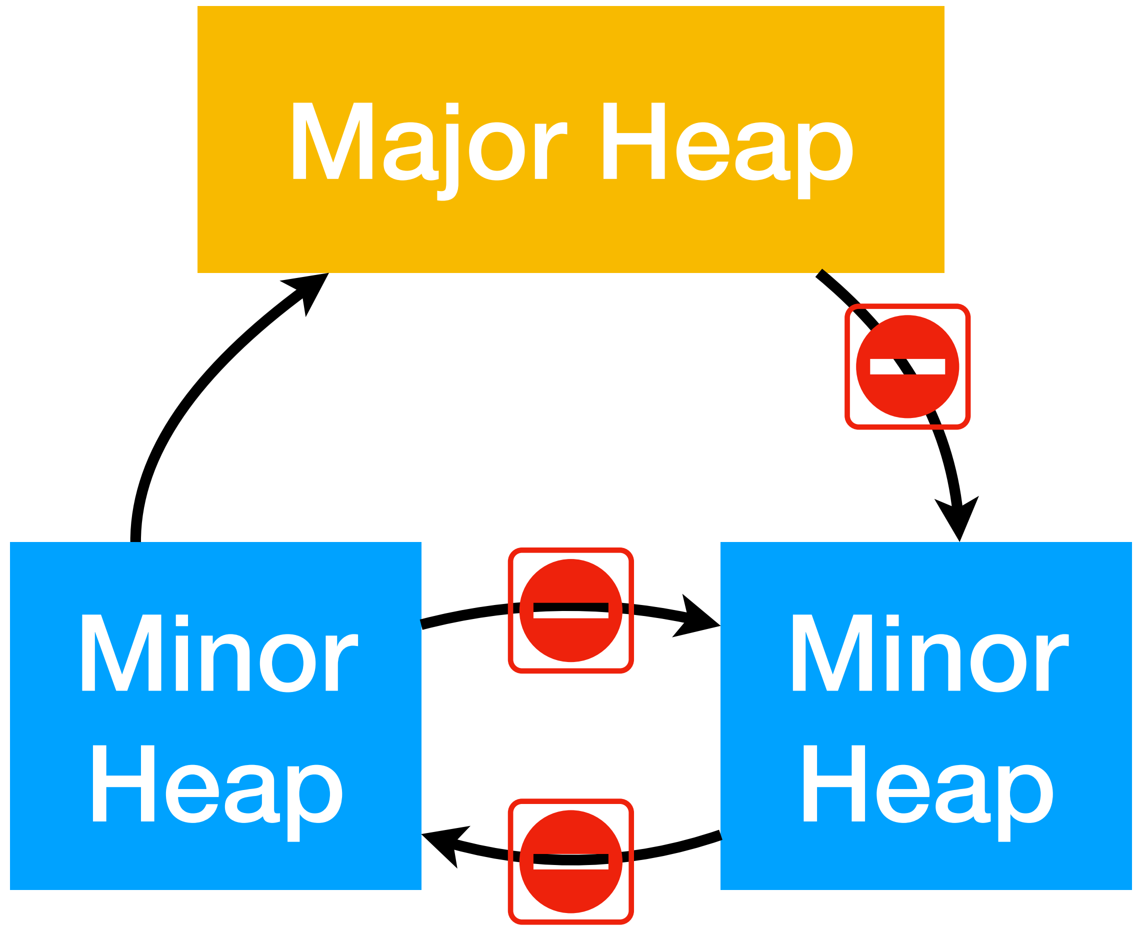
Abstract

This paper presents the design and implementation of a "quasi real-time" garbage collector for Concurrent Caml Light, an implementation of ML with threads. This two-generation system combines a fast, asyn-

the threads that execute the user's program, with as little synchronization as possible between the collector and the mutators (the threads executing the user's program).

A number of concurrent collectors have been de-

POPL '93



Parallel Allocator & GC

Multicore Garbage Collection with Local Heaps

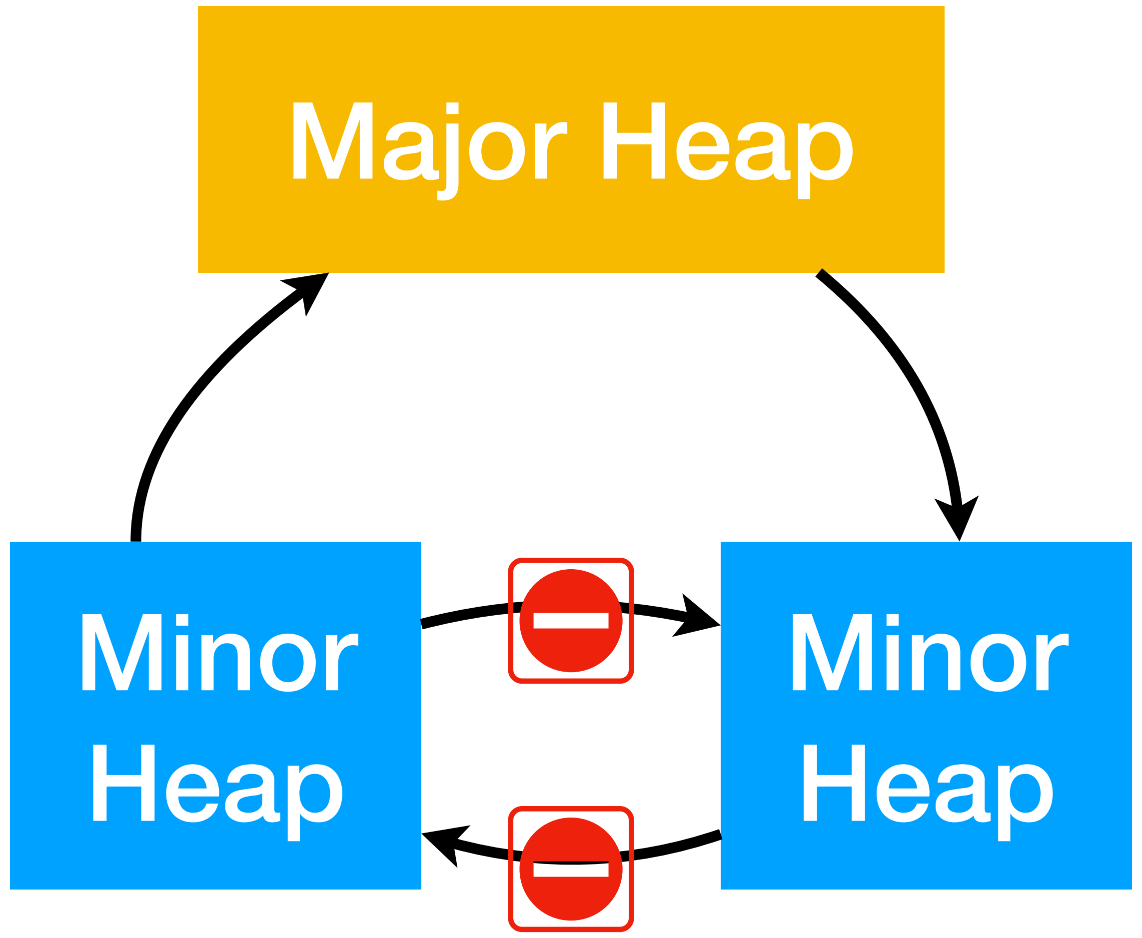
Simon Marlow
Microsoft Research, Cambridge, U.K.
simonmar@microsoft.com

Simon Peyton Jones
Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com

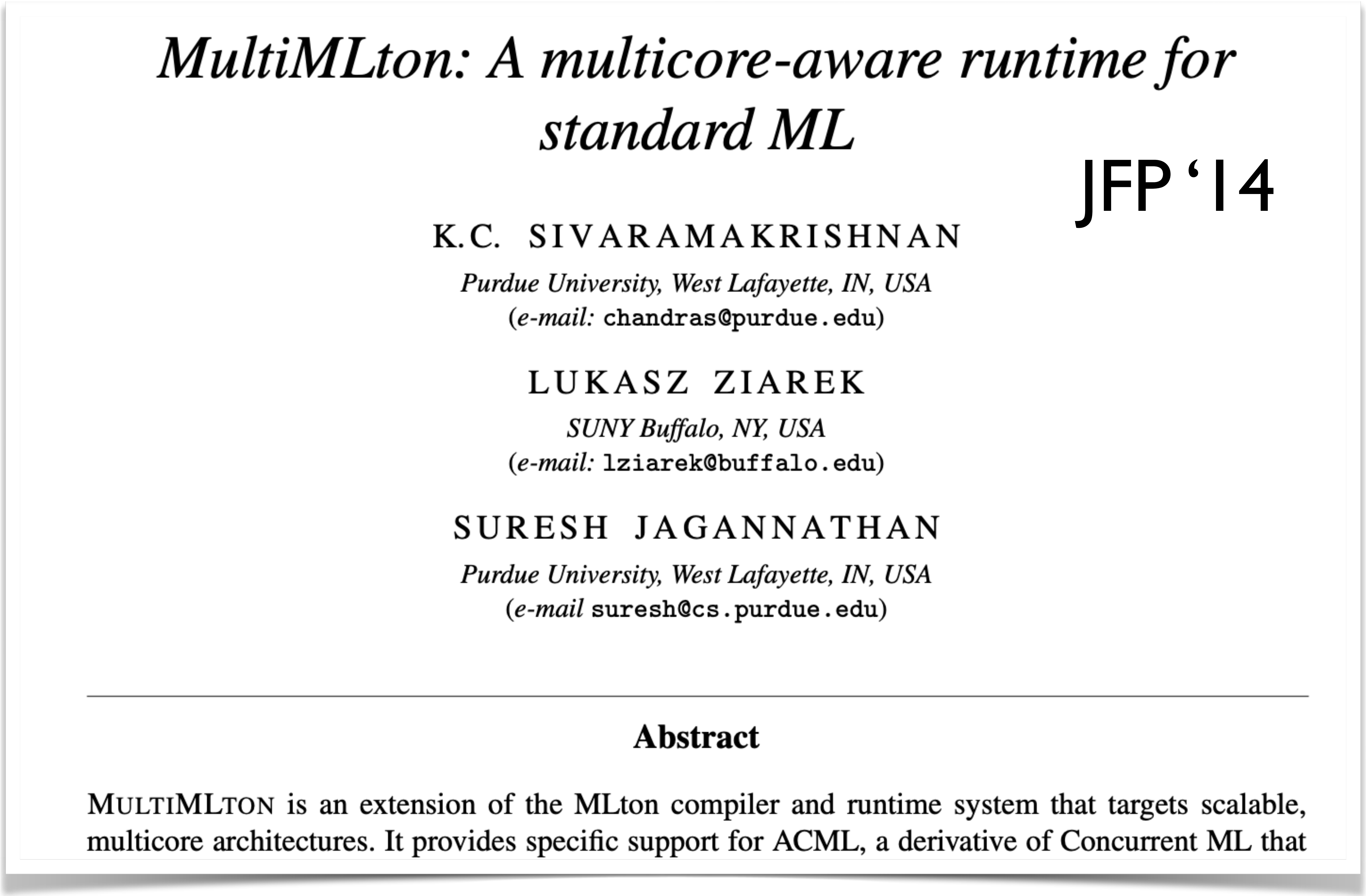
ISMM ‘11

Abstract

In a parallel, shared-memory, language with a garbage collected heap, it is desirable for each processor to perform minor garbage collections independently. Although obvious, it is difficult to make this idea pay off in practice, especially in languages where muta- to design collectors in which each processor has a private heap that can be collected independently without synchronising with the other processors; there is also a global heap for shared data. Some of the existing designs are based on static analyses to identify objects whose references never escape the current thread and



Parallel Allocator & GC


MultiMLton: A multicore-aware runtime for standard ML

K.C. SIVARAMAKRISHNAN
Purdue University, West Lafayette, IN, USA
(*e-mail:* chandras@purdue.edu)

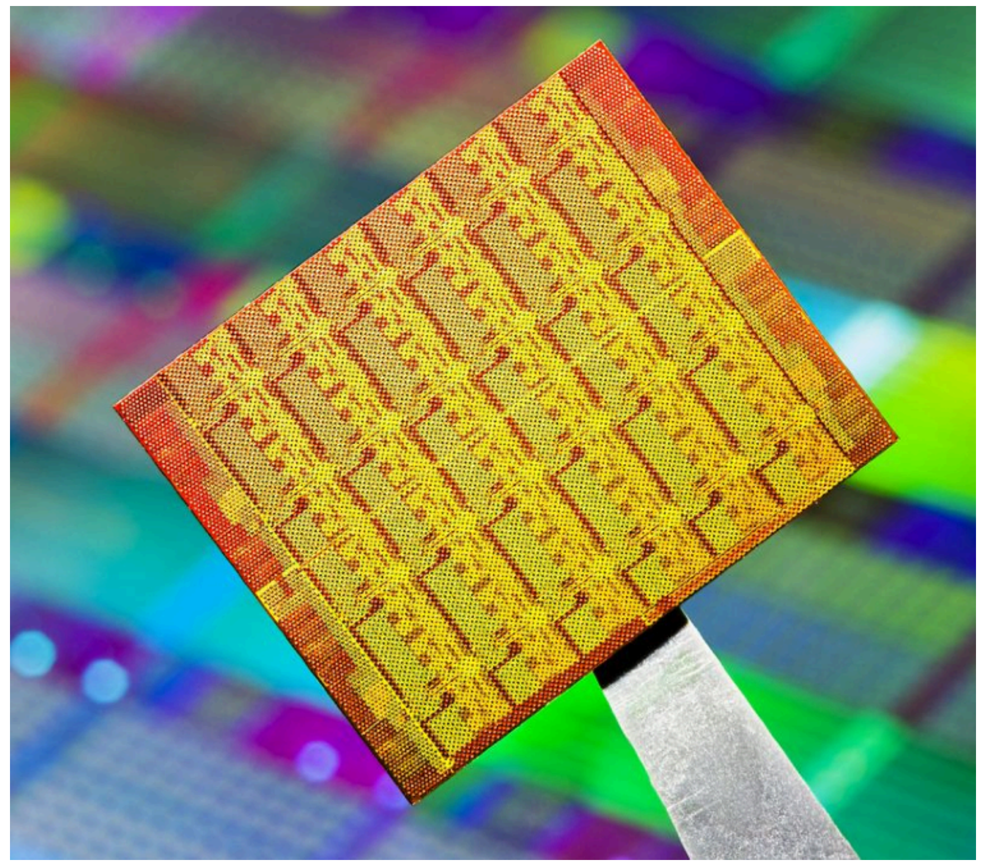
LUKASZ ZIAREK
SUNY Buffalo, NY, USA
(*e-mail:* lziarek@buffalo.edu)

SURESH JAGANNATHAN
Purdue University, West Lafayette, IN, USA
(*e-mail* suresh@cs.purdue.edu)

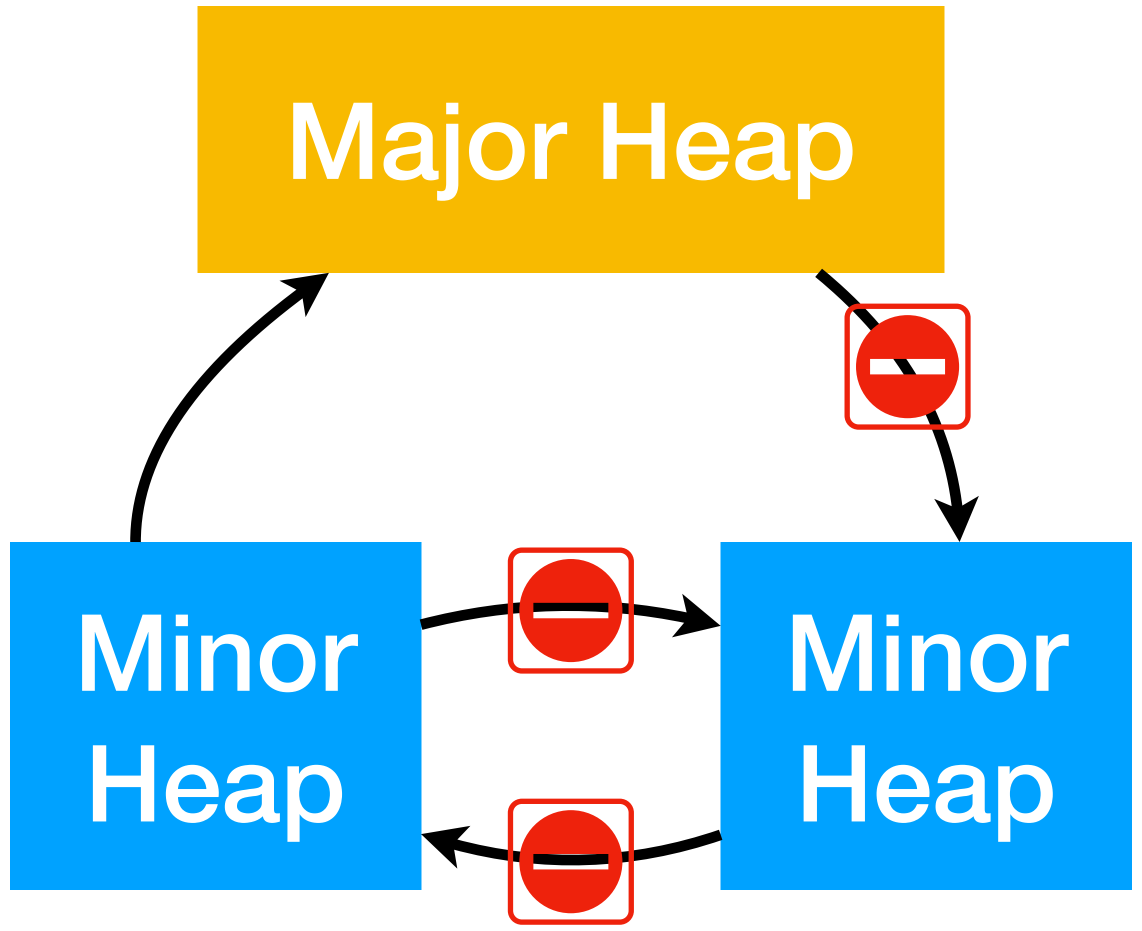
Abstract

MULTIMLTON is an extension of the MLton compiler and runtime system that targets scalable, multicore architectures. It provides specific support for ACML, a derivative of Concurrent ML that

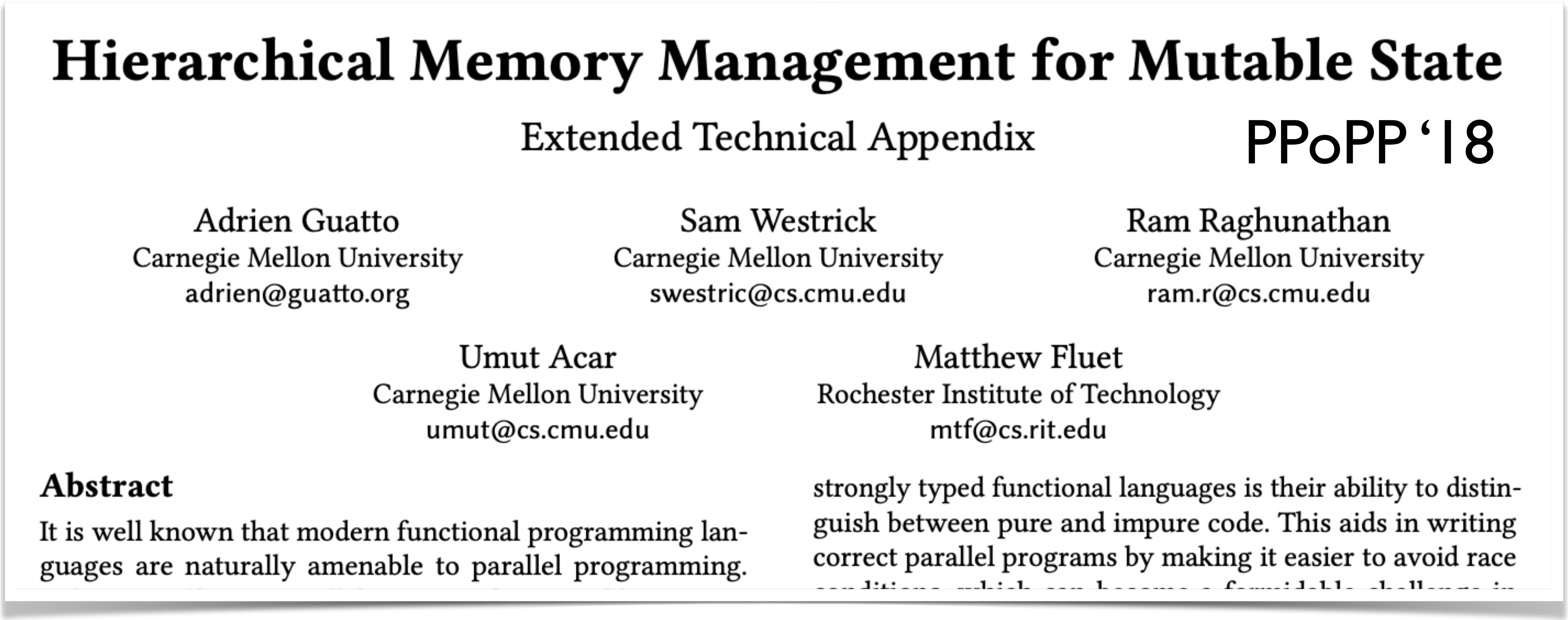
JFP '14



Intel Single-chip Cloud Computer (SCC)



Parallel Allocator & GC


Hierarchical Memory Management for Mutable State

Extended Technical Appendix

PPoPP '18

Adrien Guatto
Carnegie Mellon University
adrien@guatto.org

Sam Westrick
Carnegie Mellon University
swestric@cs.cmu.edu

Ram Raghunathan
Carnegie Mellon University
ram.r@cs.cmu.edu

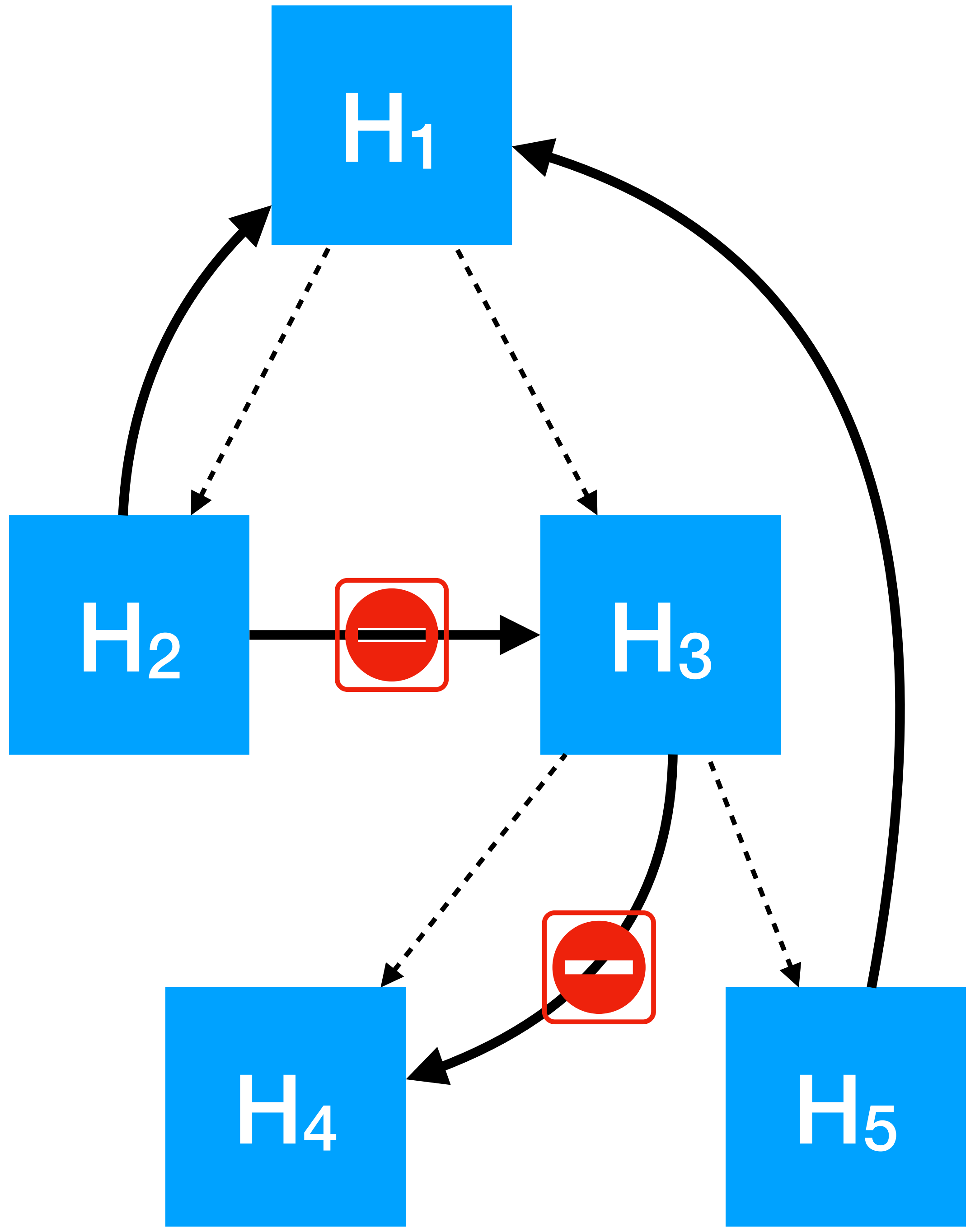
Umut Acar
Carnegie Mellon University
umut@cs.cmu.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Abstract

It is well known that modern functional programming languages are naturally amenable to parallel programming. [...] strongly typed functional languages is their ability to distinguish between pure and impure code. This aids in writing correct parallel programs by making it easier to avoid race

MaPLe



disentanglement

Parallel Allocator & GC

Entanglement Detection with Near-Zero Cost

ICFP '22

SAM WESTRICK, Carnegie Mellon University, USA
JATIN ARORA, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA

Recent research on parallel functional programming has culminated in a provably efficient (in work and space) parallel memory manager, which has been incorporated into the MPL (MaPLe) compiler for Parallel ML and

Parallel Allocator & GC

ML 2022

Thu 15 Sep 2022 09:00 - 09:50 at M1 - Language Design

★ Efficient and Scalable Parallel Functional Programming Through Disentanglement

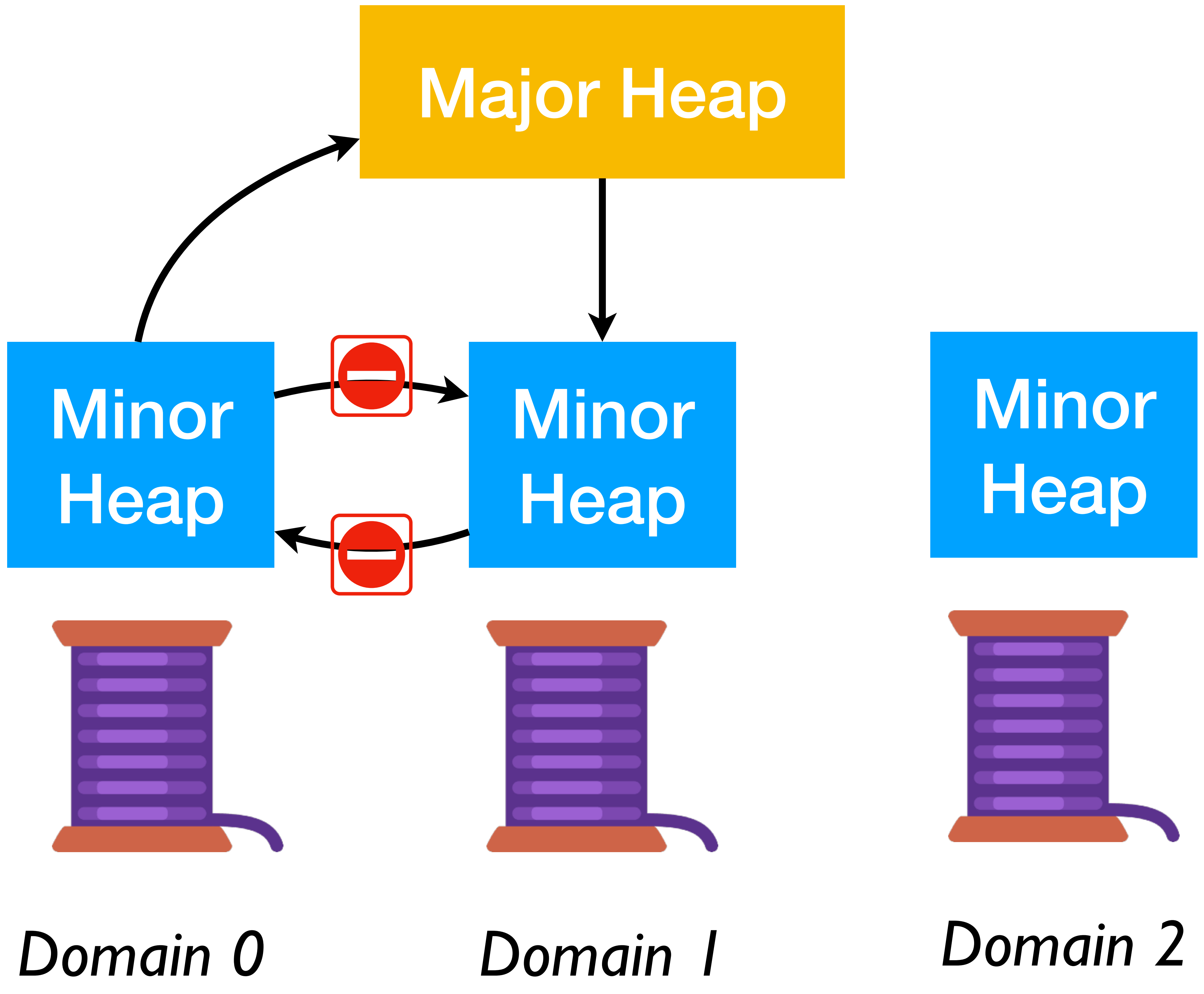
Researchers have argued for decades that functional programming simplifies parallel programming, in particular by helping programmers avoid difficult concurrency bugs arising from destructive in-place updates. However, parallel functional languages have historically underperformed in comparison to parallel programs written in lower-level languages. The difficulty is that functional programs have high demand for memory, and this demand only grows with parallelism, causing traditional parallel memory management techniques to buckle under the increased pressure.

Recent work has made progress on this problem by identifying a broadly applicable memory property called disentanglement. To exploit disentanglement for improved efficiency and scalability, we show how to partition memory into a tree of heaps, mirroring the dynamic nesting of parallel tasks. This design allows for task-local allocations and garbage collections to proceed independently and in parallel. The result is a provably efficient parallel memory manager.

These ideas have been incorporated into the MPL ("maple") compiler for Parallel ML, which offers practical efficiency and scalability for parallel functional programs. Our empirical evaluations show that, at scale (on 72 processors), MPL outperforms modern implementations of both functional and imperative languages, including Java and Go. Additionally, we show that MPL is competitive with low-level, memory-unsafe languages such as C++, in terms of both space and time.

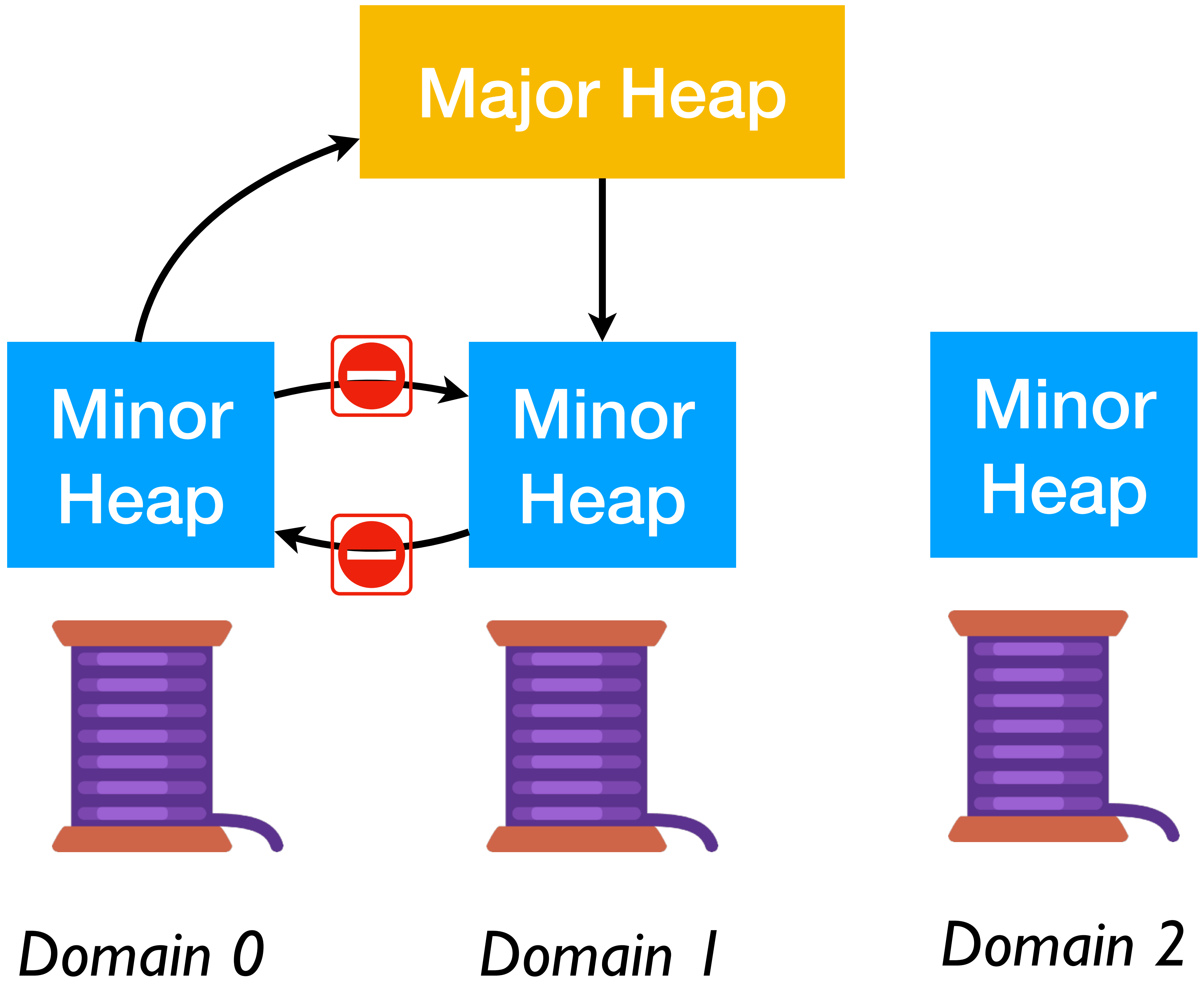
Sam Westrick
Carnegie Mellon University
United States

Parallel Allocator & GC



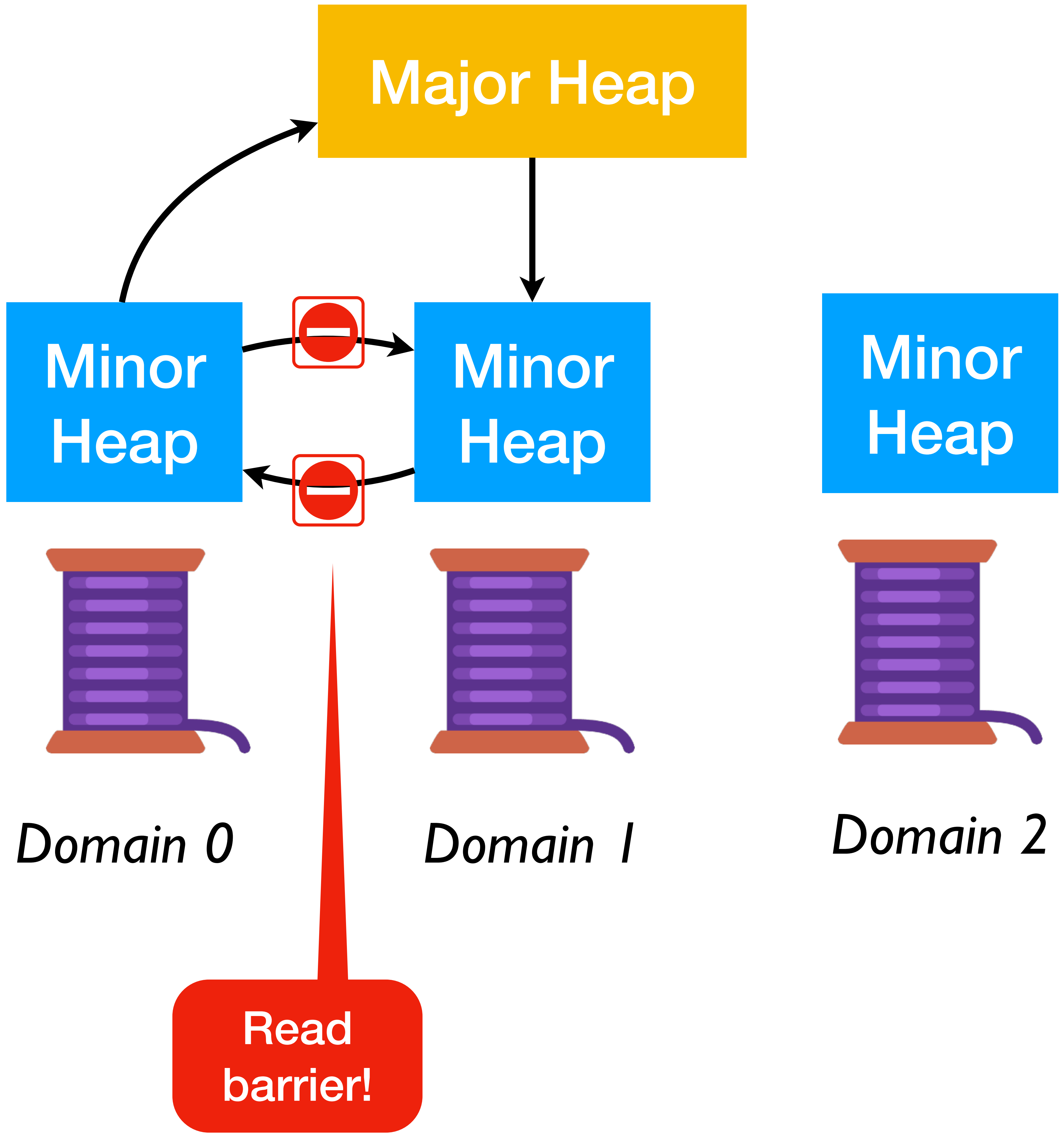
- Excellent *scalability* on 128-cores
  - Also maintains low latency on large core counts
- Mostly retains sequential *latency*, *throughput* and *memory usage* characteristics

Parallel Allocator & GC



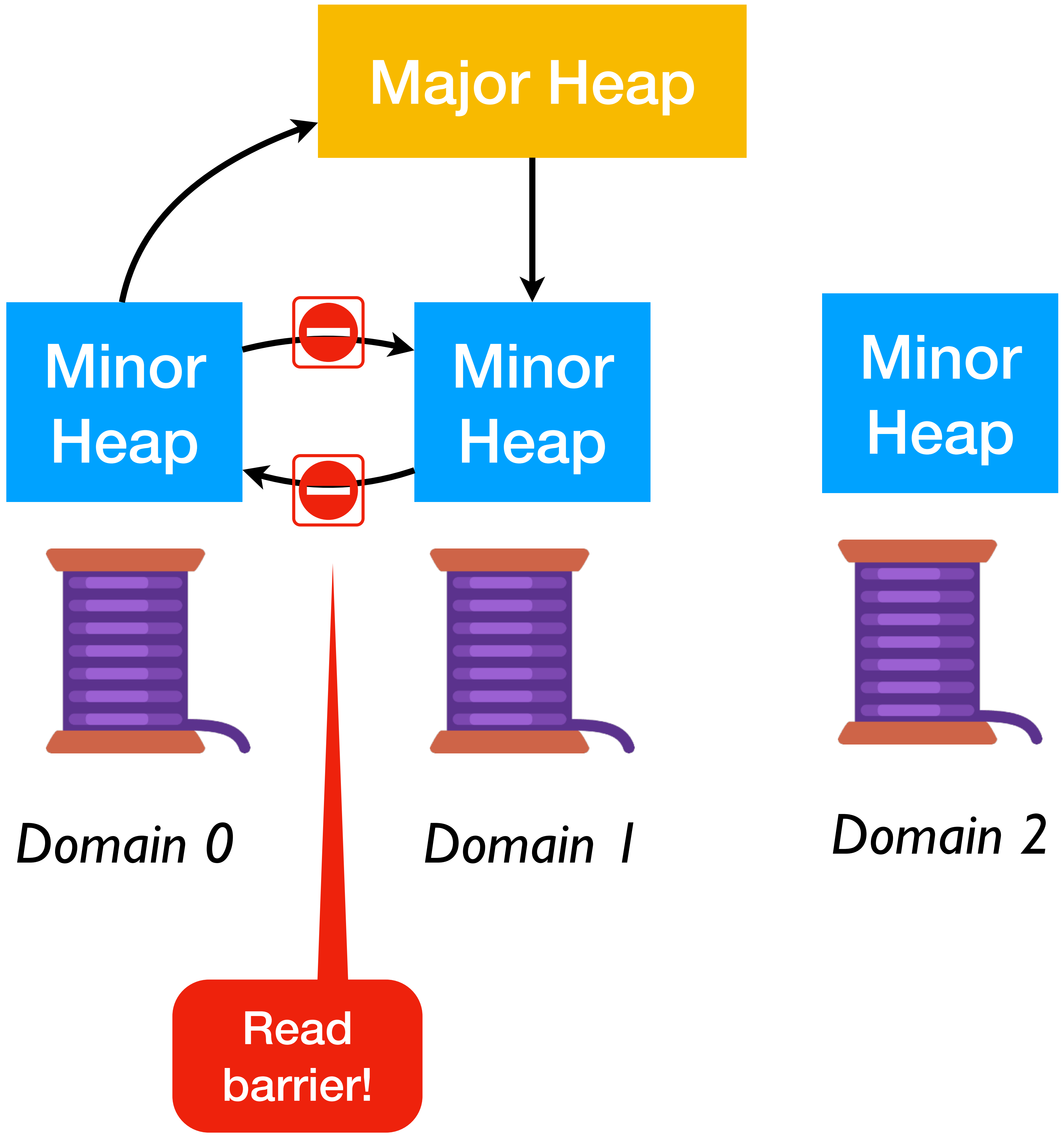
But …

Parallel Allocator & GC



But …

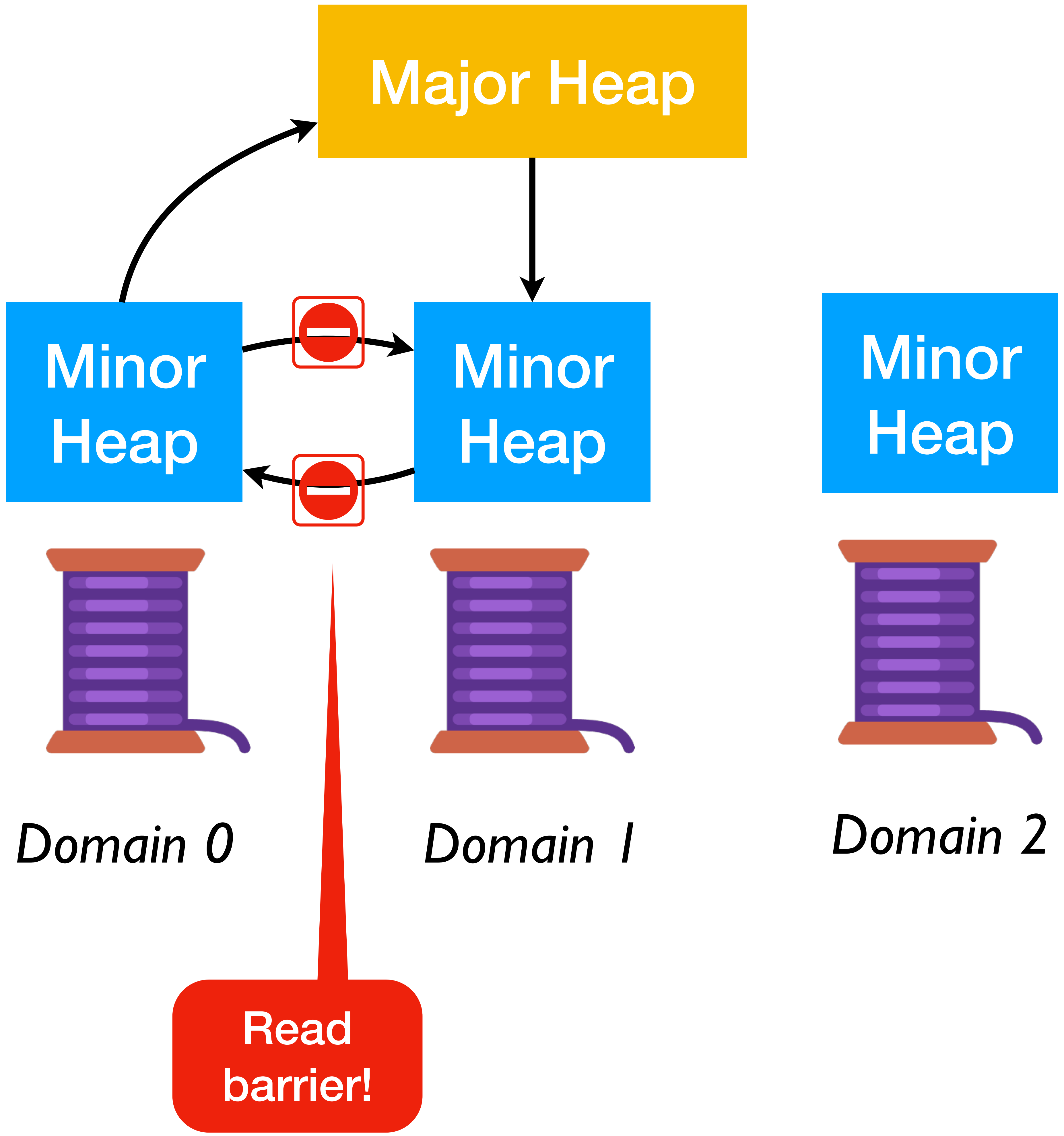
Parallel Allocator & GC



But …

- Read barrier
  - ✦ Only a branch on the OCaml side for reads
  - ✦ *Read are now GC safe points*
  - ✦ Breaks the C FFI invariants about when GC may be performed

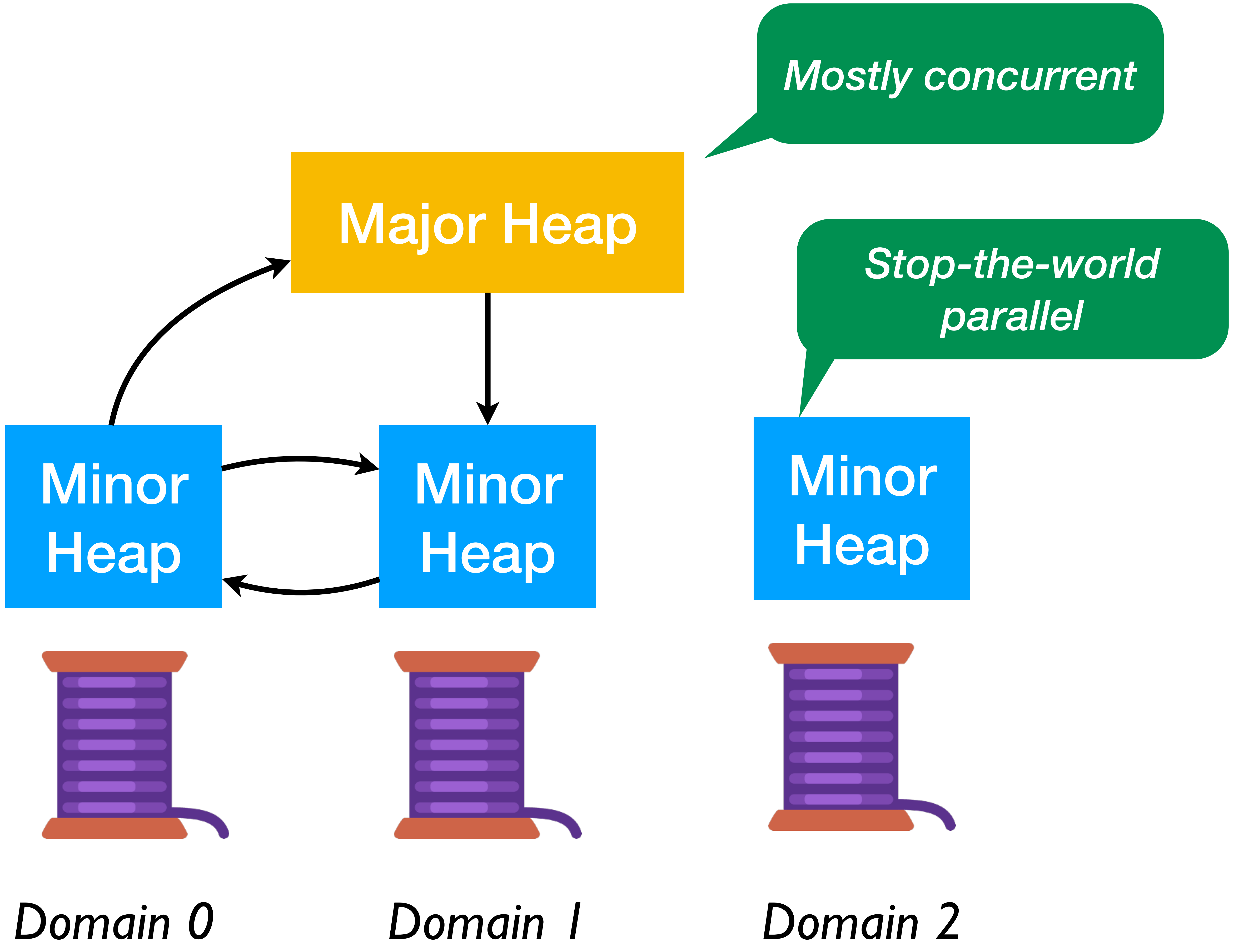
Parallel Allocator & GC



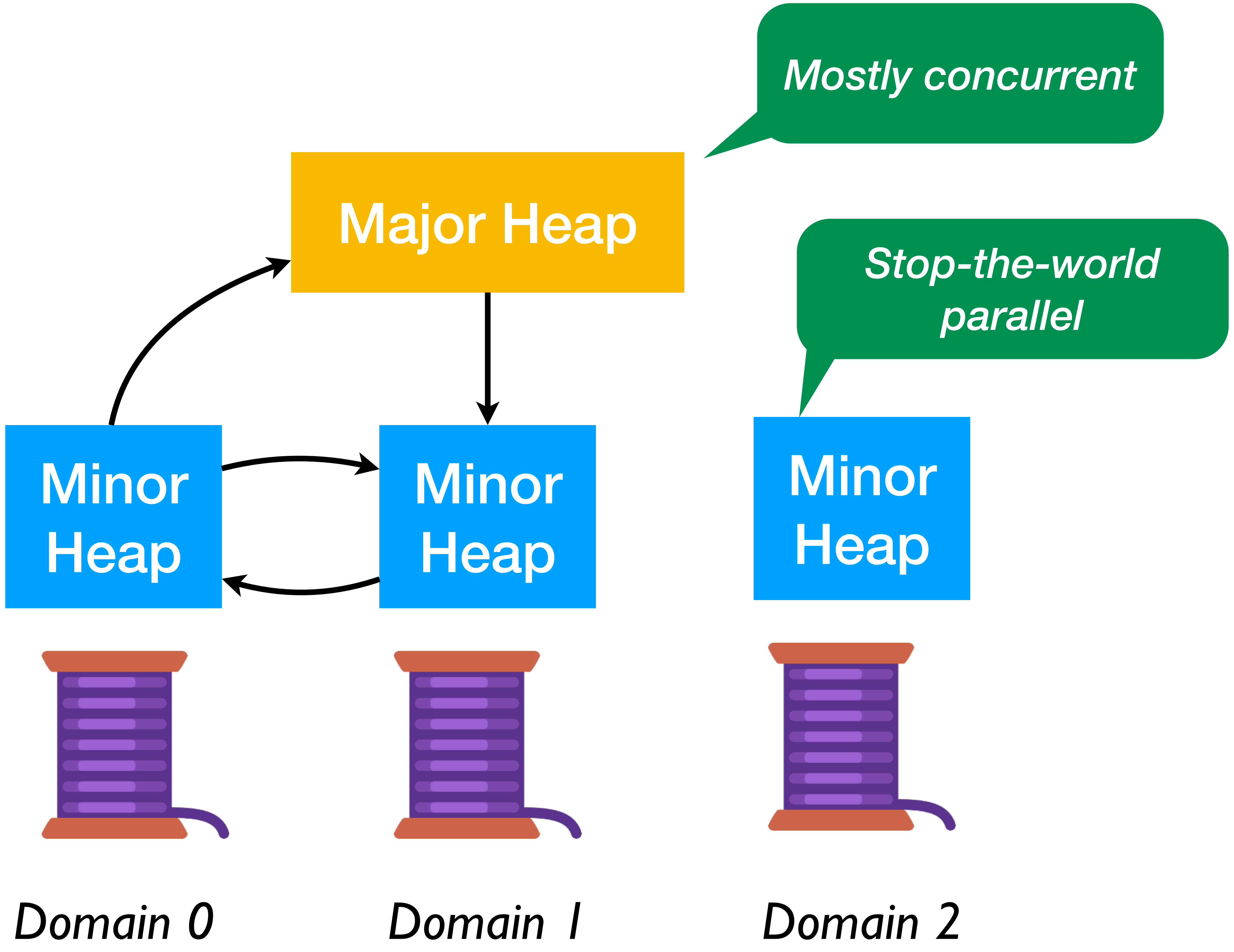
But …

- Read barrier
  - ✦ Only a branch on the OCaml side for reads
  - ✦ *Read are now GC safe points*
  - ✦ Breaks the C FFI invariants about when GC may be performed
- No push-button fix!
  - ✦ Lots of packages in the ecosystem broke.

Back to the drawing board (~2019)

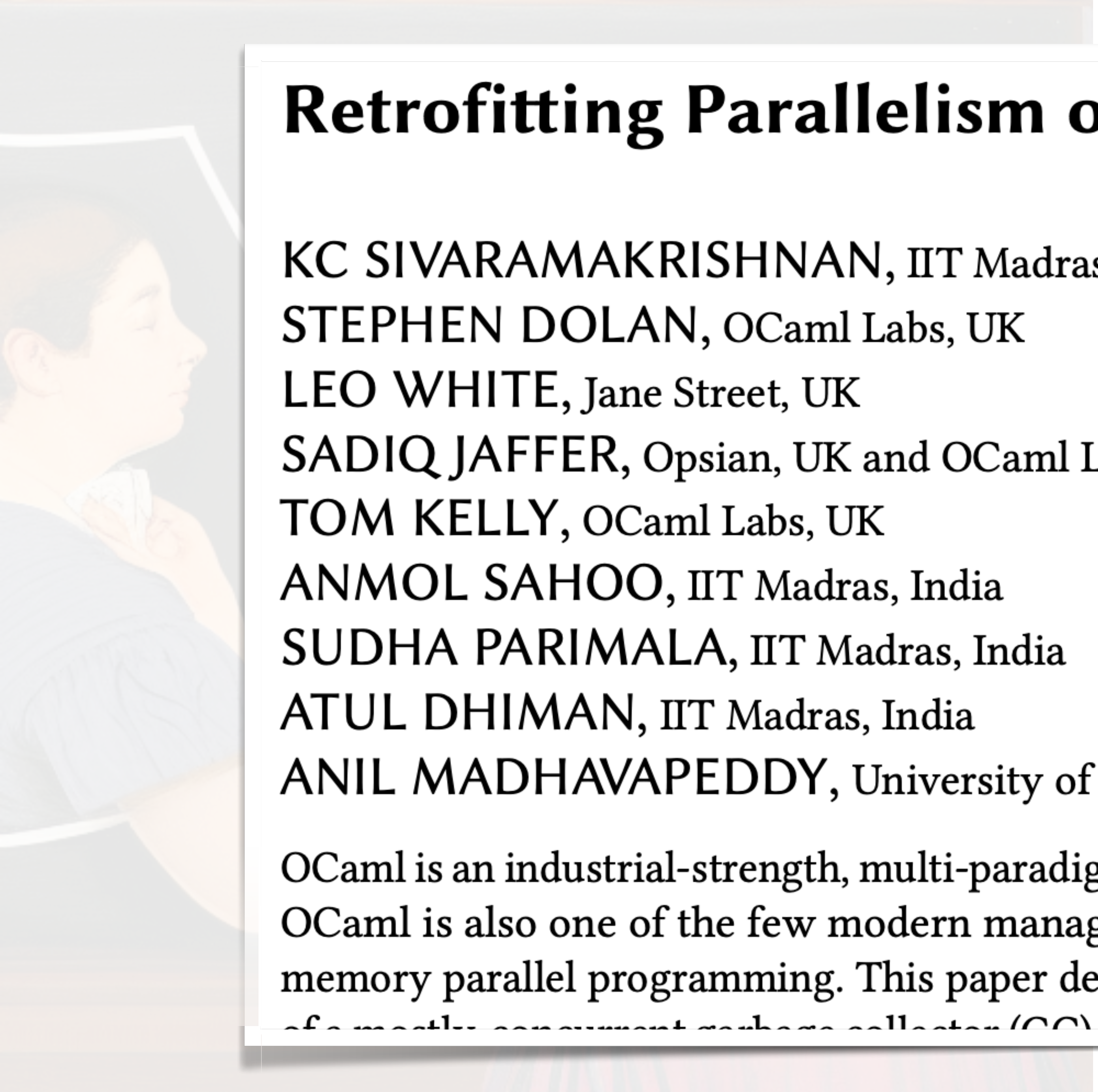


Back to the drawing board (~2019)



Bring 128-domains to a stop is surprisingly fast

Back to the drawing board (~2019)



Retrofitting Parallelism onto OCaml

ICFP ‘20

KC SIVARAMAKRISHNAN, IIT Madras, India
STEPHEN DOLAN, OCaml Labs, UK
LEO WHITE, Jane Street, UK
SADIQ JAFFER, Opsian, UK and OCaml Labs, UK
TOM KELLY, OCaml Labs, UK
ANMOL SAHOO, IIT Madras, India
SUDHA PARIMALA, IIT Madras, India
ATUL DHIMAN, IIT Madras, India
ANIL MADHAVAPEDDY, University of Cambridge Computer Laboratory, UK and OCaml Labs, UK

OCaml is an industrial-strength, multi-paradigm programming language, widely used in industry and academia. OCaml is also one of the few modern managed system programming languages to lack support for shared memory parallel programming. This paper describes the design, a full-fledged implementation and evaluation

On average, < 1% performance overhead for sequential programs

Data Races

- **Data Race:** When two threads perform *unsynchronised* access and at least one is a write.
  - Non-SC behaviour due to *compiler optimisations* and *relaxed hardware.*

Data Races

- **Data Race:** When two threads perform *unsynchronised* access and at least one is a write.
  - ✦ Non-SC behaviour due to *compiler optimisations* and *relaxed hardware.*
- *Enforcing SC behaviour slows down sequential programs!*
  - ✦ 85% on ARM64, 41% on PowerPC

Data Races

- **Data Race:** When two threads perform *unsynchronised* access and at least one is a write.
  - Non-SC behaviour due to *compiler optimisations* and *relaxed hardware.*
- *Enforcing SC behaviour slows down sequential programs!*
  - 85% on ARM64, 41% on PowerPC

OCaml needed a relaxed memory model

Second-mover Advantage

- Learn from the other language memory models

Second-mover Advantage

- Learn from the other language memory models

Swift

- DRF-SC, but catch-fire semantics on data races

Well-typed OCaml programs don't go wrong

Second-mover Advantage

- Learn from the other language memory models

Swift

- DRF-SC, but catch-fire semantics on data races

Well-typed OCaml programs don't go wrong

Java

- DRF-SC + no crash under data races
  - ✦ But scope of race is not limited in *time*

Second-mover Advantage

- Learn from the other language memory models

Swift

- DRF-SC, but catch-fire semantics on data races

Well-typed OCaml programs don't go wrong

Java

- DRF-SC + no crash under data races
  - ✦ But scope of race is not limited in *time*

- No data races by construction
  - ✦ Unsafe code memory model is ~C++11

Second-mover Advantage

- Learn from the other language memory models

- DRF-SC, but catch-fire semantics on data races

Well-typed OCaml programs don't go wrong

- DRF-SC + no crash under data races
  - ✦ But scope of race is not limited in *time*

- No data races by construction
  - ✦ Unsafe code memory model is ~C++11

Advantage: No Multicore OCaml programs in the wild!

OCaml memory model (~2017)

- Simple (*comprehensible!*) operational memory model
  - ✦ Only atomic and non-atomic locations
  - ✦ DRF-SC
  - ✦ No "out of thin air" values
  - ✦ Squeeze at most perf ⇒ write that module in C, C++ or Rust.

OCaml memory model (~2017)

- Simple (*comprehensible!*) operational memory model
  - ✦ Only atomic and non-atomic locations
  - ✦ DRF-SC
  - ✦ No "out of thin air" values
  - ✦ Squeeze at most perf ⇒ write that module in C, C++ or Rust.

GO 1.19

OCaml memory model (~2017)

- Simple (*comprehensible!*) operational memory model
  - ✦ Only atomic and non-atomic locations
  - ✦ DRF-SC
  - ✦ No "out of thin air" values
  - ✦ Squeeze at most perf ⇒ write that module in C, C++ or Rust.

GO 1.19

OCaml memory model (~2017)

- Simple (*comprehensible!*) operational memory model
  - ✦ Only atomic and non-atomic locations
  - ✦ DRF-SC
  - ✦ No "out of thin air" values
  - ✦ Squeeze at most perf ⇒ write that module in C, C++ or Rust.

GO 1.19

- **Key innovation:** *Local data race freedom*
  - ✦ Permits compositional reasoning

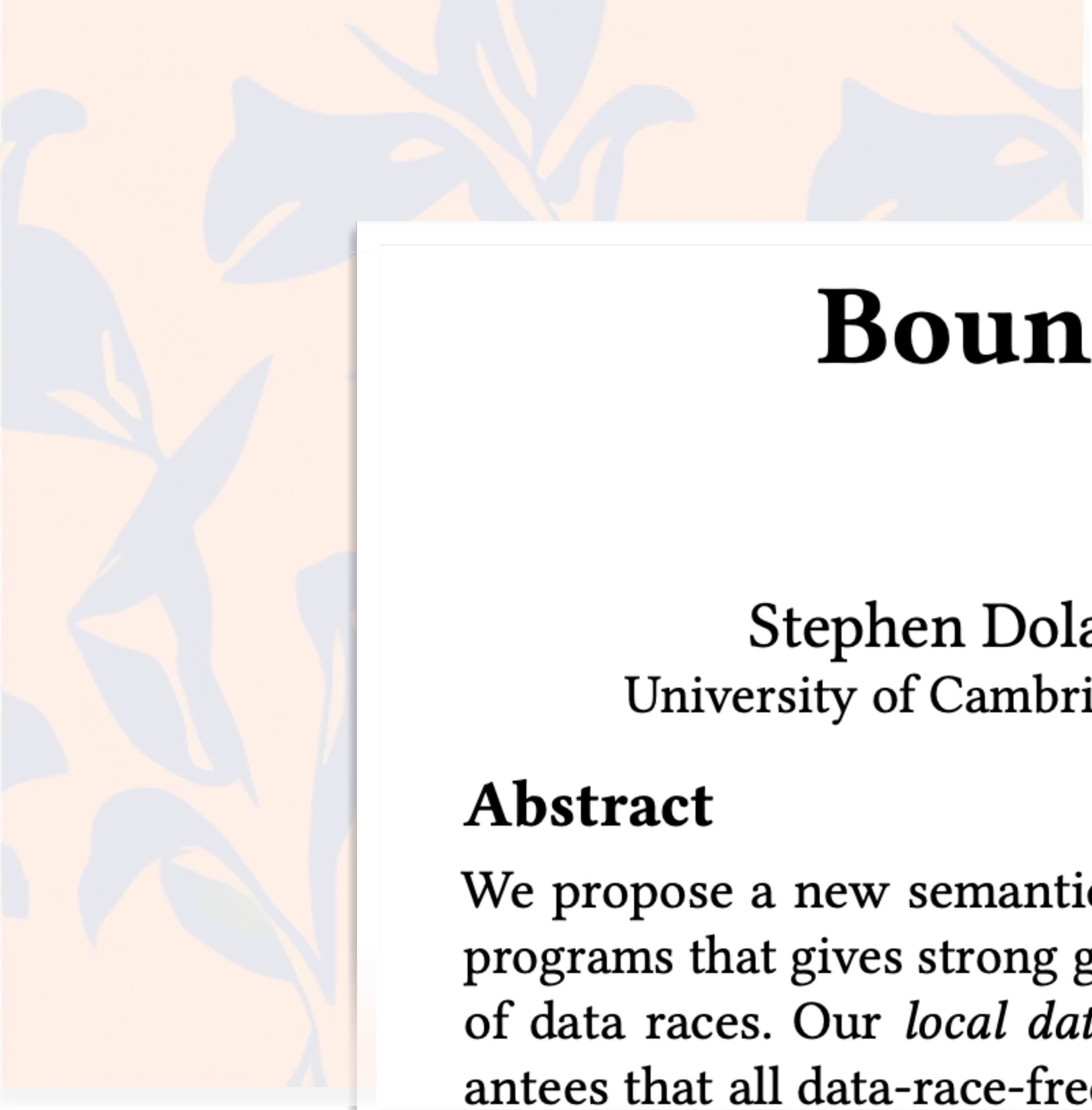
OCaml memory model (~2017)

- Simple (*comprehensible!*) operational memory model
  - ✦ Only atomic and non-atomic locations
  - ✦ DRF-SC
  - ✦ No "out of thin air" values
  - ✦ Squeeze at most perf ⇒ write that module in C, C++ or Rust.

GO 1.19

- **Key innovation:** *Local data race freedom*
  - ✦ Permits compositional reasoning
- Performance impact
  - ✦ Free on x86 and < 1% on ARM

OCaml memory model (~2017)



- Simple (*comprehensible!*) operational memory model

Bounding Data Races in Space and Time

(Extended version, with appendices)

PLDI '18

Stephen Dolan, University of Cambridge, UK

KC Sivaramakrishnan, University of Cambridge, UK

Anil Madhavapeddy, University of Cambridge, UK

Abstract

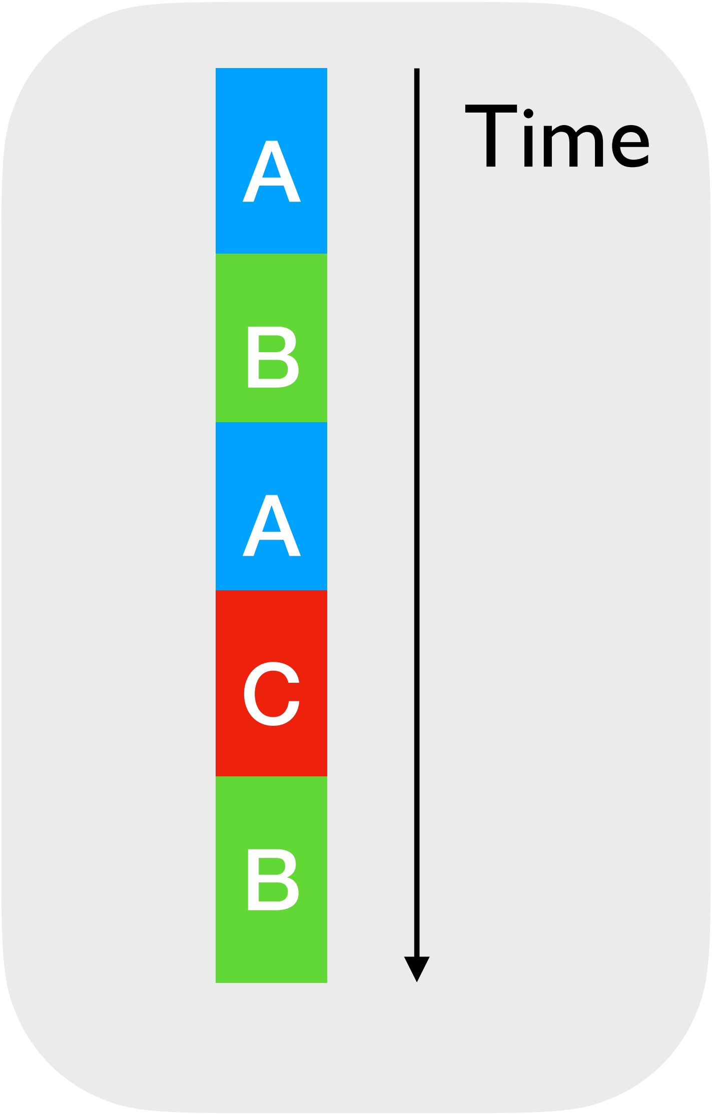
We propose a new semantics for shared-memory parallel programs that gives strong guarantees even in the presence of data races. Our *local data race freedom* property guarantees that all data-race-free portions of programs exhibit

The primary reasoning tools provided to programmers by these models are the *data-race-freedom (DRF) theorems*. Programmers are required to mark as *atomic* all variables used for synchronisation between threads, and to avoid *data races*, which are concurrent accesses (except concurrent reads) to

- Performance impact
  - ✦ Free on x86 and < 1% on ARM

Concurrency (~2015)

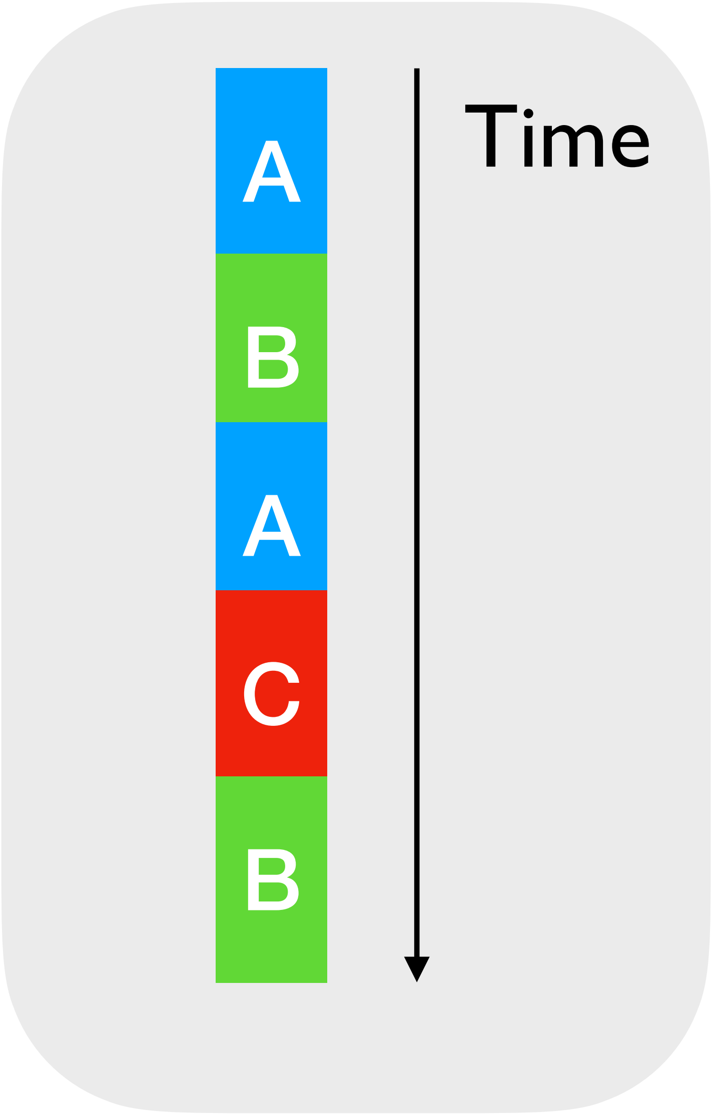
- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - Language-level threads



Overlapped execution

Concurrency (~2015)

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads

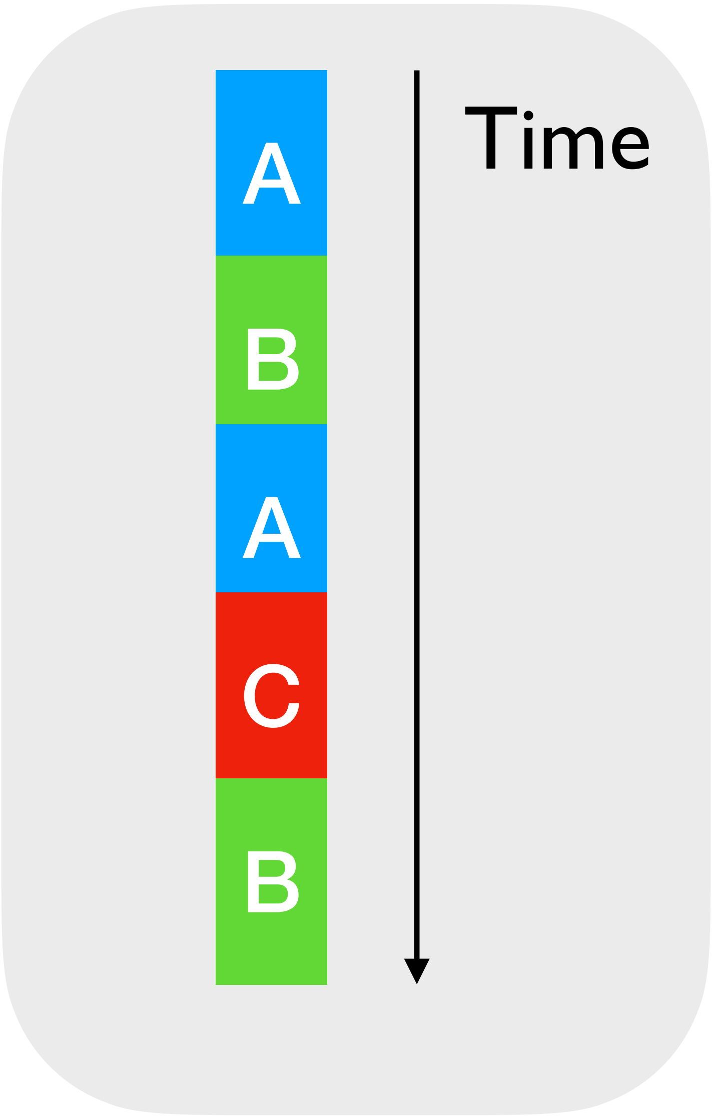


Overlapped execution

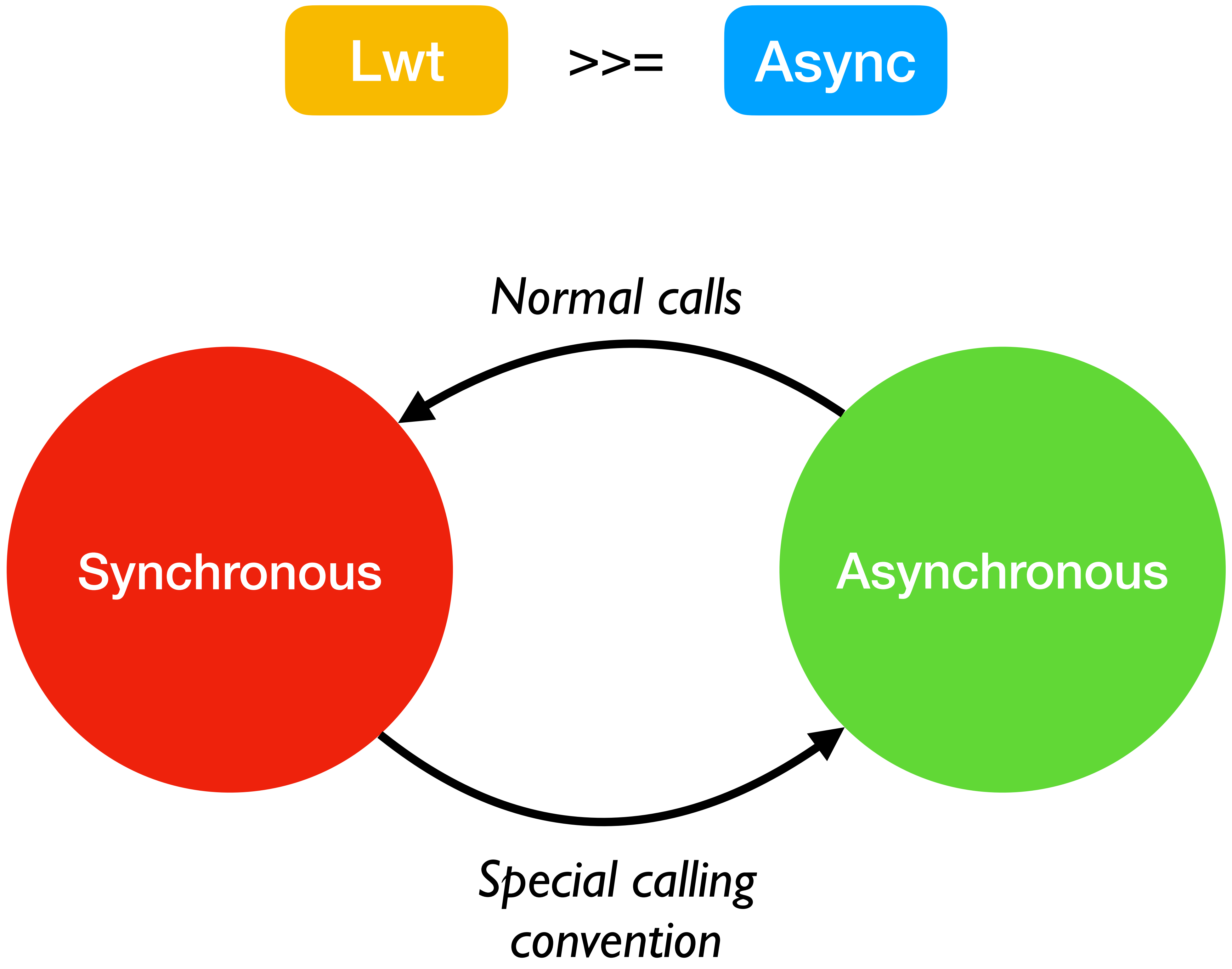
Lwt >>= Async

Concurrency (~2015)

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - Language-level threads



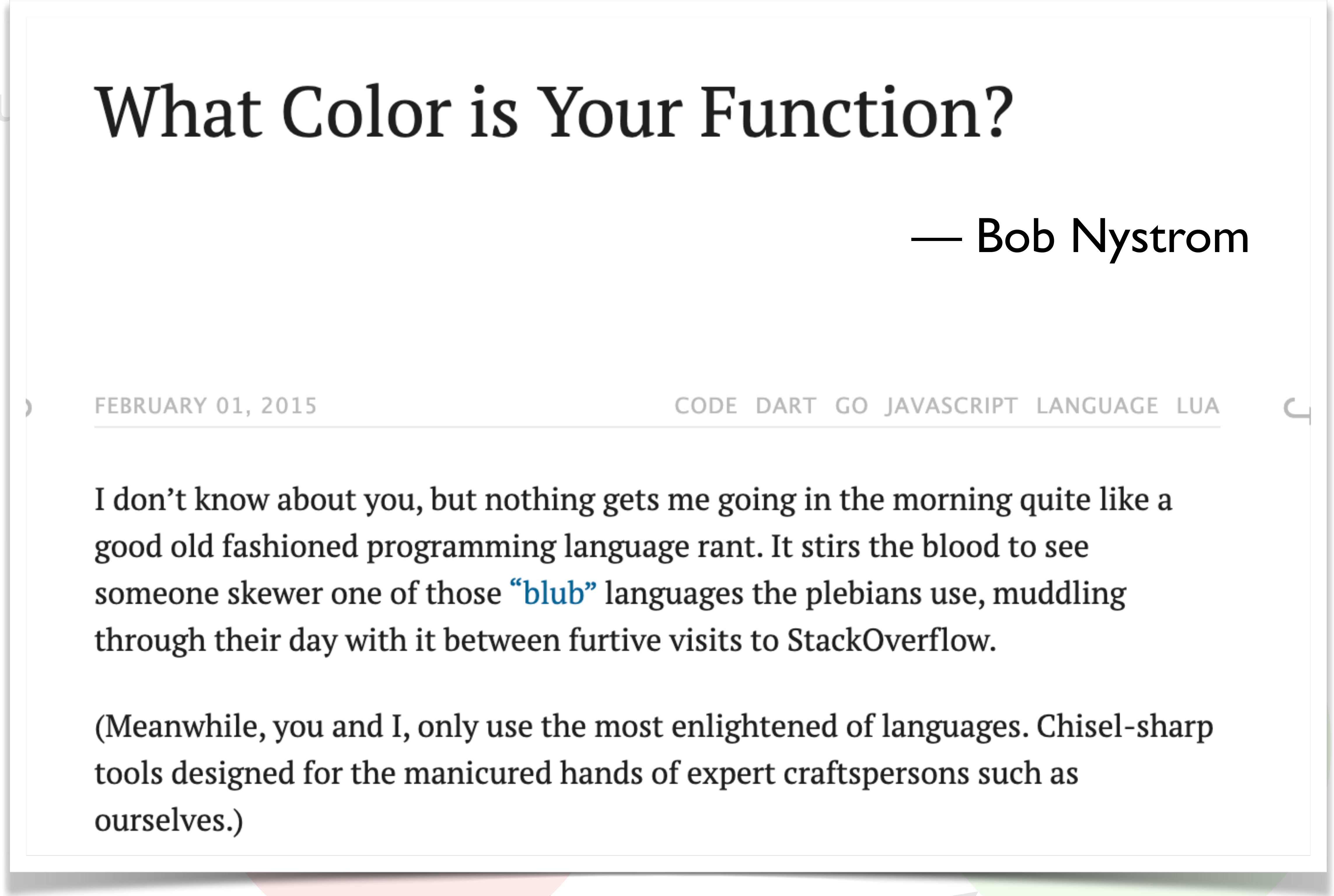
Overlapped execution



Concurrency (~2015)

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - Langu

Overlapped execution



What Color is Your Function?

— Bob Nystrom

FEBRUARY 01, 2015 CODE DART GO JAVASCRIPT LANGUAGE LUA

I don't know about you, but nothing gets me going in the morning quite like a good old fashioned programming language rant. It stirs the blood to see someone skewer one of those "blub" languages the plebians use, muddling through their day with it between furtive visits to StackOverflow.

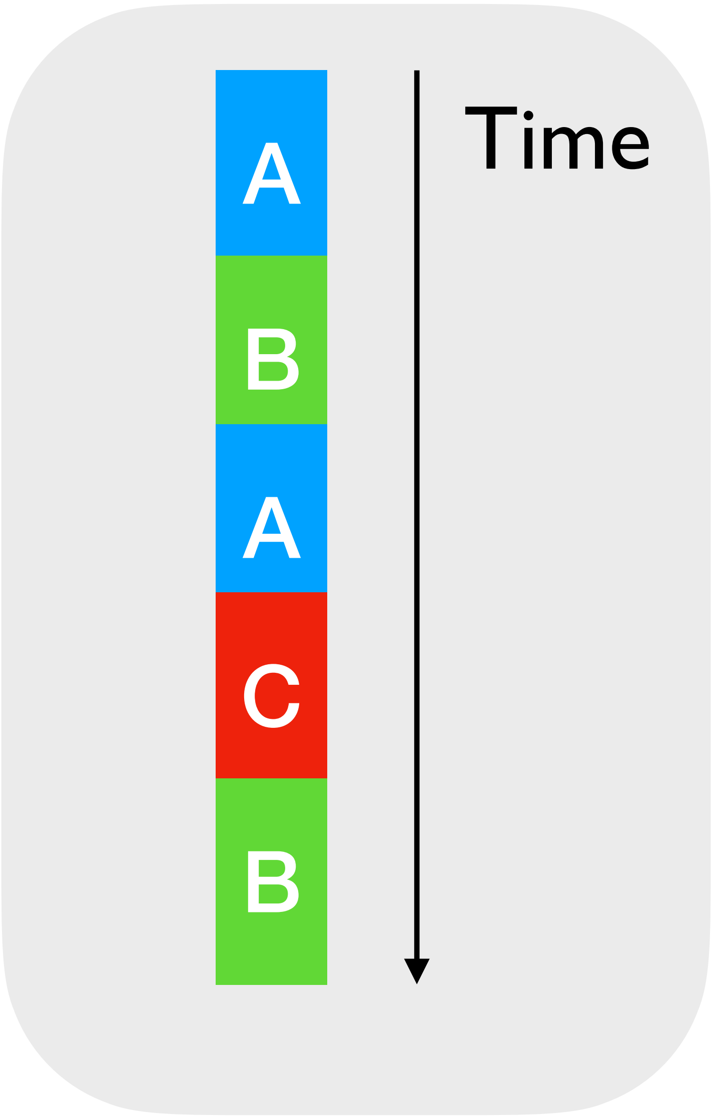
(Meanwhile, you and I, only use the most enlightened of languages. Chisel-sharp tools designed for the manicured hands of expert craftspersons such as ourselves.)

Special calling convention

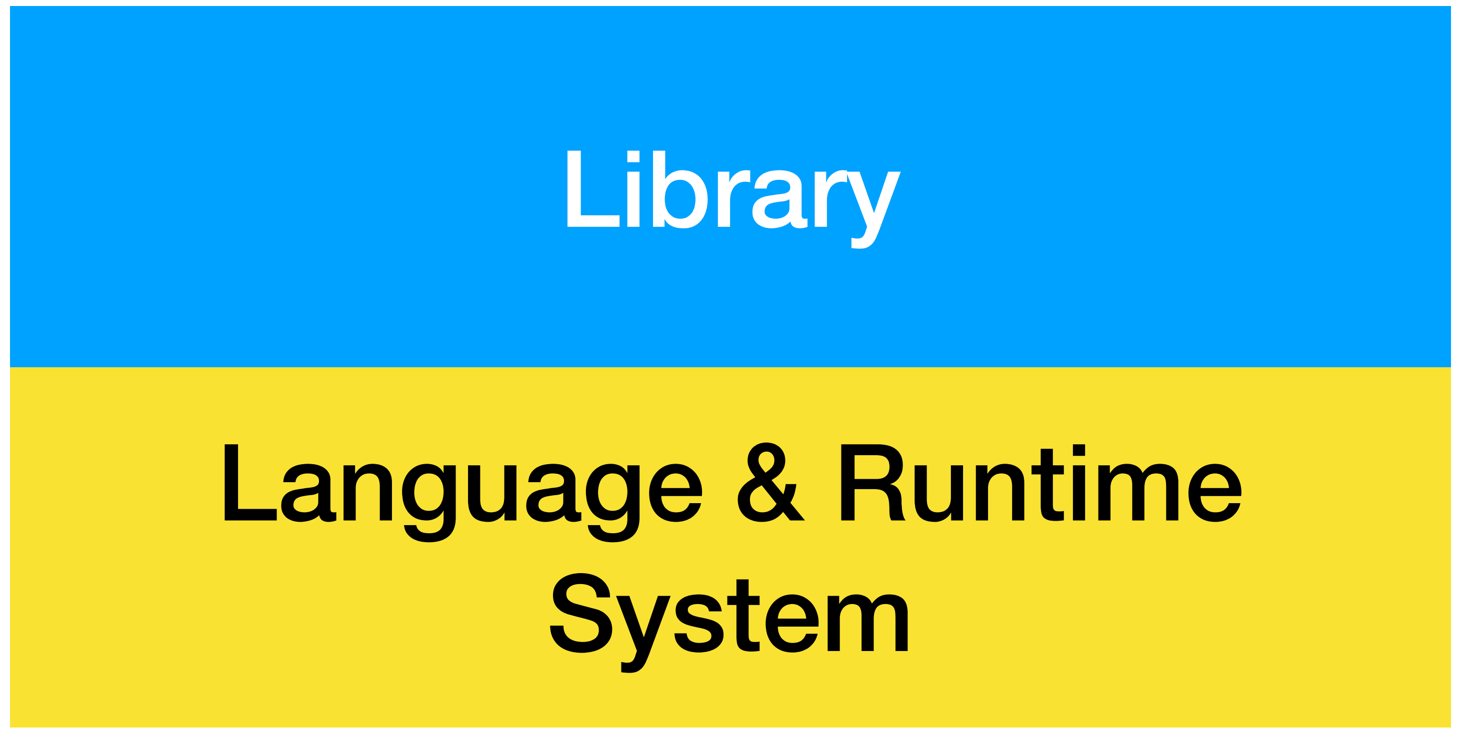
Eliminate function colours with native concurrency support

Concurrency

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads

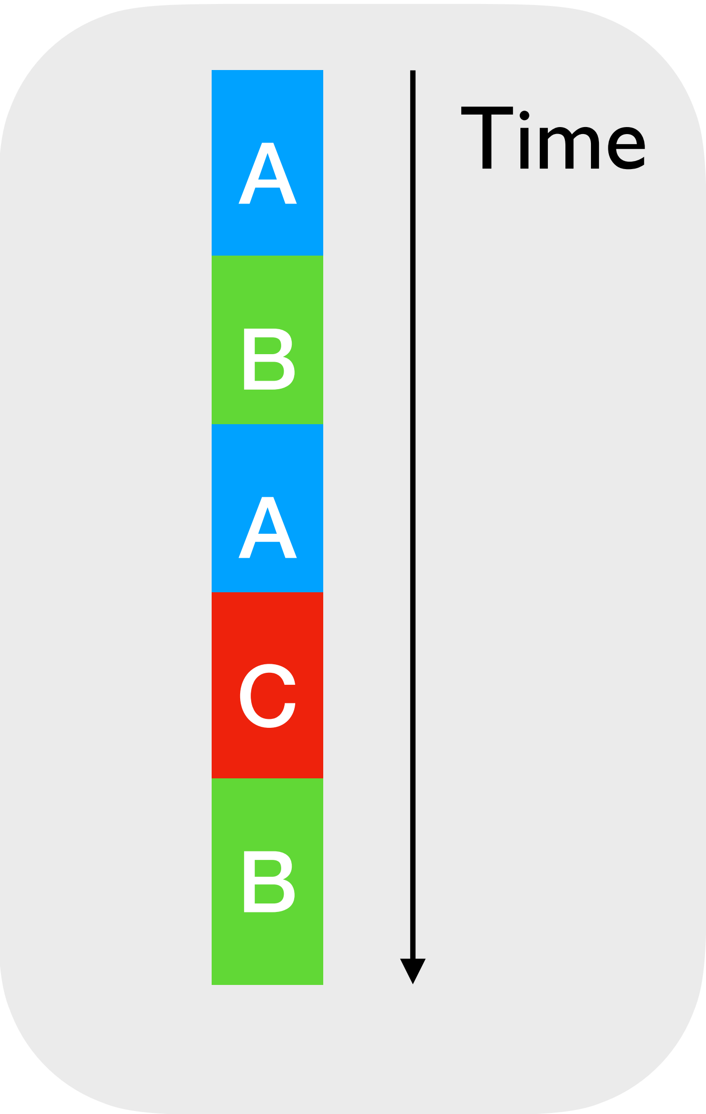


Overlapped execution

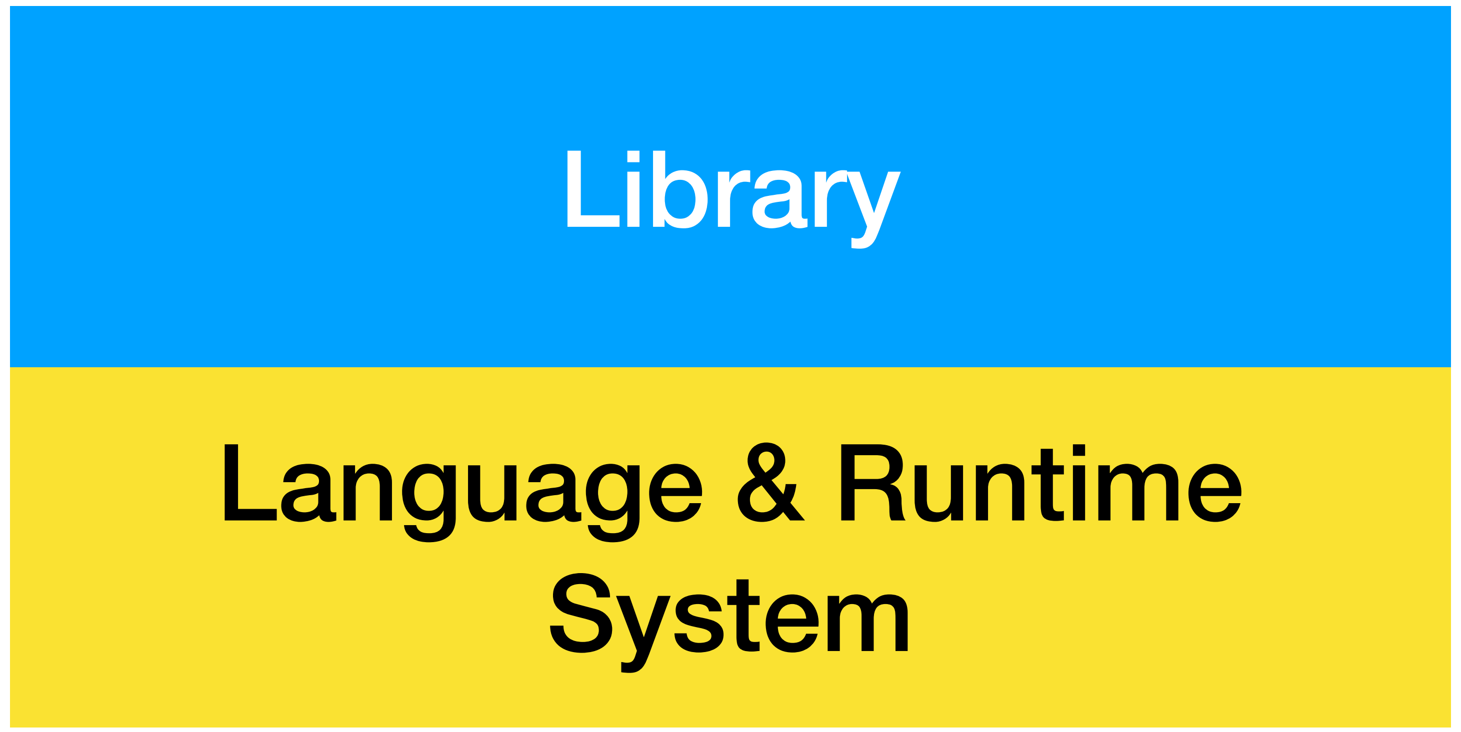


Concurrency

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads

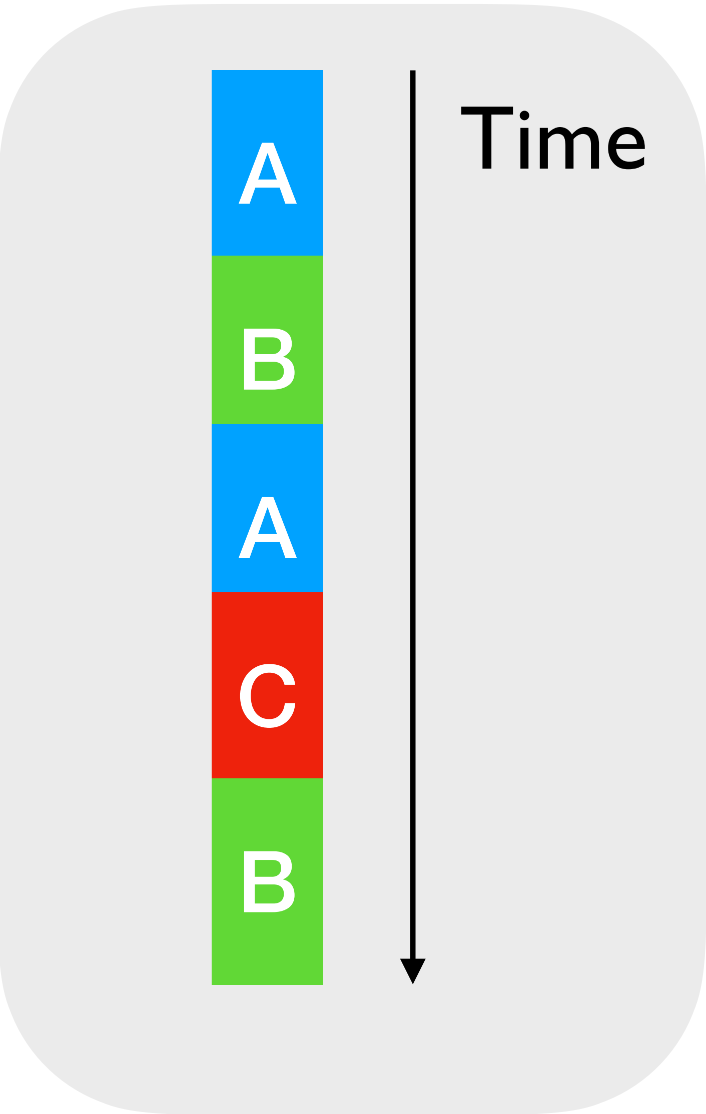


Overlapped execution

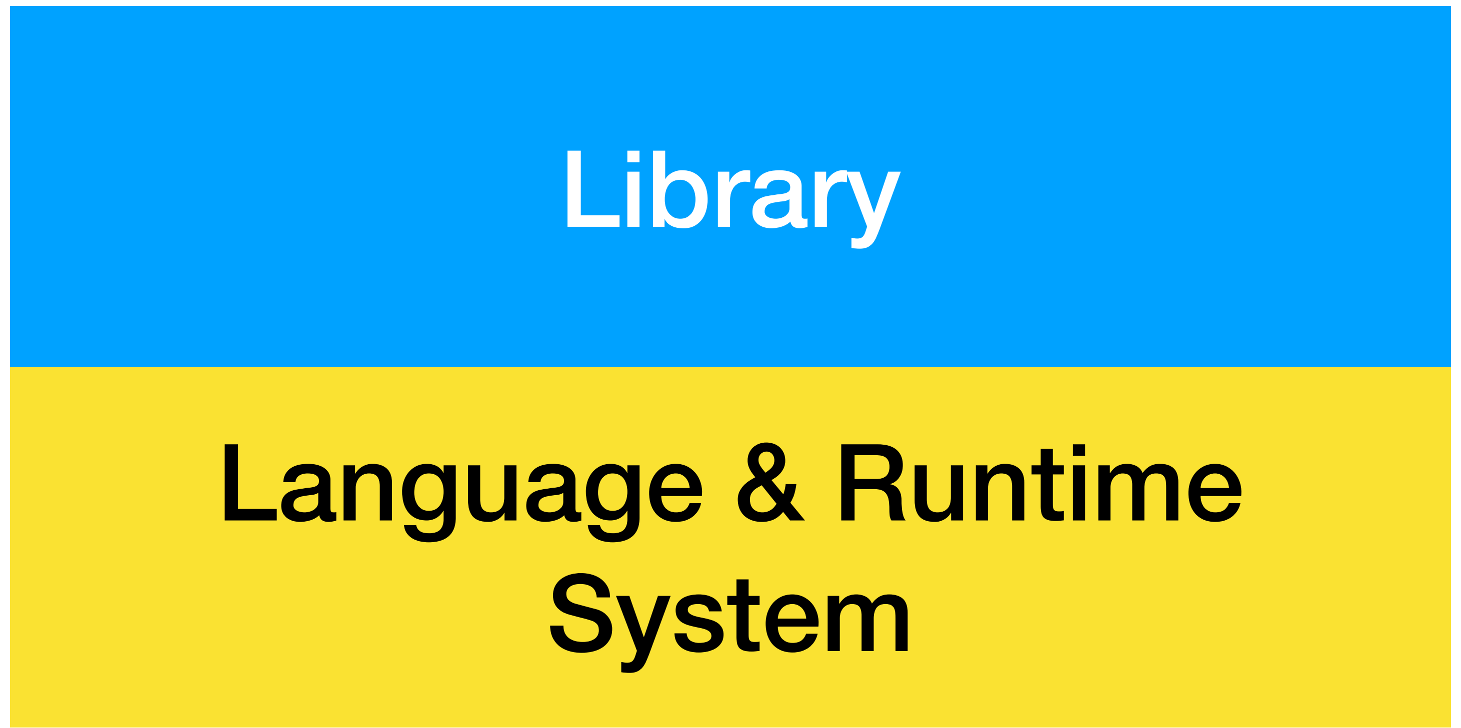


Concurrency

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads



Overlapped execution



Maintenance Burden

C and not Haskell

Lack of flexibility

Concurrency

JFP '14

Composable Scheduler Activations for Haskell

KC SIVARAMAKRISHNAN
University of Cambridge

TIM HARRIS*
Oracle Labs

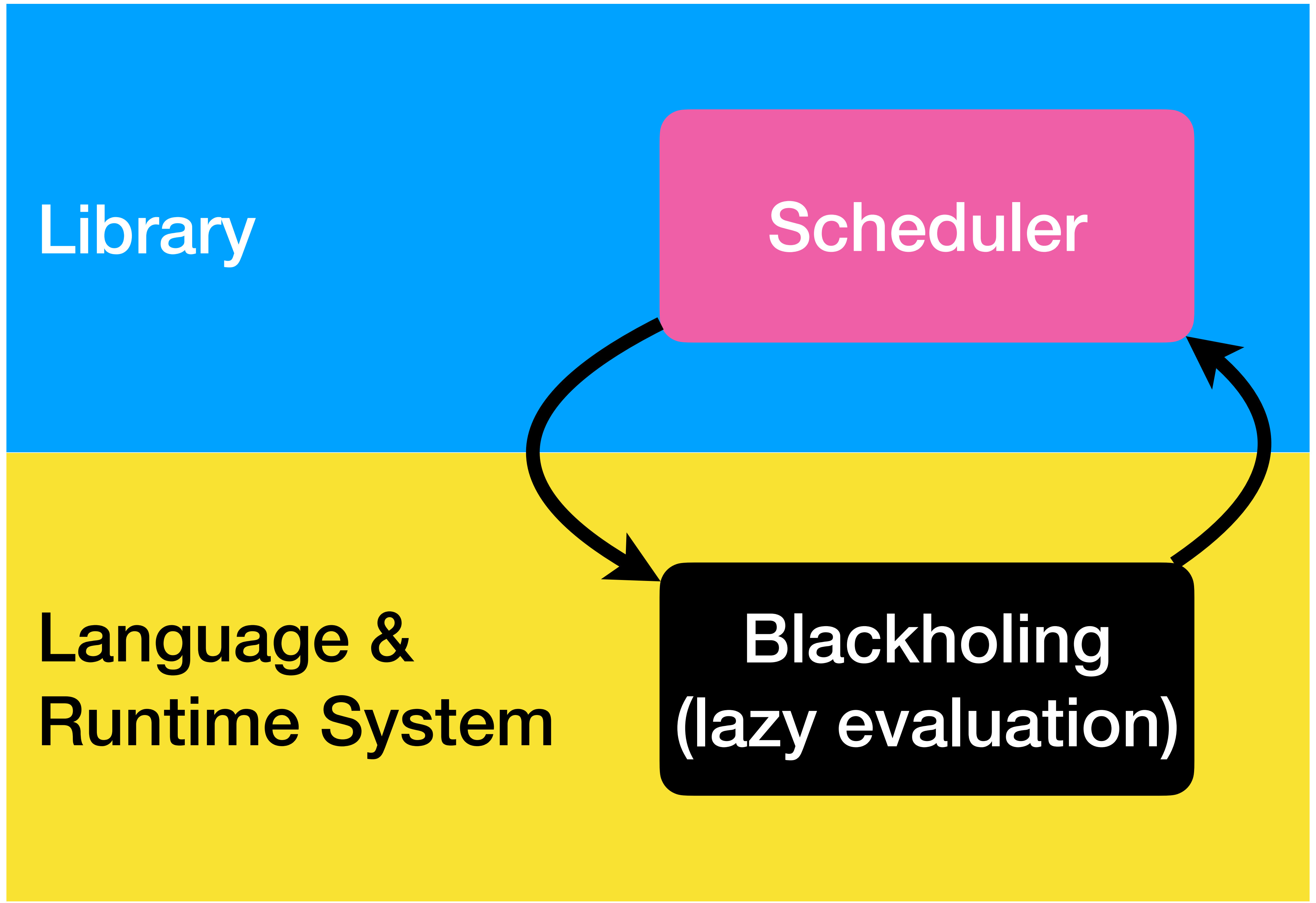
SIMON MARLOW*
Facebook UK Ltd.

SIMON PEYTON JONES
Microsoft Research, Cambridge

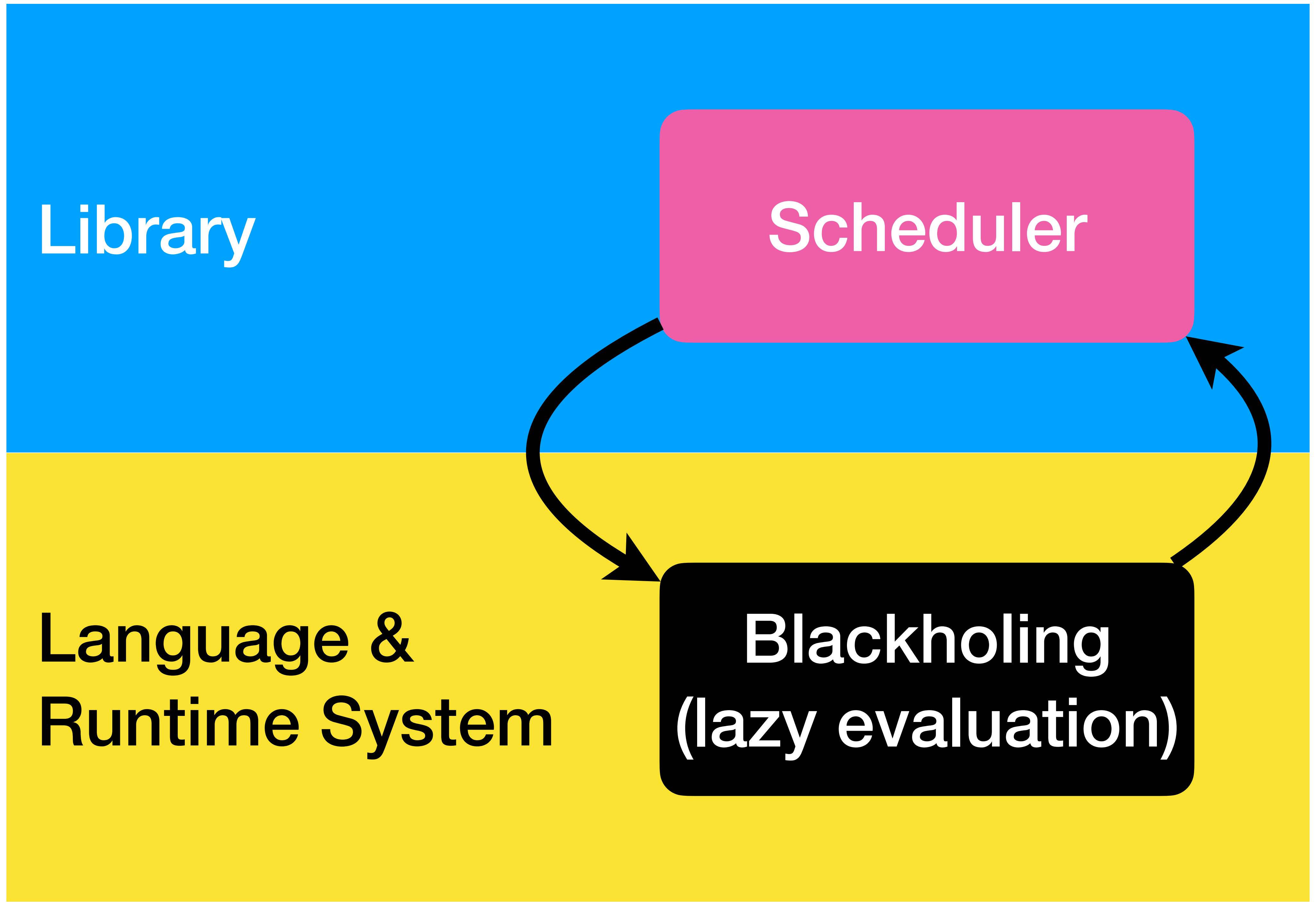
Abstract

The runtime for a modern, concurrent, garbage collected language like Java or Haskell is like an operating system: sophisticated, complex, performant, but alas very hard to change. If more of the runtime system were in the high level language, it would be far more modular and malleable. In

Concurrency



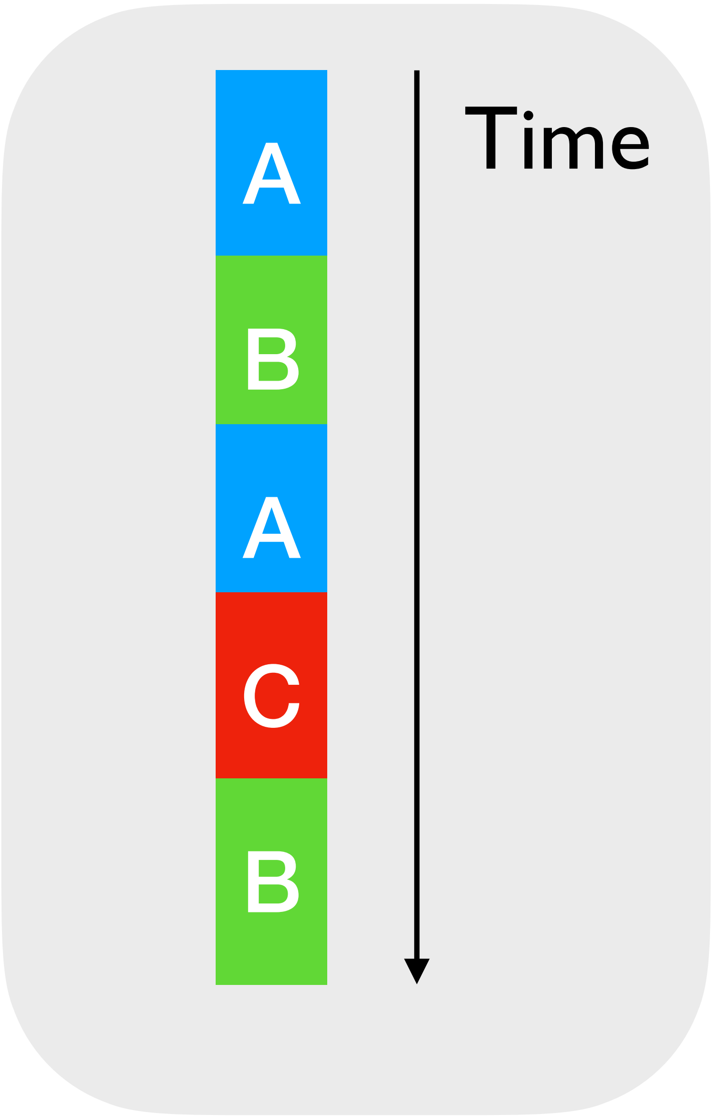
Concurrency



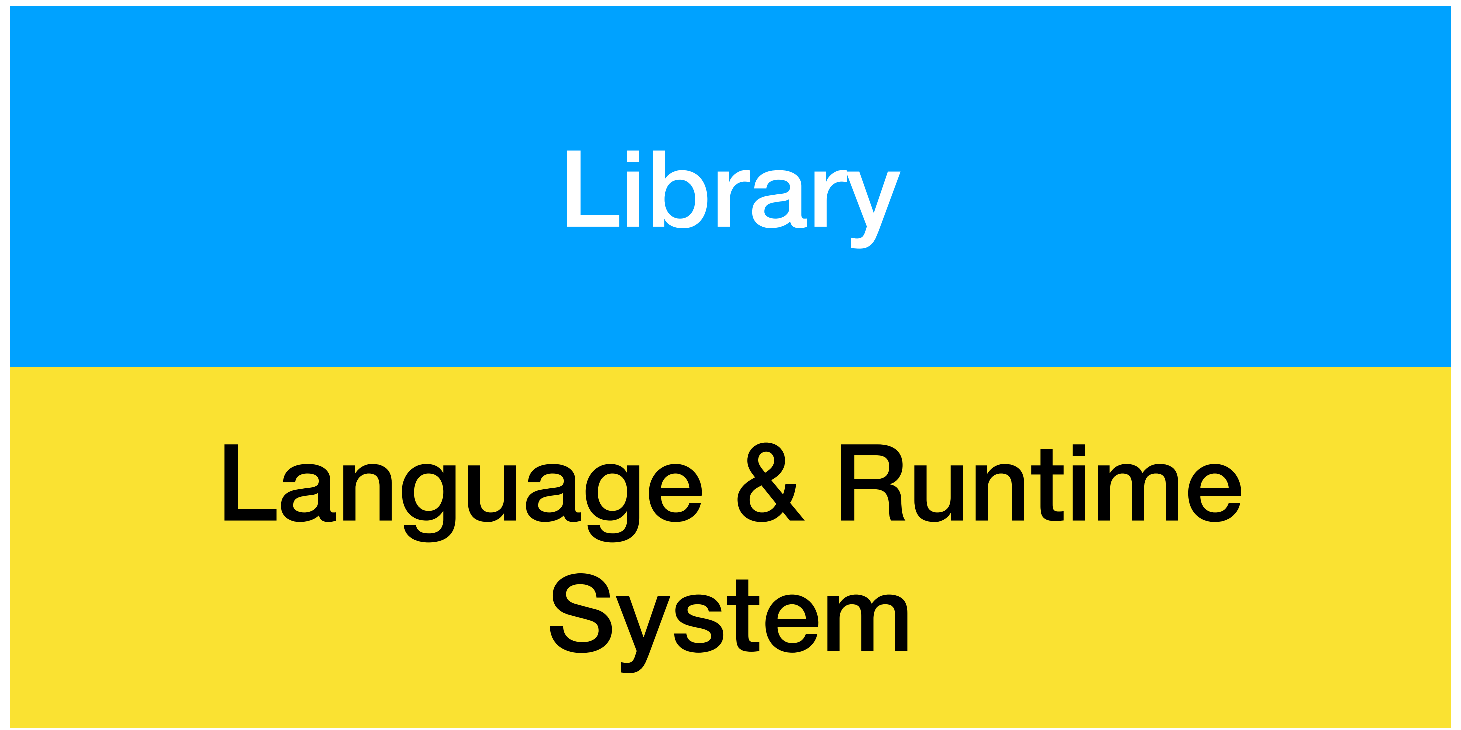
Hard to undo adding a feature into the RTS

Concurrency

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads



Overlapped execution



First-class continuations!

How to continue?

Representing Control in the Presence of One-Shot Continuations*

Carl Bruggeman Oscar Waddell R. Kent Dybvig

Indiana University Computer Science Department
Lindley Hall 215
Bloomington, Indiana 47405
{*bruggema,owaddell,dyb*}*@cs.indiana.edu*

Abstract

Traditional first-class continuation mechanisms allow a captured continuation to be invoked multiple times. Many con-

multi-shot continuations. We present performance measurements that demonstrate that one-shot continuations are indeed more efficient than multi-shot continuations for certain applications, such as thread systems.

PLDI '96

`call/1cc`

Chez Scheme

How to continue?

Representing Control in the Presence of One-Shot Continuations*

PLDI '96

`call/1cc`

Chez Scheme

Composable Asynchronous Events

PLDI '11

Lukasz Ziarek, KC Sivaramakrishnan, Suresh Jagannathan
Purdue University
{lziarek, chandras, suresh}@cs.purdue.edu

Abstract

Although asynchronous communication is an important feature of many concurrent systems, building *composable* abstractions that leverage asynchrony is challenging. This is because an asynchronous operation necessarily involves two distinct threads of control – the thread that initiates the operation, and the thread

kind of strong encapsulation, especially in the presence of communication that spans abstraction boundaries. Consequently, changing the implementation of a concurrency abstraction by adding, modifying, or removing behaviors often requires pervasive change to the users of the abstraction. Modularity is thus compromised.

This is particularly true for asynchronous behavior generated in-

MultiMLton

How to continue?

An argument against call/cc

— Oleg Kiselyov

We argue against `call/cc` as a core language feature, as the distinguished control operation to implement natively relegating all others to libraries. The primitive `call/cc` is a *bad abstraction* -- in various meanings of `bad' shown below, -- and its capture of the continuation of the whole program is not practically useful. The only reward for the hard work to capture the whole continuation efficiently is more hard work to get around the capture of the whole continuation. Both the users and the implementors are better served with a set of well-chosen control primitives of various degrees of generality with well thought-out interactions.

Need delimited continuations

How to continue?

EFF TRY IT OUT LEARN MORE CONTRIBUTE

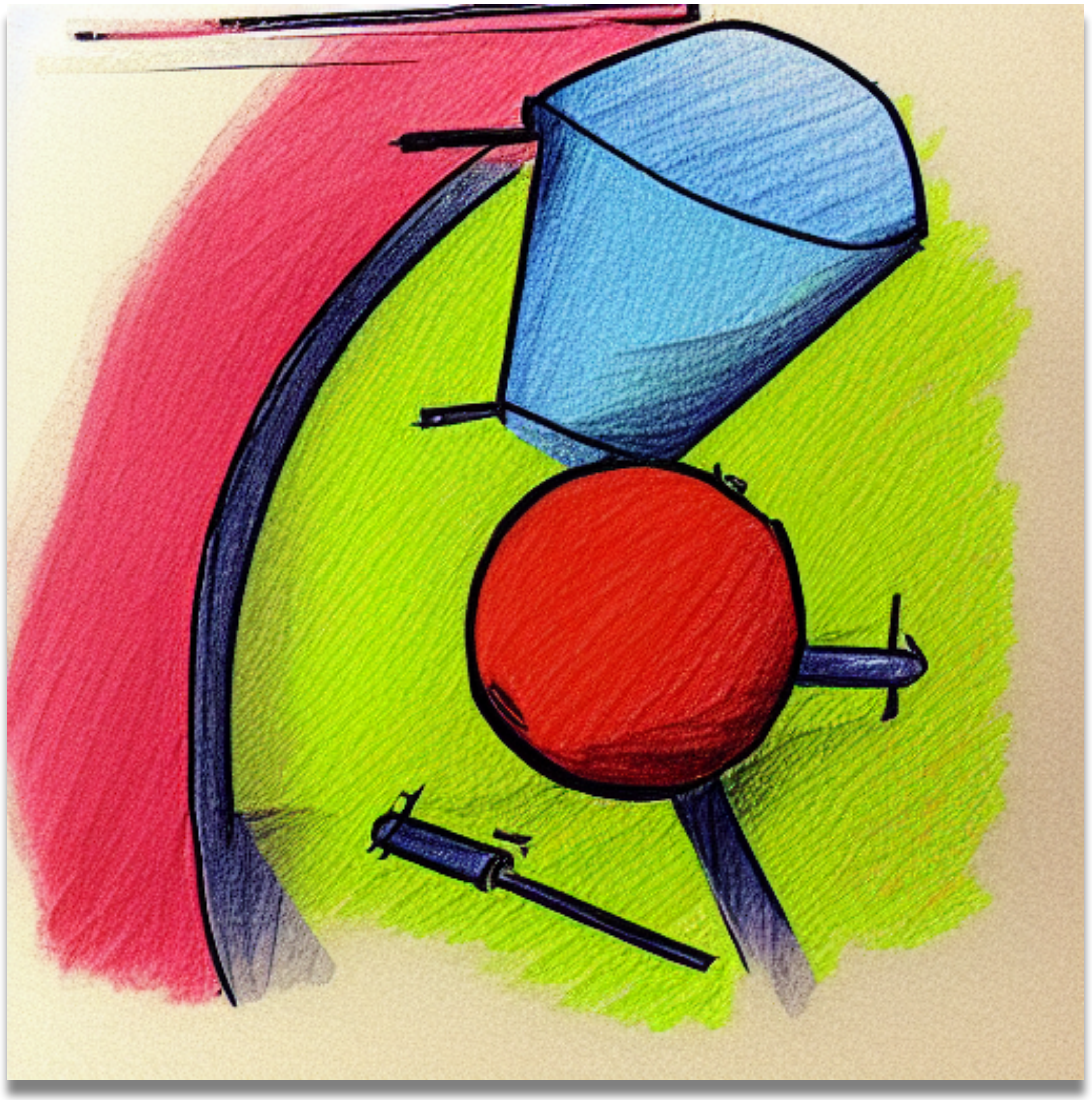
Eff is a functional programming language based on **algebraic effect handlers**. This means that *Eff* provides handlers of not only exceptions, but of any computational effect, allowing you to **redirect output**, wrap state modifications in **transactions**, schedule **asynchronous threads**, and much much more…

TRY IT OUT IN YOUR BROWSER!

or install it on your computer

Structured delimited continuations

Ease of comprehension



```
exception E

let comp () =
  print_string (raise E)

let main () =
  try comp ()
  with E ->
    print_string “Raised”
```

Exception

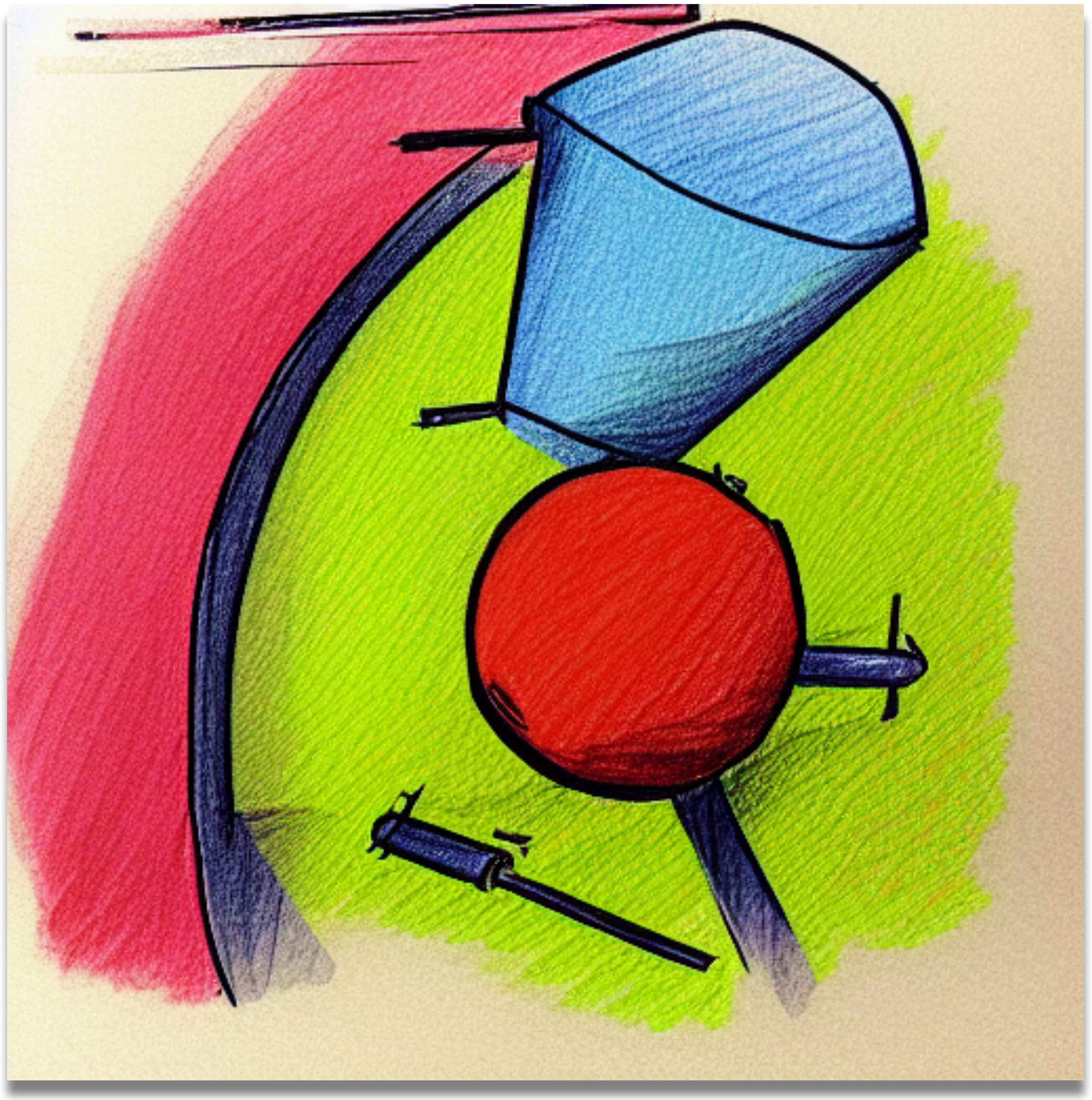
```
effect E : string

let comp () =
  print_string (perform E)

let main () =
  try comp ()
  with effect E k ->
    continue k “Handled"
```

delimited continuation

Effect handler

- Effect handler ~= Resumable exceptions + *computation* may be resumed later

Ease of comprehension



```
exception E

let comp () =
  print_string (raise E)

let main () =
  try comp ()
  with E ->
    print_string “Raised”
```

Exception

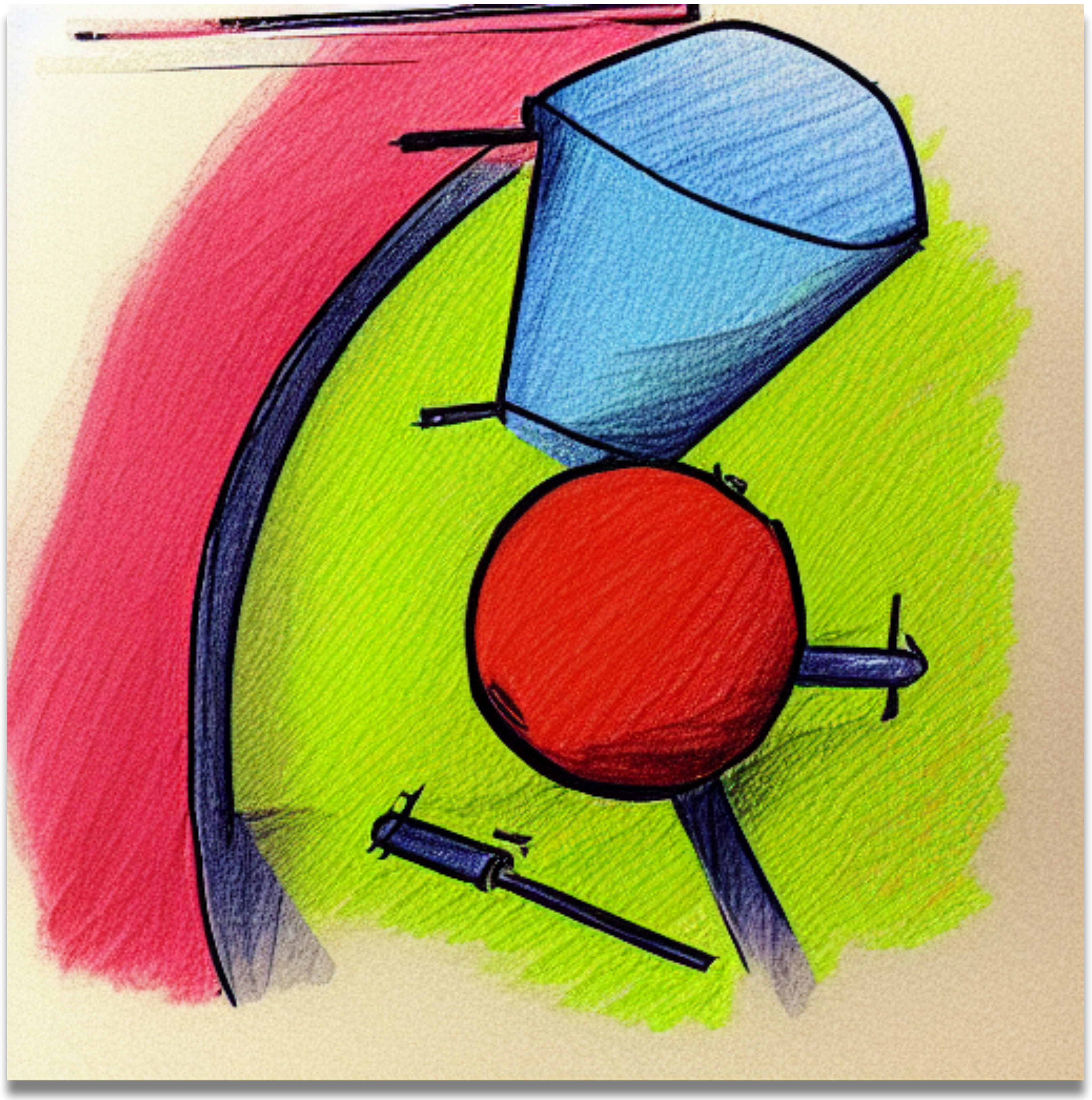
```
effect E : string

let comp () =
  print_string (perform E)

let main () =
  try comp ()
  with effect E k ->
    continue k “Handled"
```

delimited continuation

Effect handler

- Effect handler ~= Resumable exceptions + *computation* may be resumed later
- *Easier* than `shift/reset, control/prompt`
  - ✦ No *prompts* or *answer-type polymorphism*

Ease of comprehension



```
exception E

let comp () =
  print_string (raise E)

let main () =
  try comp ()
  with E ->
    print_string "Raised"
```

Exception

```
effect E : string

let comp () =
  print_string (perform E)

let main () =
  try comp ()
  with effect E k ->
    continue k "Handled"
```

delimited continuation

Effect handler

- Effect handler ~= Resumable exceptions + *computation* may be resumed later
- *Easier* than `shift/reset, control/prompt`
  - ✦ No *prompts* or *answer-type polymorphism*

Effect handlers : *shift/reset* :: *while* : *goto*

How to continue?

Effective Concurrency through Algebraic Effects

Stephen Dolan[1], Leo White[2], KC Sivaramakrishnan[1], Jeremy Yallop[1], and Anil Madhavapeddy[1]

[1]University of Cambridge
[2]Jane Street Capital

OCaml '15

Algebraic effects and handlers provide a modular abstraction for expressing effectful computation, allowing the programmer to separate the expression of an effectful computation from its implementation. We present an extension to OCaml for programming with linear algebraic effects, and demonstrate its use in expressing concurrency

The basic tenet of programming with algebraic effects is that performing an effectful computation is separate from its interpretation [1, 5]. In particular, the interpretation is dynamically chosen based on the context in which an effect is performed. In our example, spawning a new thread and yielding control to another are effectful ac-

One-shot delimited continuations exposed through effect handlers

Ease of comprehension ~> Impact

facebook

React

Ease of comprehension ~> Impact

facebook

React

React Docs Tutorial Blog Community

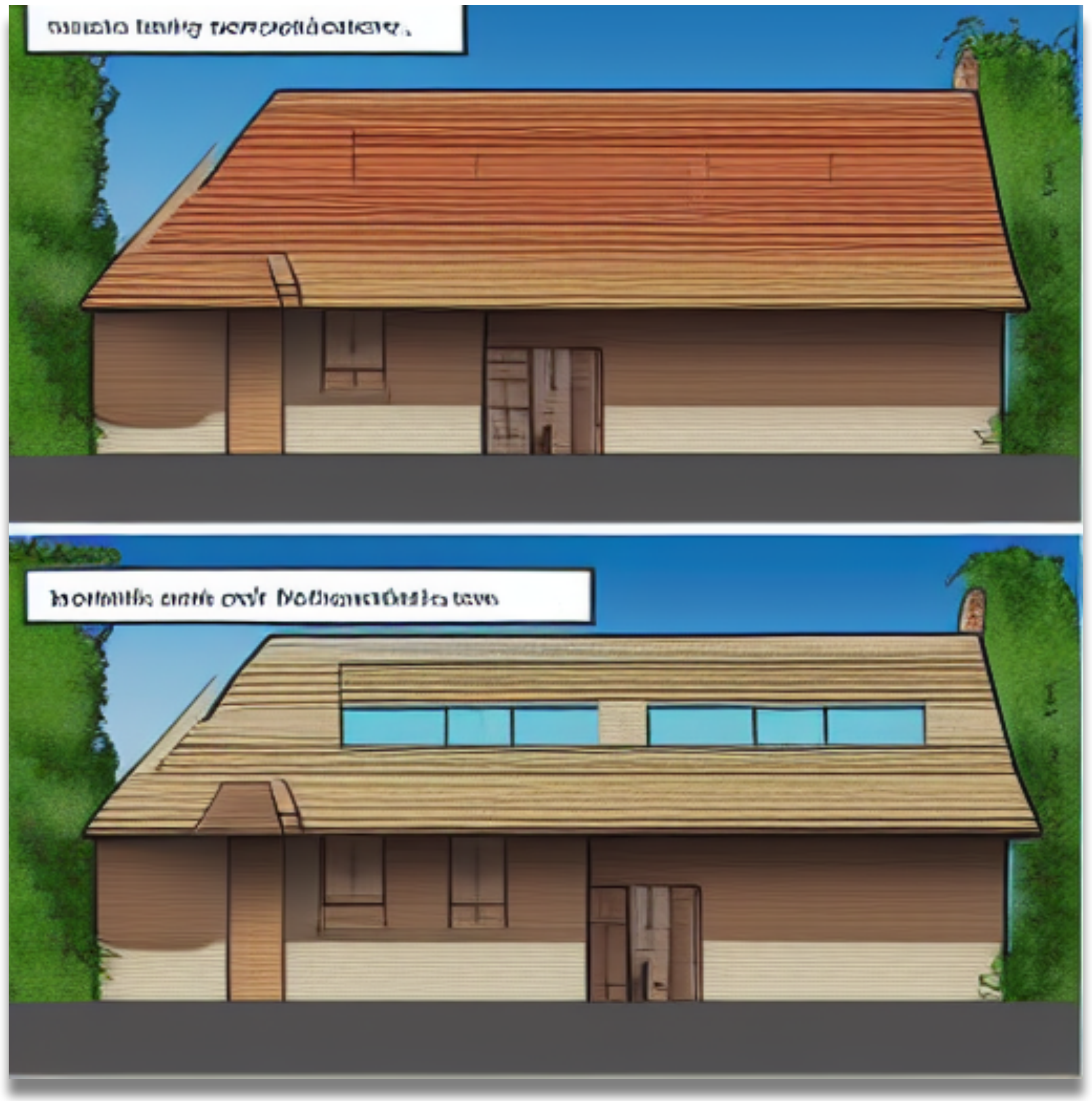
What is the prior art for Hooks?

Hooks synthesize ideas from several different sources:

- Our old experiments with functional APIs in the react-future repository.
- React community's experiments with render prop APIs, including Ryan Florence's Reactions Co
- Dominic Gannaway's `adopt` keyword proposal as a sugar syntax for render props.
- State variables and state cells in DisplayScript.
- Reducer components in ReasonReact.
- Subscriptions in Rx.
- Algebraic effects in Multicore OCaml.

Sebastian Markbåge came up with the original design for Hooks, later refined by Andrew Clark, Sophie Alpert, Dominic Gannaway, and other members of the React team.

Retrofitting Effect Handlers



Koka
A Functional Language with Effect Types and Handlers

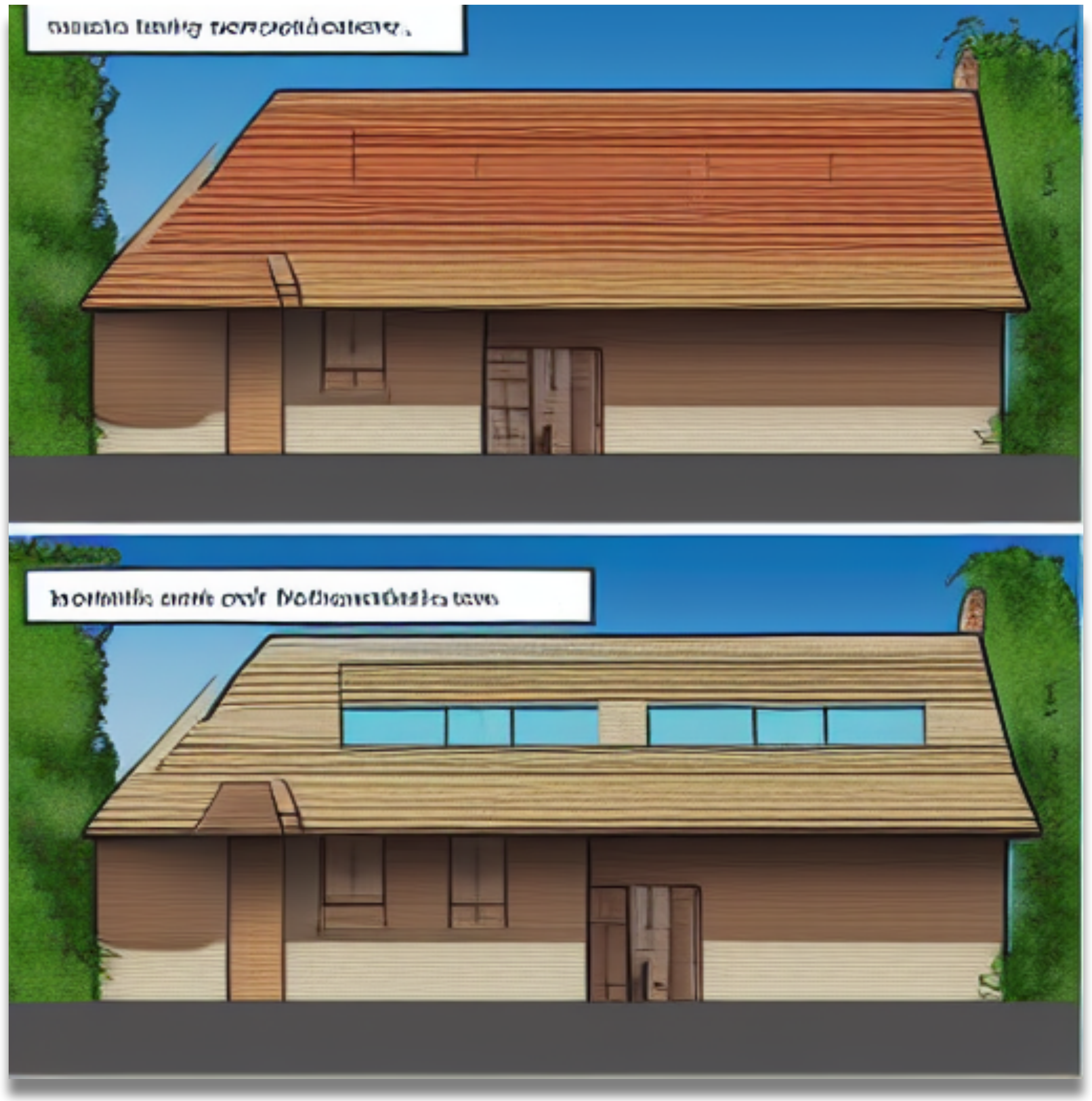
Effekt Language
A research language with effect handlers and lightweight effect polymorphism

EFF   TRY IT OUT   LEARN MORE   CONTRIBUTE

Eff is a functional programming language based on **algebraic effect handlers**. This means that *Eff* provides handlers of not only exceptions, but of any

- Don’t break existing code ⇒ *No effect system*
  - ✦ No syntax and just functions

Retrofitting Effect Handlers



Koka
A Functional Language with Effect Types and Handlers

Effekt Language
A research language with effect handlers and lightweight effect polymorphism

EFF | TRY IT OUT | LEARN MORE | CONTRIBUTE

Eff is a functional programming language based on **algebraic effect handlers**. This means that *Eff* provides handlers of not only exceptions, but of any

- Don't break existing code ⇒ *No effect system*
  - ✦ No syntax and just functions
- Focus on preserving
  - ✦ *Performance* of legacy code (< 1% impact)
  - ✦ *Compatibility* of tools — gdb, perf

Retrofitting Effect Handlers

Retrofitting Effect Handlers onto OCaml

PLDI '21

KC Sivaramakrishnan
IIT Madras
Chennai, India
kcsrk@cse.iitm.ac.in

Stephen Dolan
OCaml Labs
Cambridge, UK
stephen.dolan@cl.cam.ac.uk

Leo White
Jane Street
London, UK
leo@lpw25.net

Tom Kelly
OCaml Labs
Cambridge, UK
tom.kelly@cantab.net

Sadiq Jaffer
Opsian and OCaml Labs
Cambridge, UK
sadiq@toao.com

Anil Madhavapeddy
University of Cambridge and OCaml Labs
Cambridge, UK
avsm2@cl.cam.ac.uk

Abstract

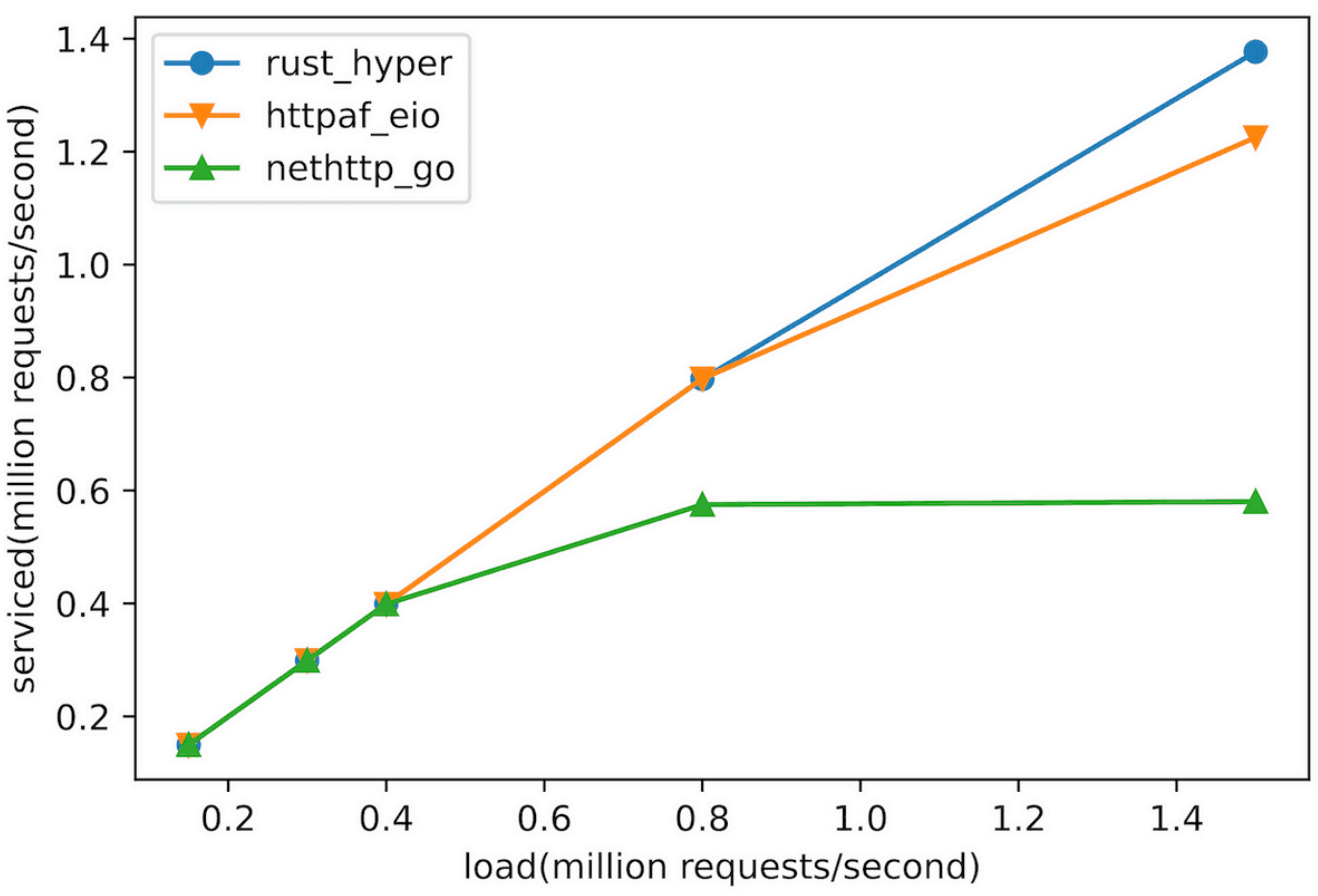
Effect handlers have been gathering momentum as a mechanism for modular programming with user-defined effects.

1 Introduction

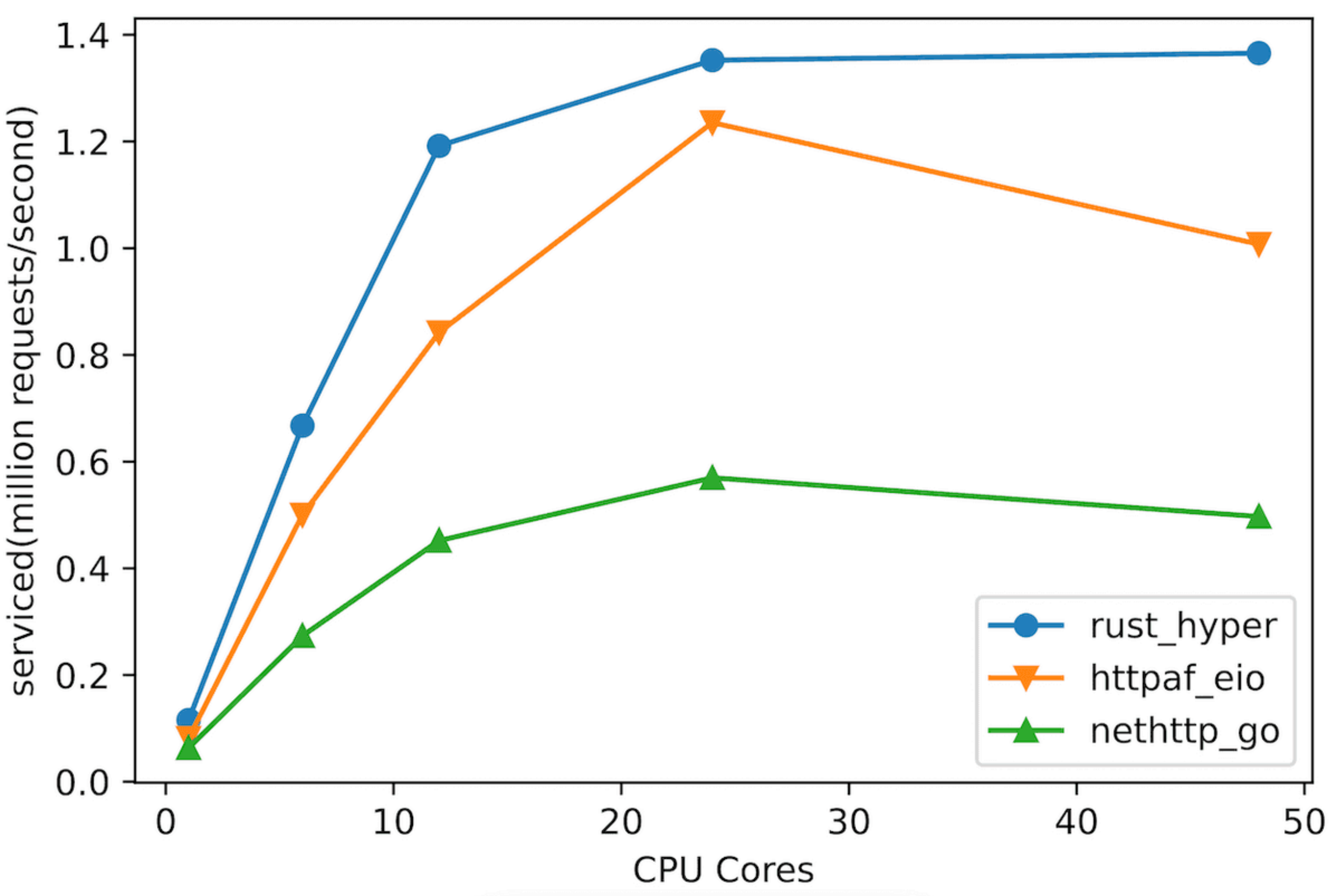
Effect handlers [45] provide a modular foundation for user-defined effects. The key idea is to separate the definition of

✦ *Compatibility* of tools — gdb, perf

Eio — Direct-style effect-based concurrency

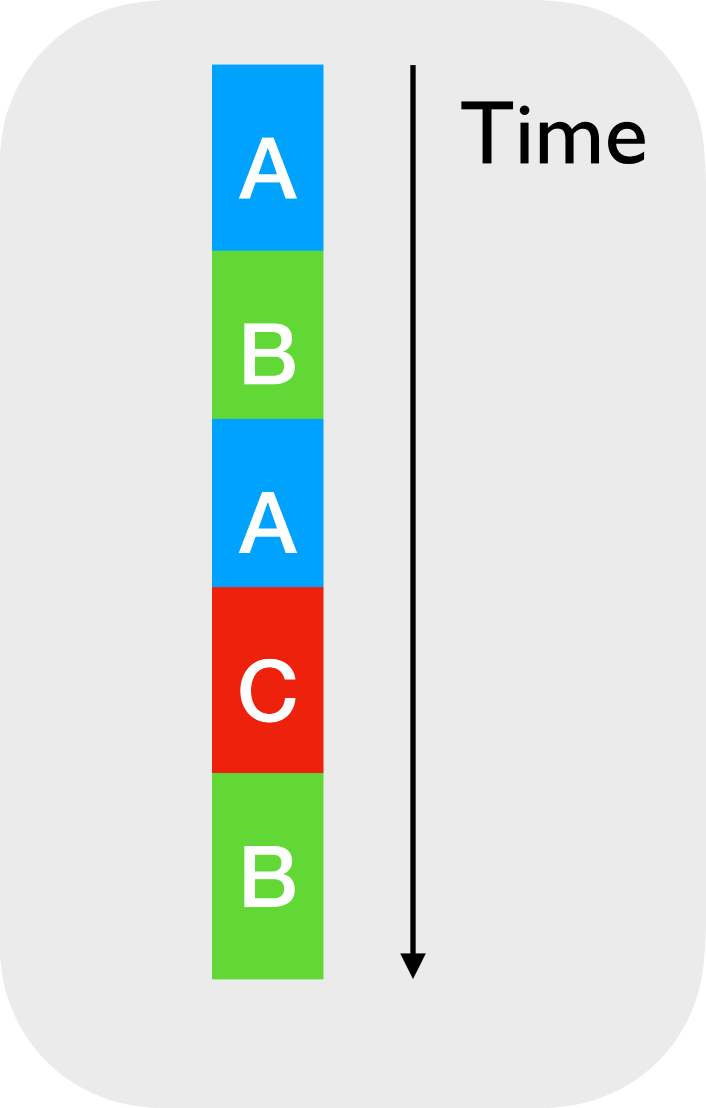


HTTP server performance using 24 cores



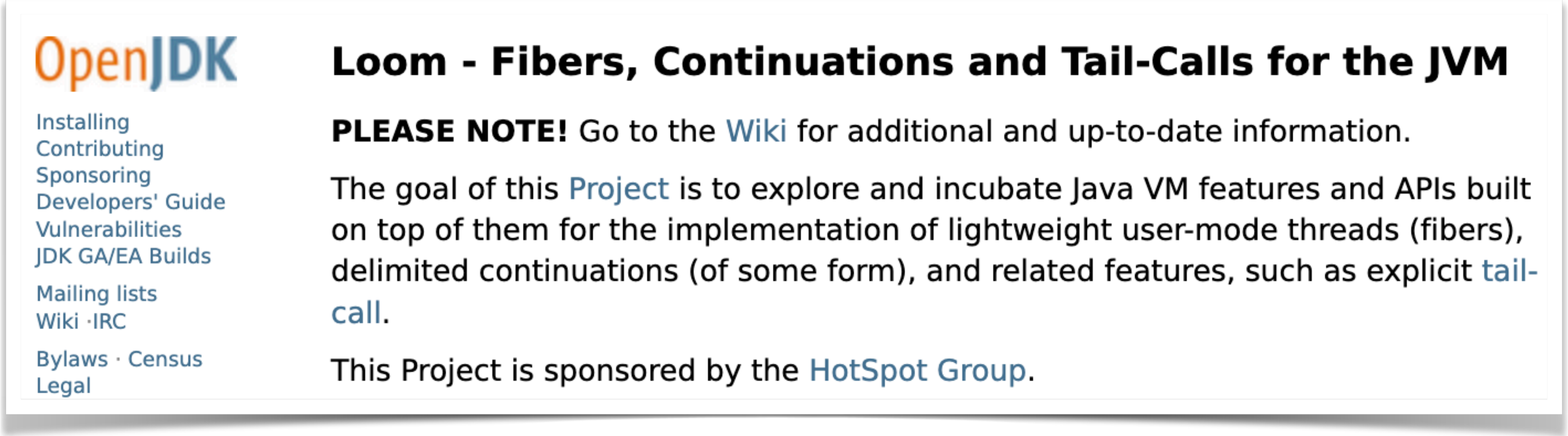
HTTP server scaling maintaining a constant load of 1.5 million requests per second

Concurrency (~2022)



Overlapped execution

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads



OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds

Mailing lists
Wiki ·IRC

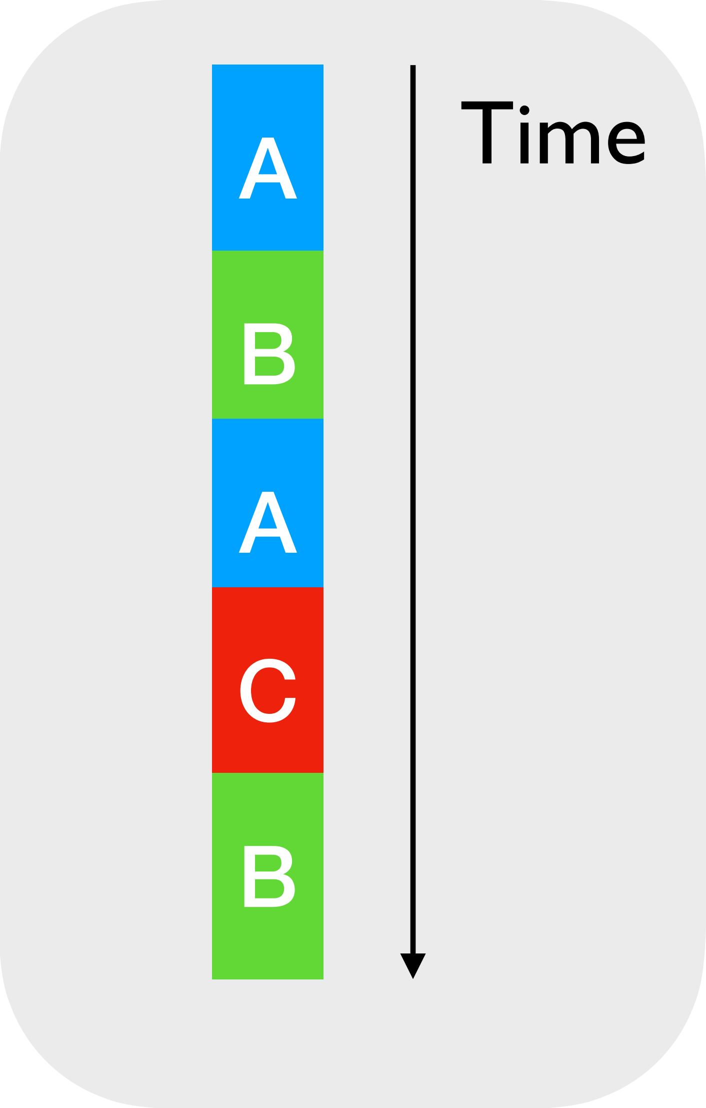
Bylaws · Census
Legal

Loom - Fibers, Continuations and Tail-Calls for the JVM

PLEASE NOTE! Go to the Wiki for additional and up-to-date information.

The goal of this Project is to explore and incubate Java VM features and APIs built on top of them for the implementation of lightweight user-mode threads (fibers), delimited continuations (of some form), and related features, such as explicit tail-call.

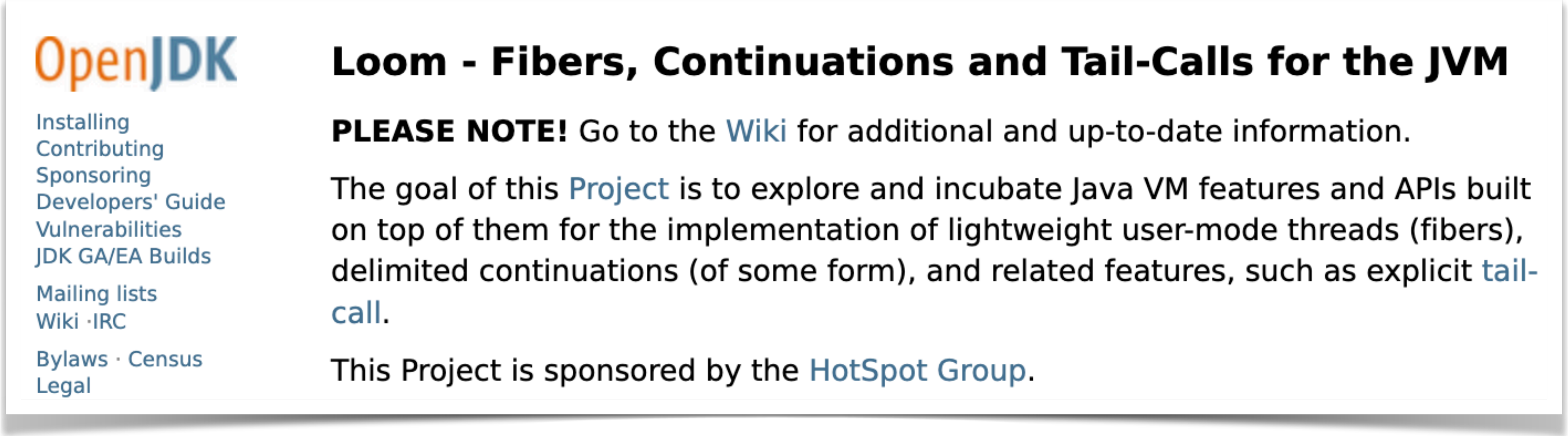
This Project is sponsored by the HotSpot Group.

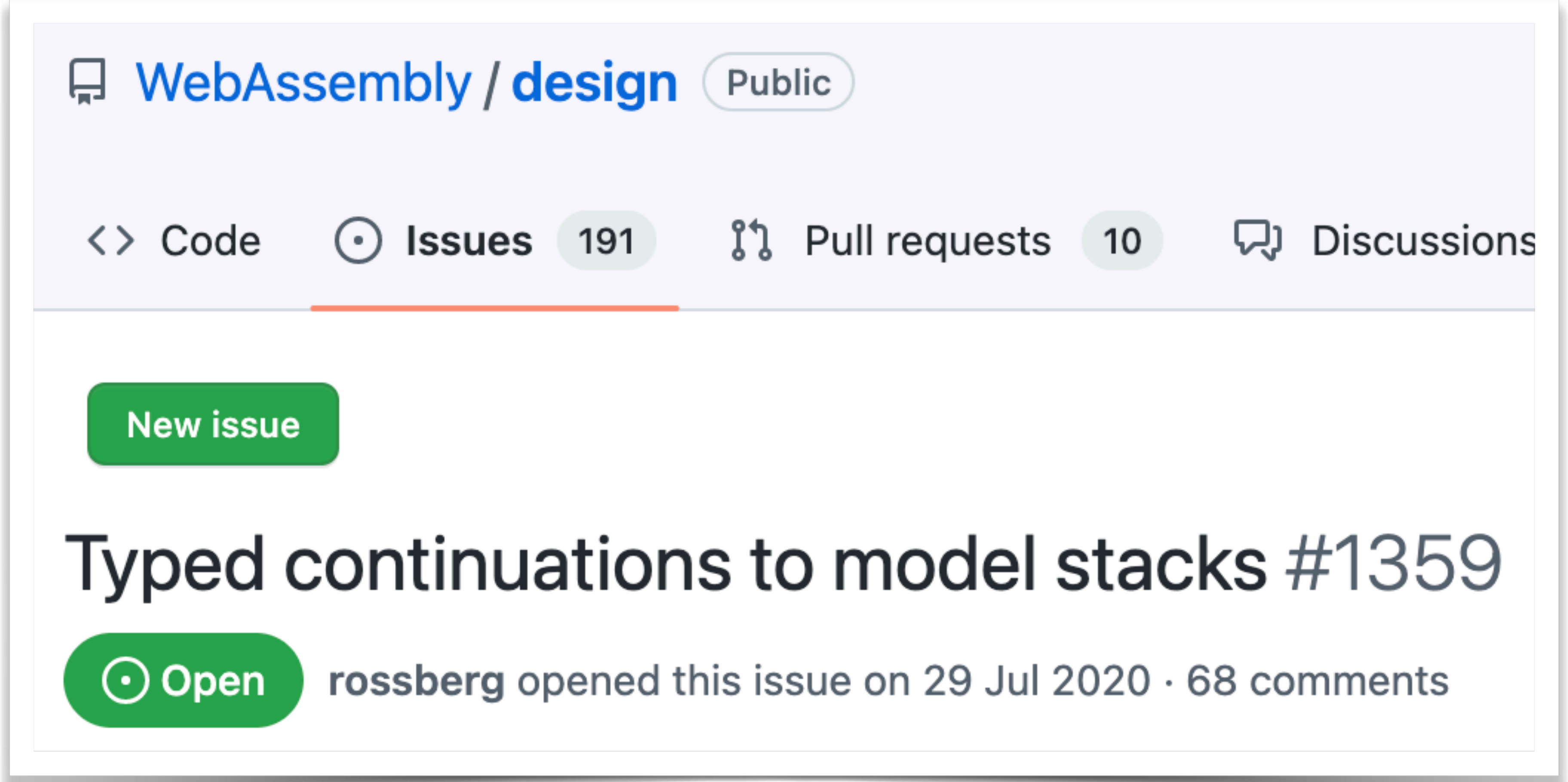


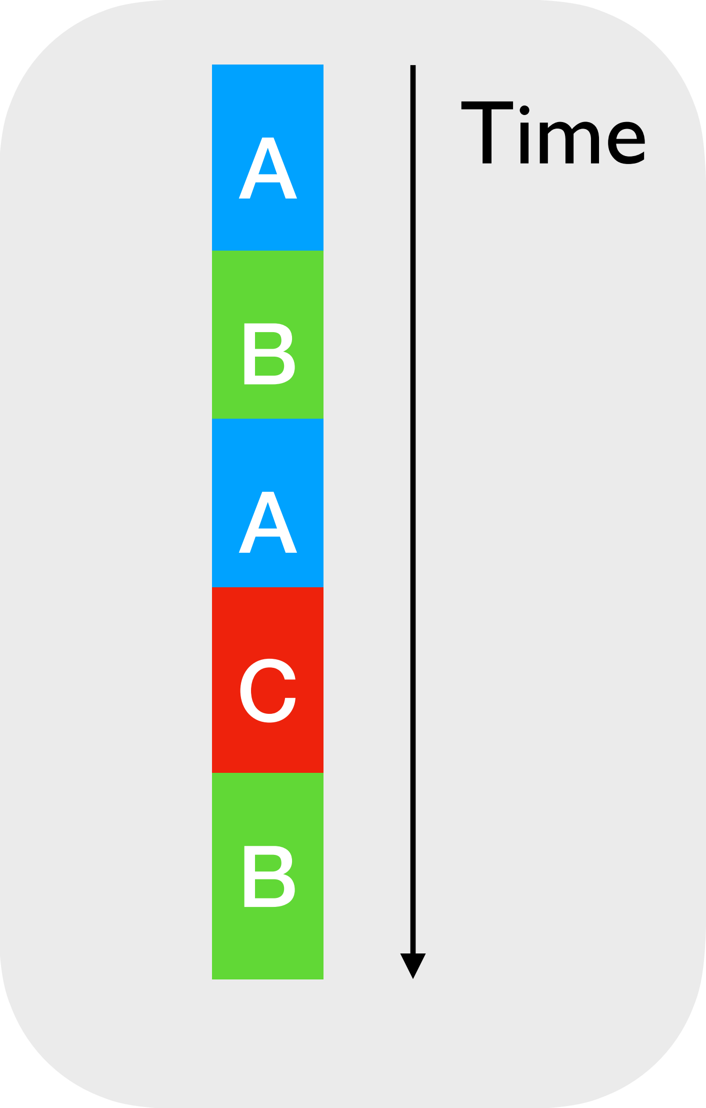
Overlapped execution

Concurrency (~2022)

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads







Overlapped execution

Concurrency (~2022)

- *Parallelism* is a resource; *concurrency* is a programming abstraction
  - ✦ Language-level threads

OpenJDK

Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki ·IRC
Bylaws · Census
Legal

Loom - Fibers, Continuations and Tail-Calls for the JVM

PLEASE NOTE! Go to the Wiki for additional and up-to-date information.

The goal of this Project is to explore and incubate Java VM features and APIs built on top of them for the implementation of lightweight user-mode threads (fibers), delimited continuations (of some form), and related features, such as explicit tail-call.

This Project is sponsored by the HotSpot Group.

WebAssembly / **design** Public

Code · Issues 191 · Pull requests 10 · Discussions

New issue

Typed continuations to model stacks #1359

Open **rossberg** opened this issue on 29 Jul 2020 · 68 comments

Native, first-class, delimited continuations

Merged **Alexis King** requested to merge `lexi.lambda/ghc:first-clas…` into `master` 5 months ago

~2 days ago

4 unresolved threads

Overview 64 · Commits 1 · Pipelines 24 · Changes 56

This MR implements GHC proposal 313: Delimited continuation primops by adding native support for delimited continuations to the GHC RTS.

Takeaways

Care for Users

- Transition to the new version should be a *no-op* or *push-button* solution
  - *Most code likely to remain sequential*

Care for Users

- Transition to the new version should be a *no-op* or *push-button* solution
  - *Most code likely to remain sequential*
- Build tools to ease the transition

	4.14	5.0+alpha-repo	number of revdeps
0install.2.18	☑	☒	1
BetterErrors.0.0.1	☑	☒	7
TCSLib.0.3	☑	☒	1
absolute.0.1	☑	☒	0
acgtk.1.5.3	☑	☒	0
advi.2.0.0	☑	☒	0
aez.0.3	☑	☒	0
ahrocksdb.0.2.2	☒	☒	0
aio.0.0.3	☑	☒	0
alt-ergo-free.2.2.0	☑	☒	7
amqp-client-async.2.2.2	☑	☒	0
amqp-client-lwt.2.2.2	☑	☒	0
ancient.0.9.1	☑	☒	0
apron.v0.9.13	☑	☒	17

OPAM Health Check: http://check.ocamllabs.io/

Benchmarking

Rigorously, *Continuously* on *Real* programs

- OCaml users don’t just run synthetic benchmarks

Benchmarking

Rigorously, *Continuously* on *Real* programs

- OCaml users don't just run synthetic benchmarks
- **Sandmark** — Real-world programs picked from wild
  - ✦ Coq
  - ✦ Menhir (parser-generator)
  - ✦ Alt-ergo (solver)
  - ✦ Irmin (database)

… and their large set of OPAM dependencies

Benchmarking

Rigorously, *Continuously* on *Real* programs

Program **P**: OCaml 4.14 = *19s* OCaml 5.0 = *18s*

Are the speedups / slowdowns statistically significant?

Benchmarking

Rigorously, *Continuously* on *Real* programs

Program **P**: OCaml 4.14 = 19s OCaml 5.0 = 18s

Are the speedups / slowdowns statistically significant?

- Modern OS, arch, micro-arch effects become significant at small scales
  - *20% speedup by inserting fences*

Benchmarking

Rigorously, *Continuously* on *Real* programs

Program **P**: OCaml 4.14 = *19s* OCaml 5.0 = *18s*

Are the speedups / slowdowns statistically significant?

- Modern OS, arch, micro-arch effects become significant at small scales
  - *20% speedup by inserting fences*
- *Tune the machine to remove noise*

Benchmarking

Rigorously, *Continuously* on *Real* programs

Program **P**: OCaml 4.14 = *19s* OCaml 5.0 = *18s*

Are the speedups / slowdowns statistically significant?

- Modern OS, arch, micro-arch effects become significant at small scales
  - *20% speedup by inserting fences*
- *Tune the machine to remove noise*
- Useful to measure *instructions retired* along with *real time*

Benchmarking

ICFP 2019 (series) / OCaml 2019 (series) / OCaml 2019 /

Benchmarking the OCaml compiler: our experience

OCaml '19

Track

OCaml 2019

When

Fri 23 Aug 2019 14:20 - 14:45 at Pine - Tools Chair(s): Thomas Gazagnaire

Abstract

Our proposed presentation would describe what we did to build continuous benchmarking websites that take a controlled experiment approach to running the operf-micro and sandmark benchmarking suites against tracked git branches of the OCaml compiler. Our goal was to produce tools to allow for the efficient upstreaming of multicore related functionality into the OCaml compiler.

The presentation will cover the available OCaml benchmarking suites; how other compiler communities handle performance benchmark; how we put together our experimental setup to collect controlled data; the interesting experience that should be shared from the project; interesting future work and extensions that could be pursued in the future.

programs

significant?

portant at small

ong with *real time*

Tuning the machine for benchmarking

Benchmarking

STABILIZER: Statistically Sound Performance Evaluation

Charlie Curtsinger Emery D. Berger
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
{charlie,emery}@cs.umass.edu

ASPLOS '13

Abstract

Researchers and software developers require effective performance

1. Introduction

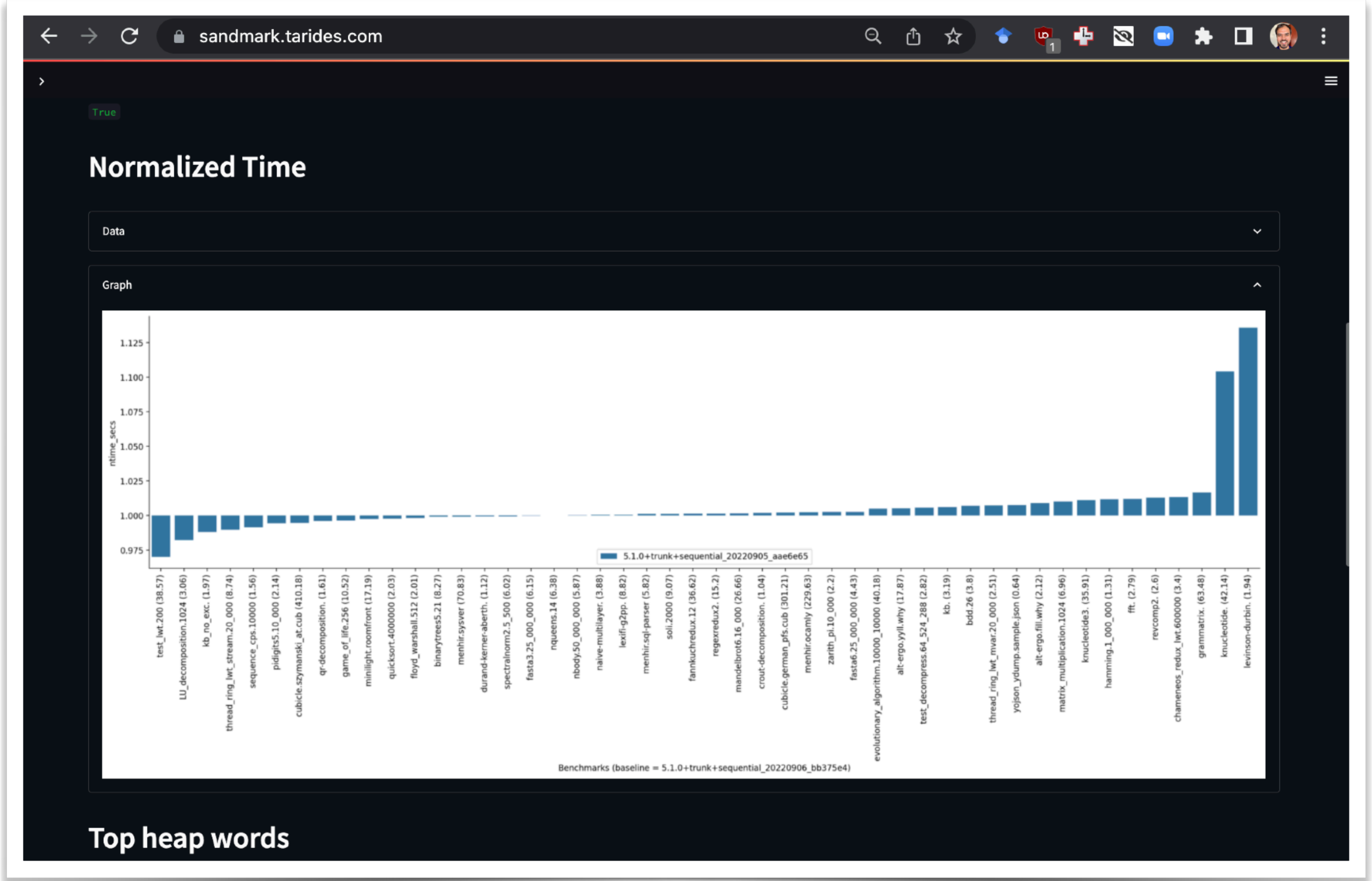
The task of performance evaluation forms a key part of both sys-

Tuning the machine for benchmarking

Benchmarking

Rigorously, *Continuously* on *Real* programs

- Continuous benchmarking as a service
  - sandmark.tarides.com



Invest in tooling

Reuse existing tools; if not build them!

Invest in tooling

Reuse existing tools; if not build them!

- rr = gdb + *record-and-replay debugging*

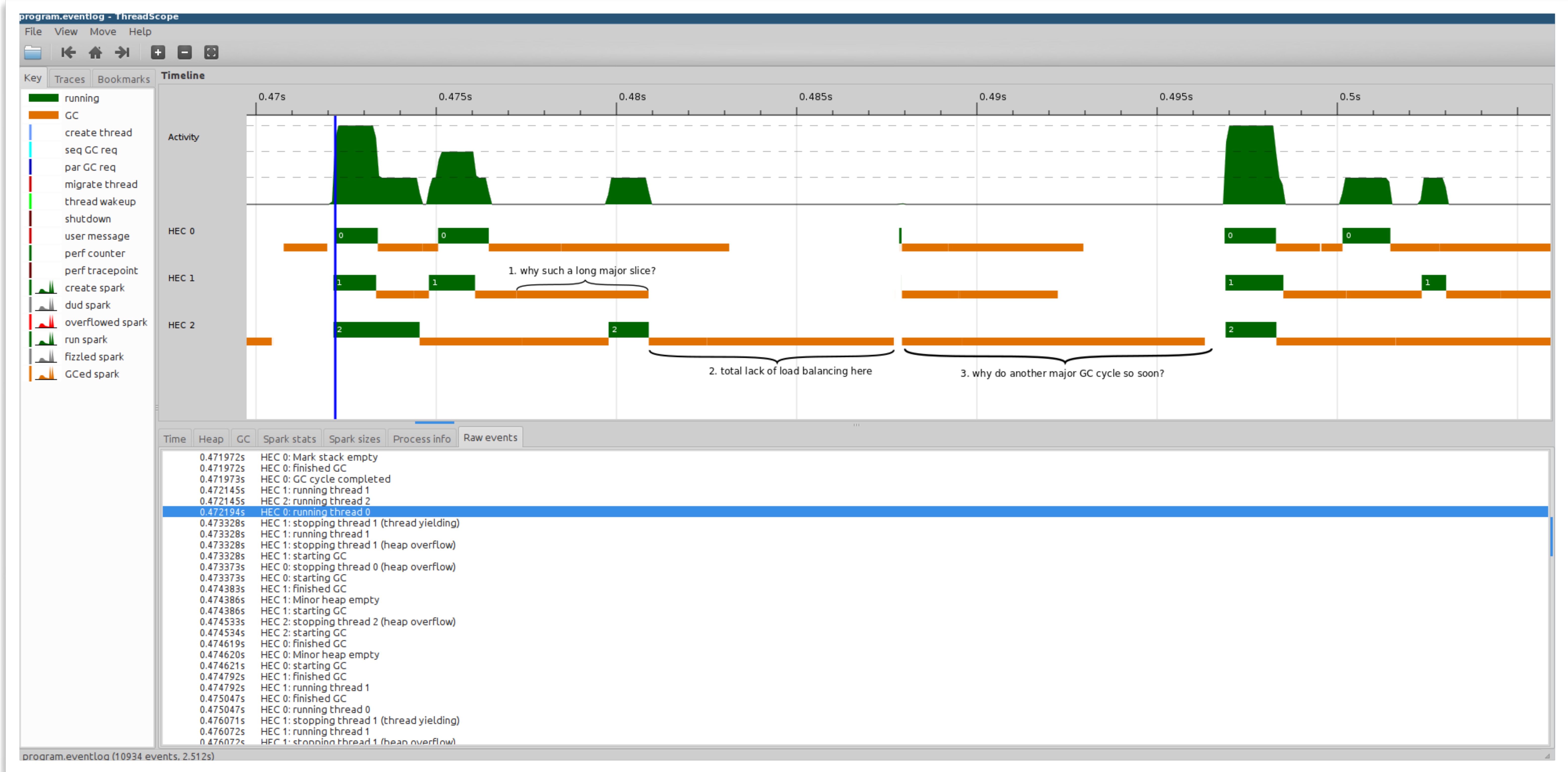
Invest in tooling

Reuse existing tools; if not build them!

- rr = gdb + *record-and-replay debugging*
- OCaml 5 + *ThreadSanitizer*
  - ✦ *Detect data races dynamically*

Invest in tooling

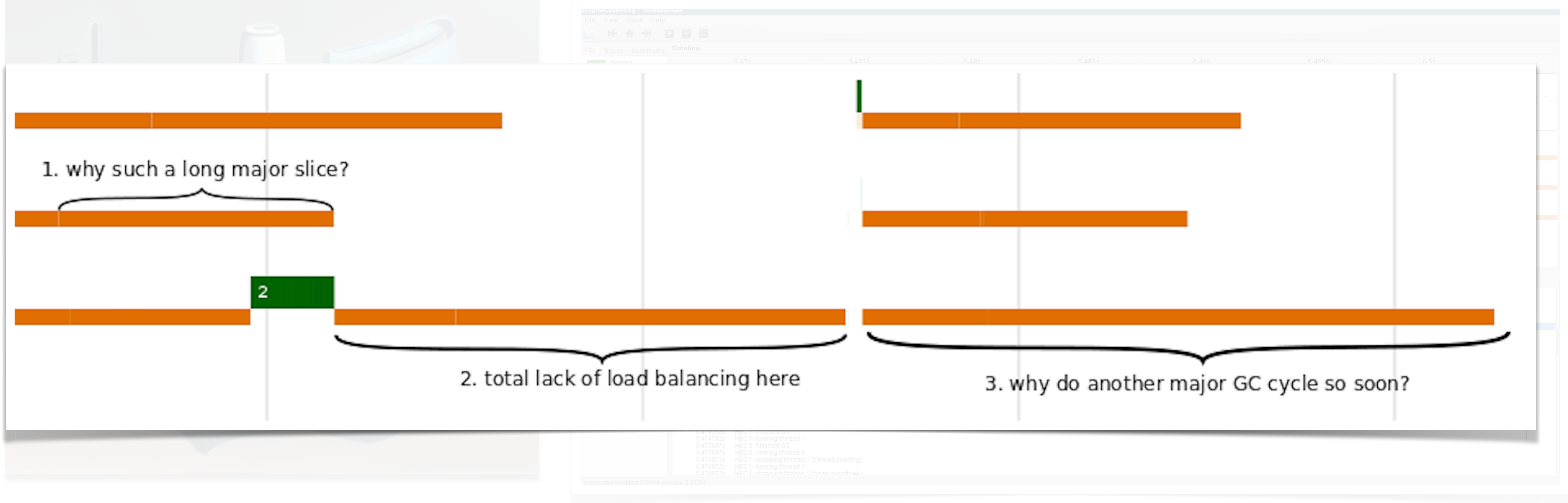
Tracing



GHC's ThreadScope

Invest in tooling

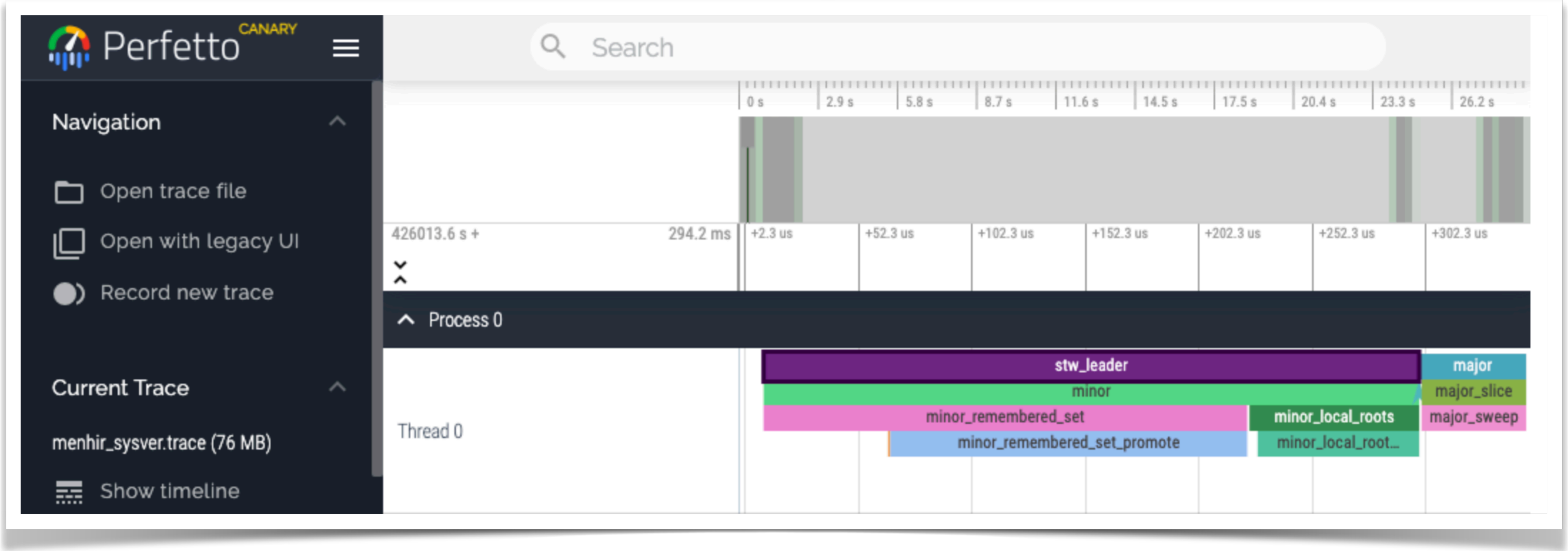
Tracing



GHC's ThreadScope

Invest in tooling

Tracing



Runtime Events: CTF-based tracing

Invest in tooling

OCaml Users and Developers Workshop 2022

Fri 16 Sep 2022 11:00 - 11:20 at M1 - Session 2 Chair(s): Oleg Kiselyov

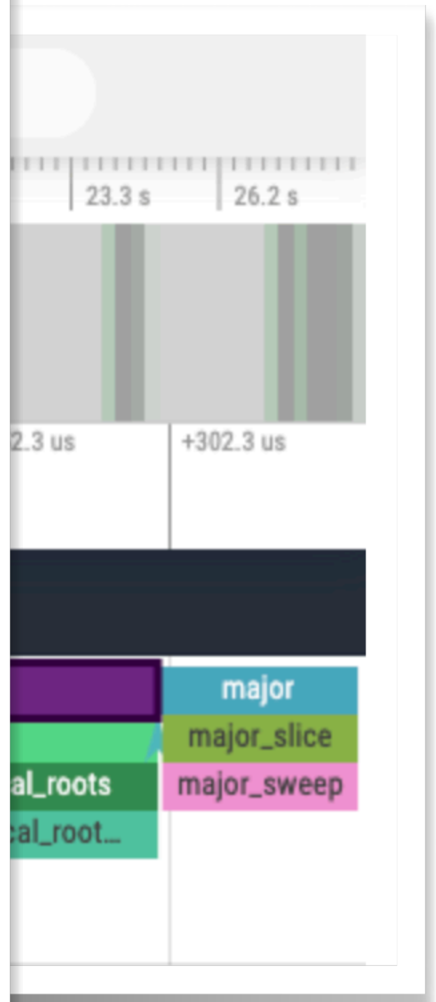
★ Continuous Monitoring of OCaml Applications using Runtime Events

VIRTUAL

The upcoming 5.0 release of OCaml includes a new runtime tracing system designed for continuous monitoring of OCaml applications called Runtime Events. It enables very low overhead, programmatic access to performance data emitted by the OCaml runtime and garbage collector. This talk focuses on the implementation of Runtime Events and the user experience of writing applications exploiting this new feature.

Sadiq Jaffer
University of Cambridge and Tarides
United Kingdom

Patrick Ferris
Tarides



Convincing caml-devel

- Quite a challenge maintaining a separate fork for 7+ years!
  - Multiple *rebases* to keep it up-to-date with mainline
  - In hindsight, smaller PRs are better

Convincing caml-devel

- Quite a challenge maintaining a separate fork for 7+ years!
  - Multiple *rebases* to keep it up-to-date with mainline
  - In hindsight, smaller PRs are better
- *Peer-reviewed papers* adds credibility to the effort

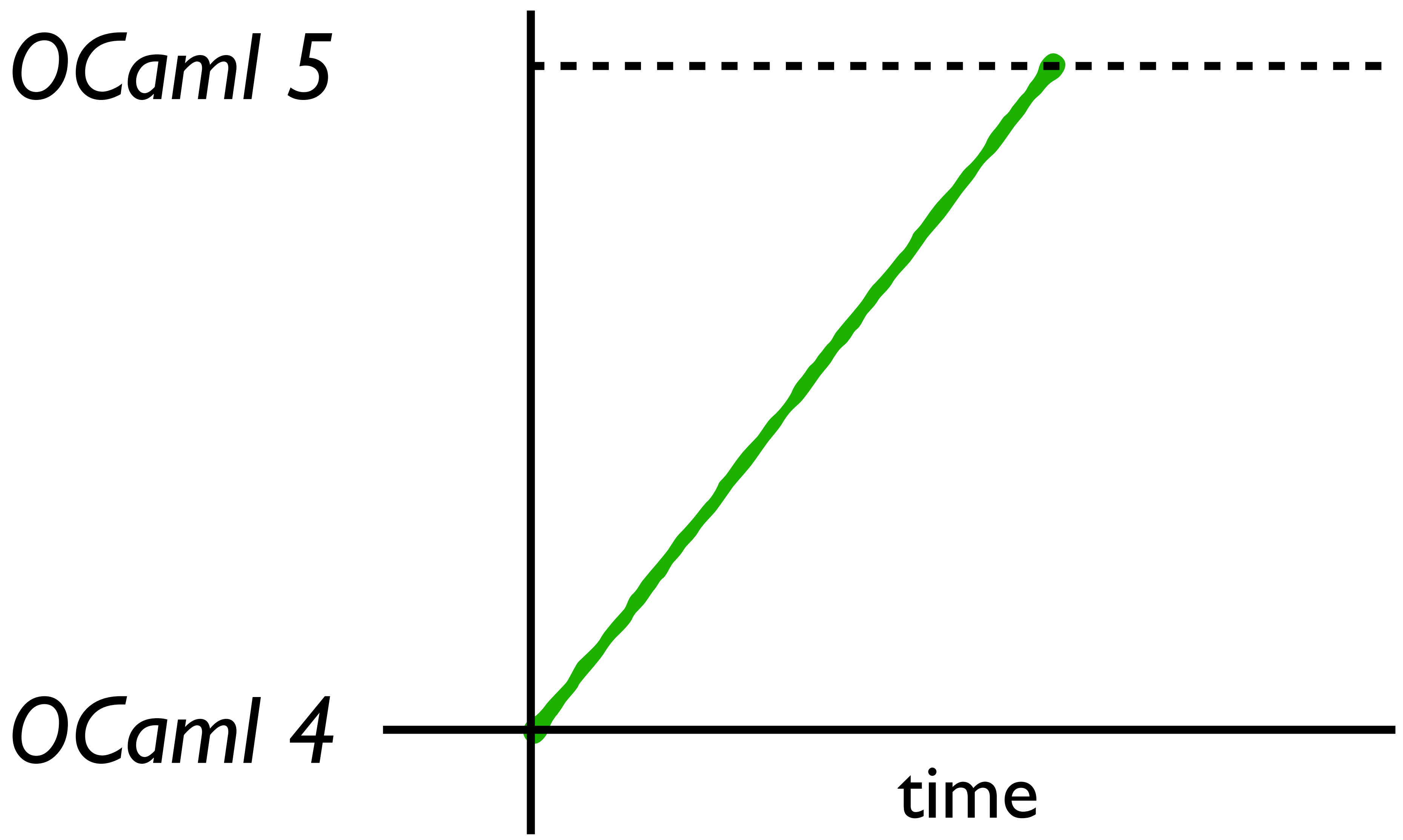
Convincing caml-devel

- Quite a challenge maintaining a separate fork for 7+ years!
  - Multiple *rebases* to keep it up-to-date with mainline
  - In hindsight, smaller PRs are better
- *Peer-reviewed papers* adds credibility to the effort
- *Open-source* and actively-maintained always
  - Lots of (academic) users from early days

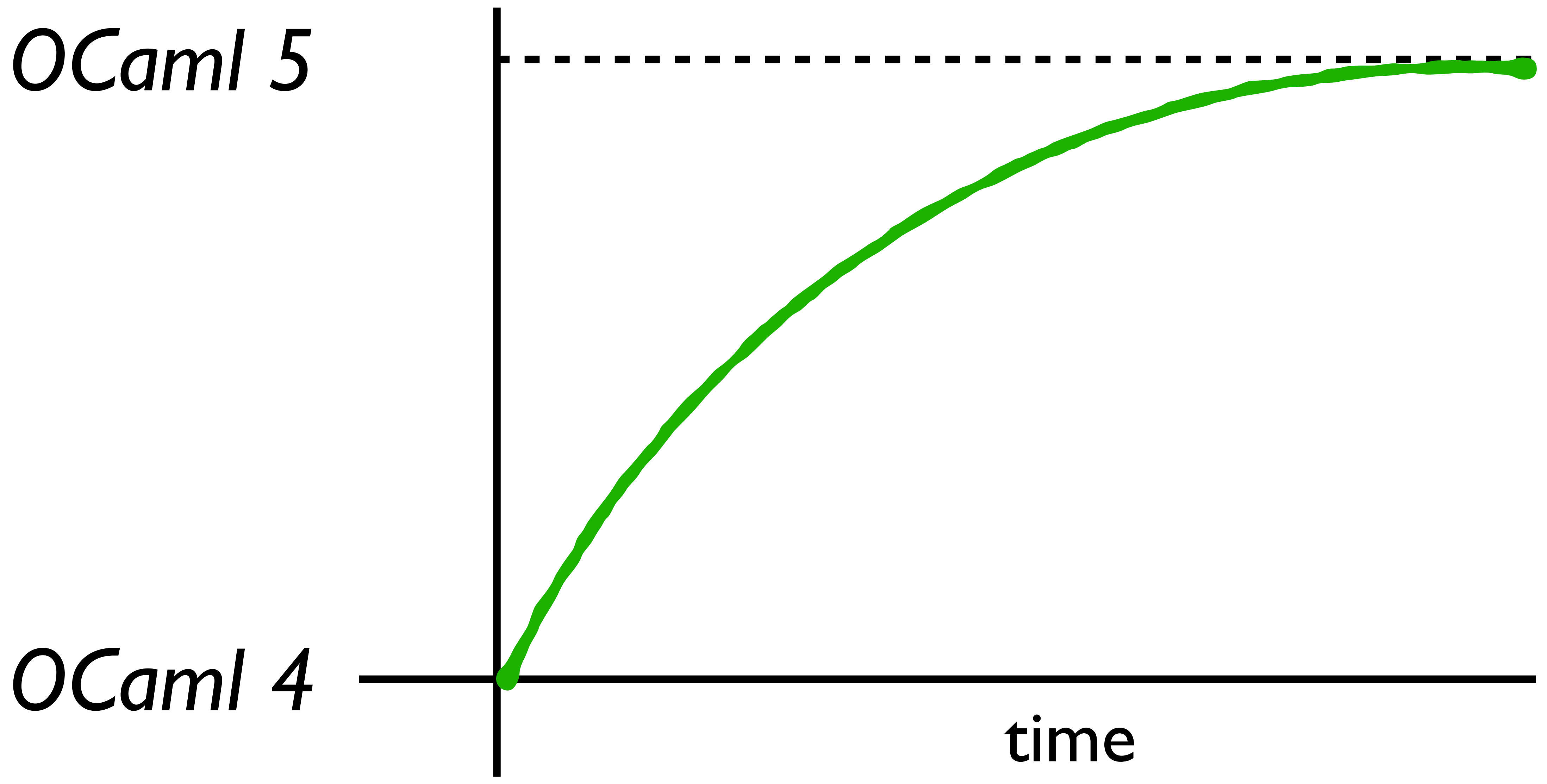
Convincing caml-devel

- Quite a challenge maintaining a separate fork for 7+ years!
  - Multiple *rebases* to keep it up-to-date with mainline
  - In hindsight, smaller PRs are better
- *Peer-reviewed papers* adds credibility to the effort
- *Open-source* and actively-maintained always
  - Lots of (academic) users from early days
- Continuous benchmarking, OPAM health check

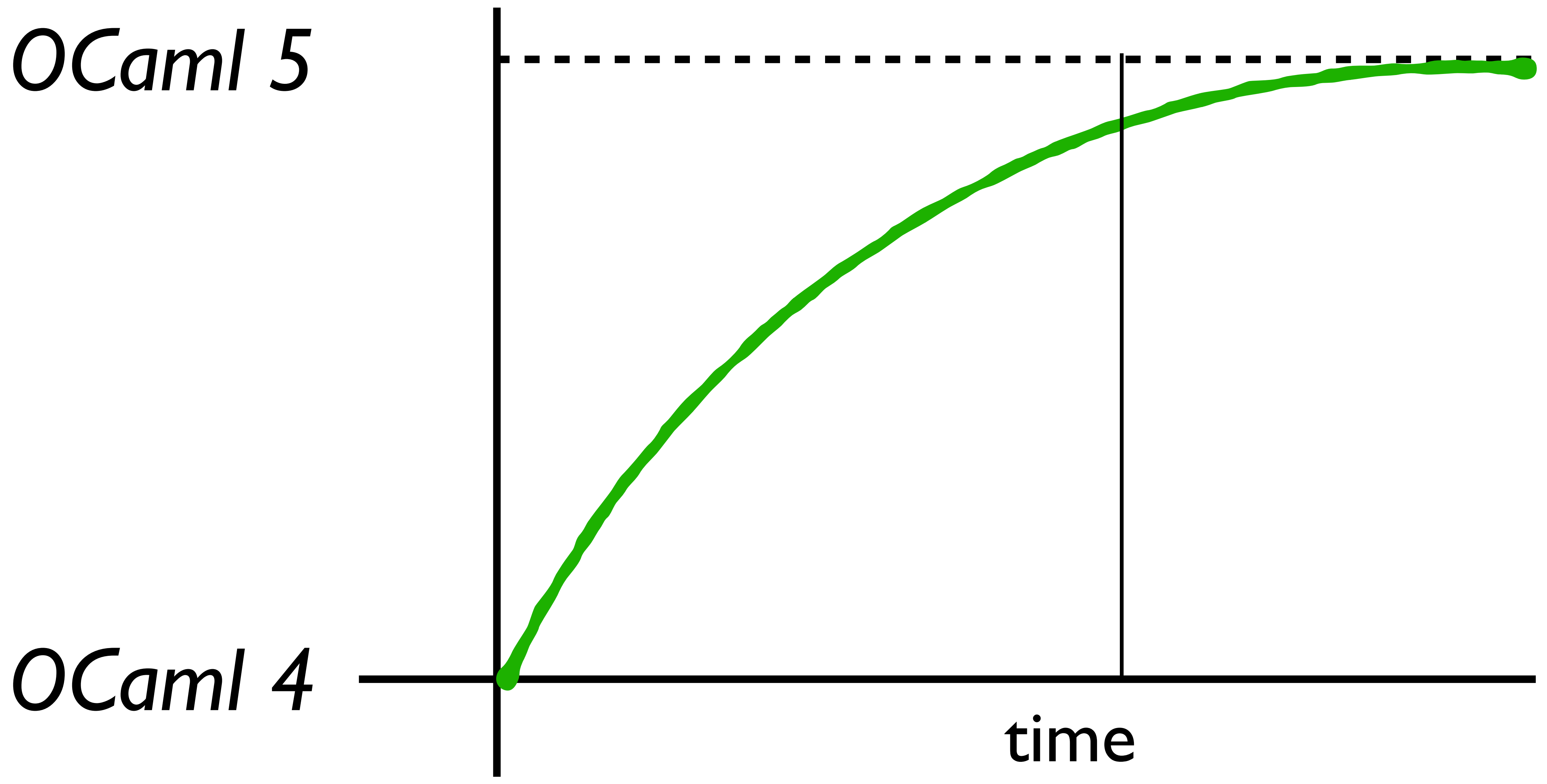
Growing the language



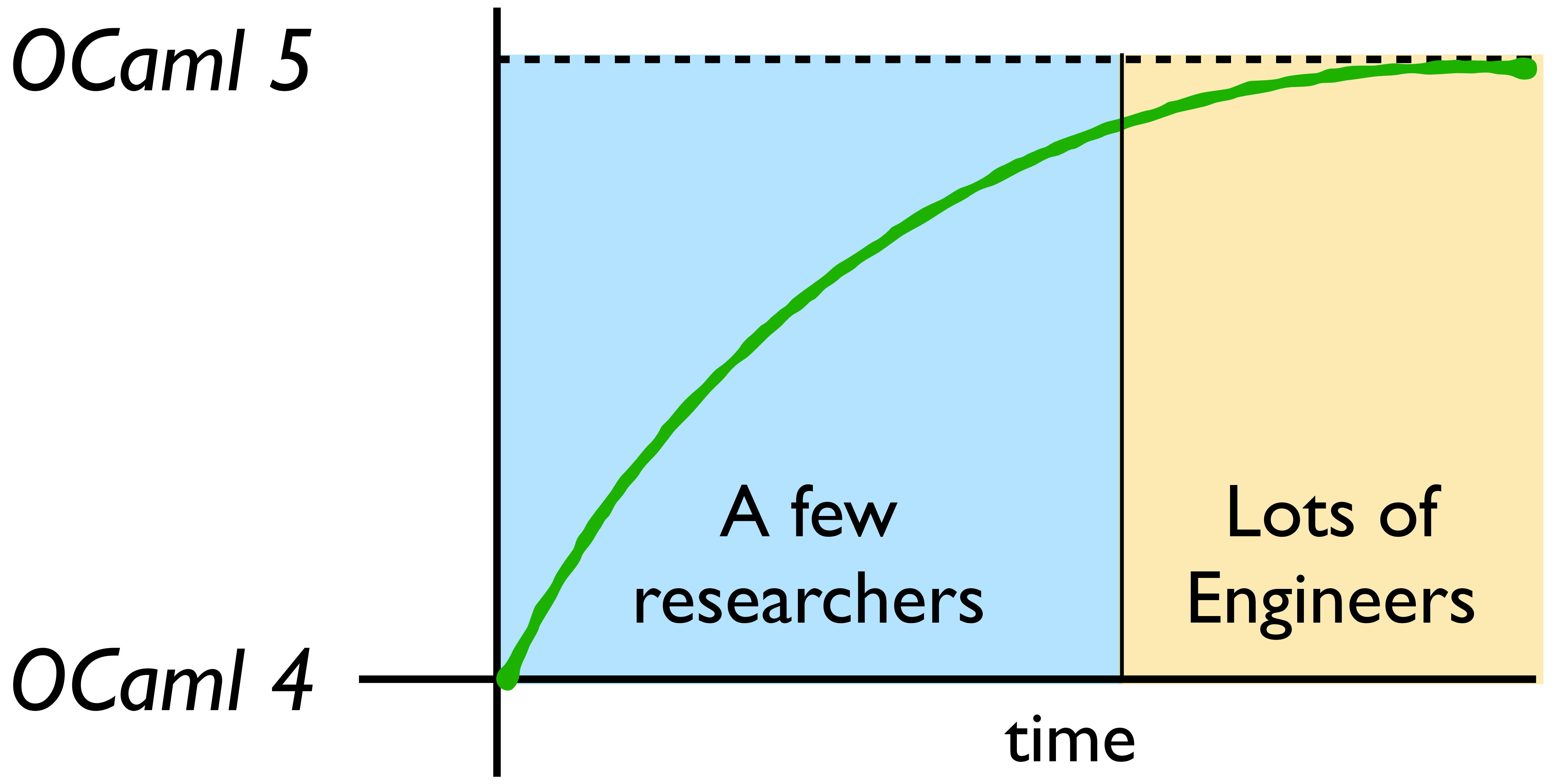
Growing the language



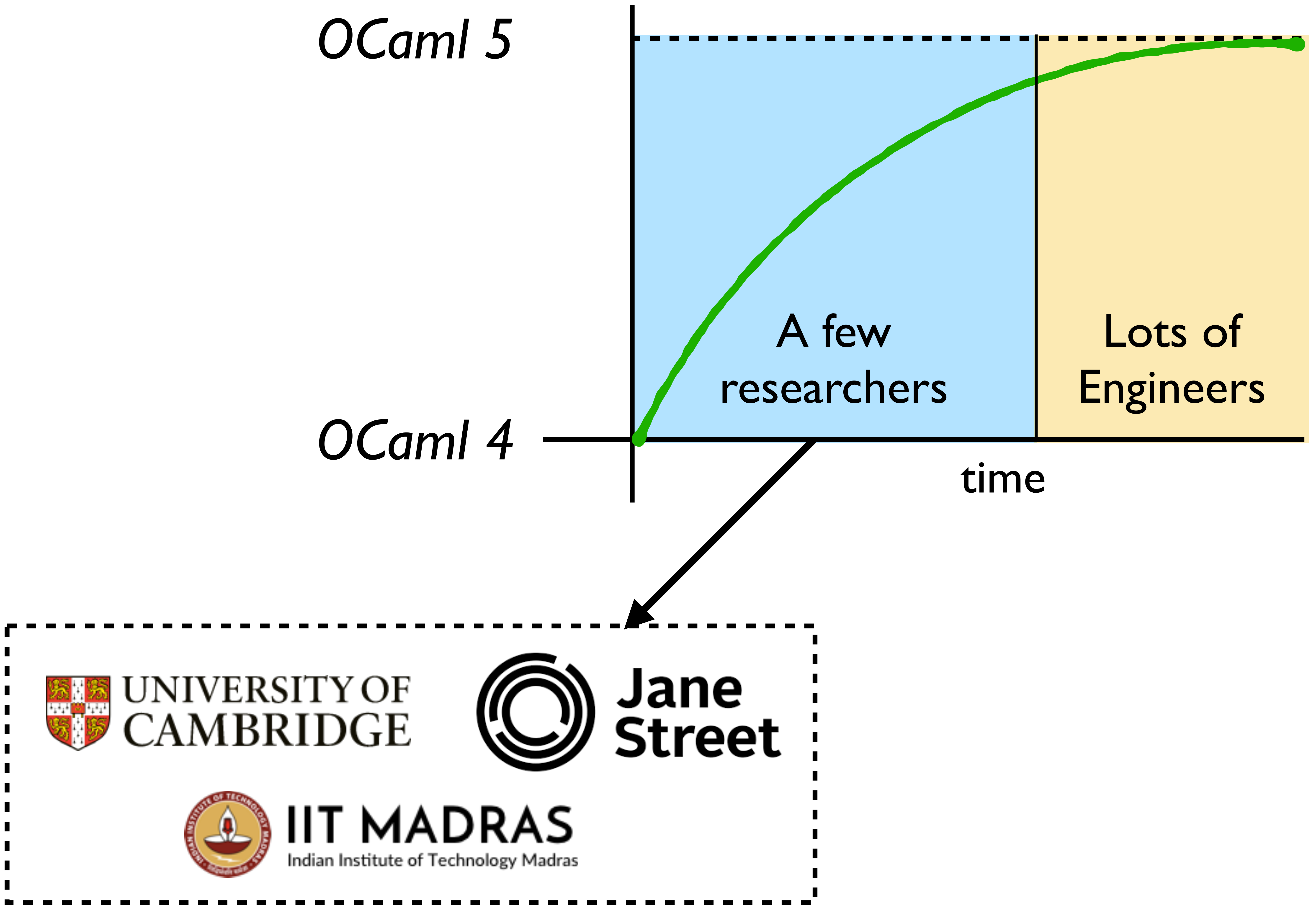
Growing the language



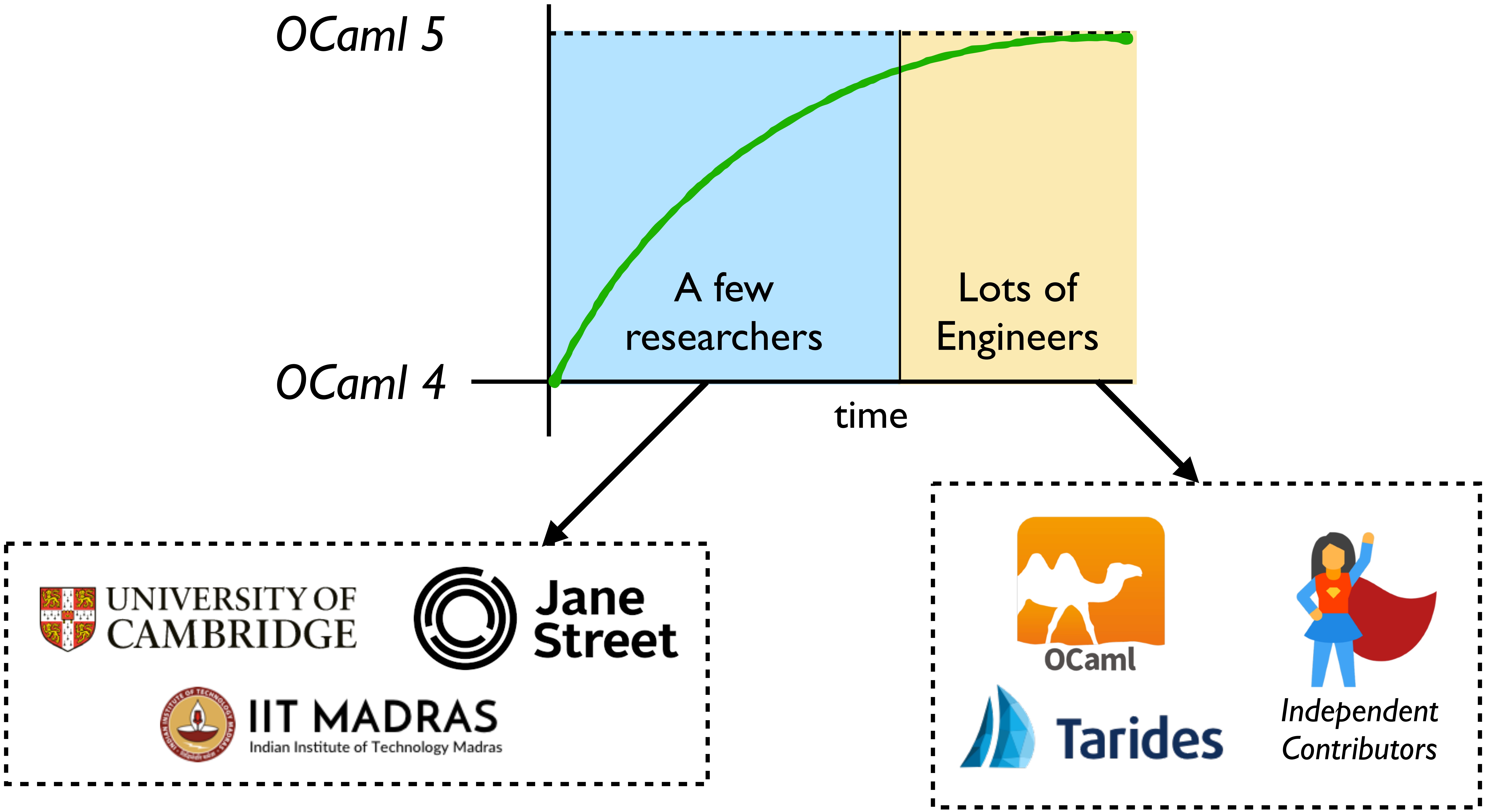
Growing the language



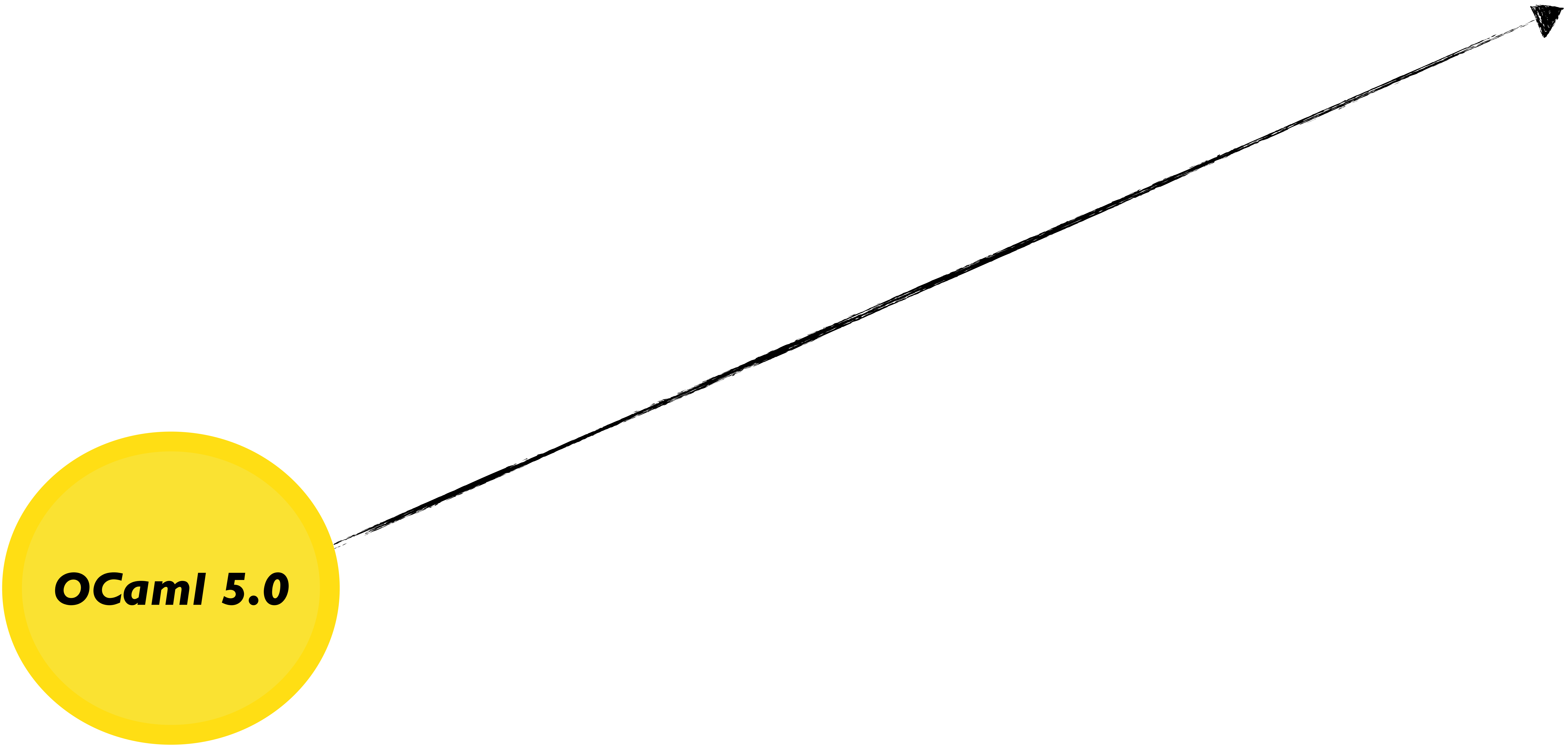
Growing the language



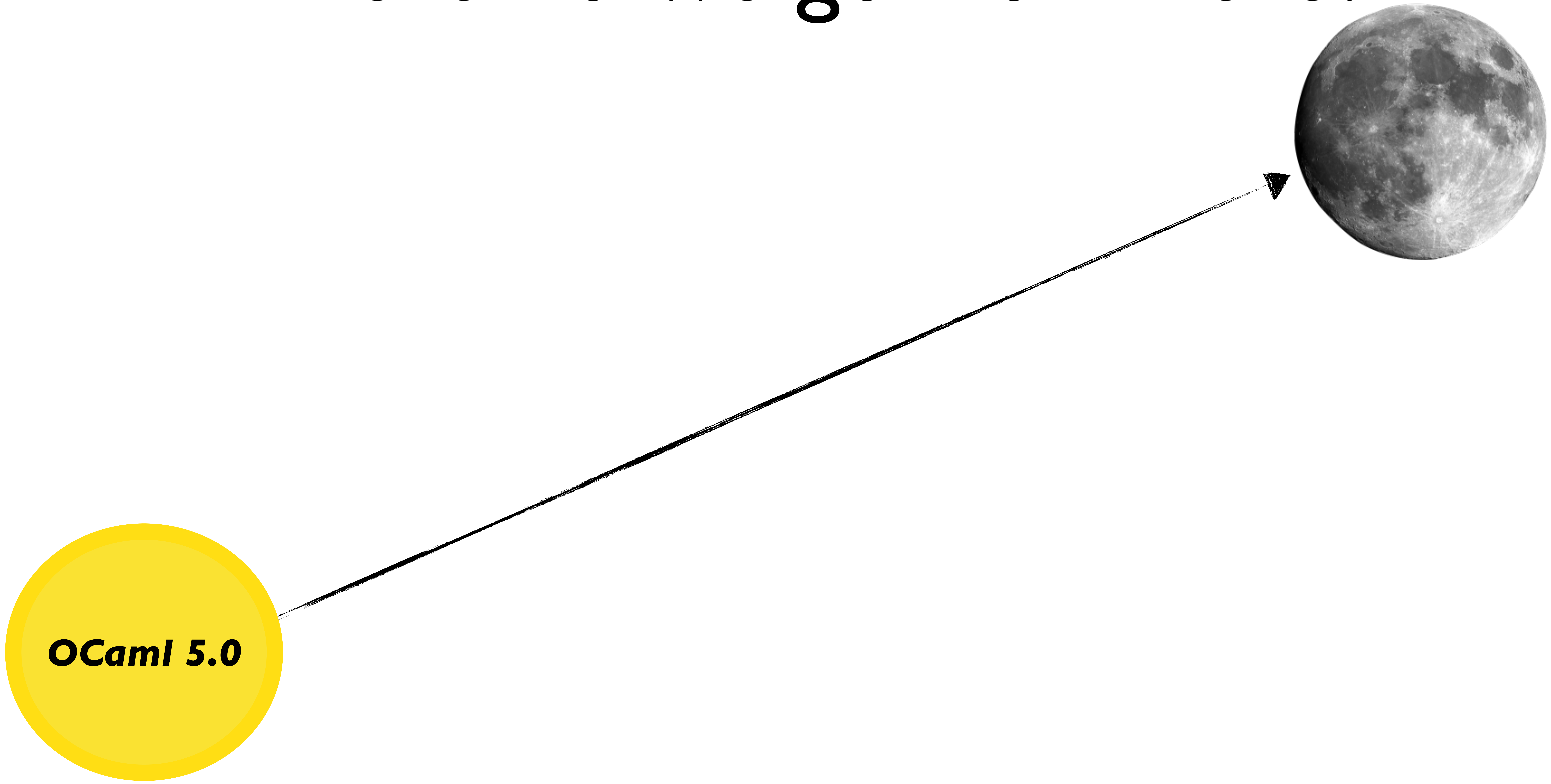
Growing the language



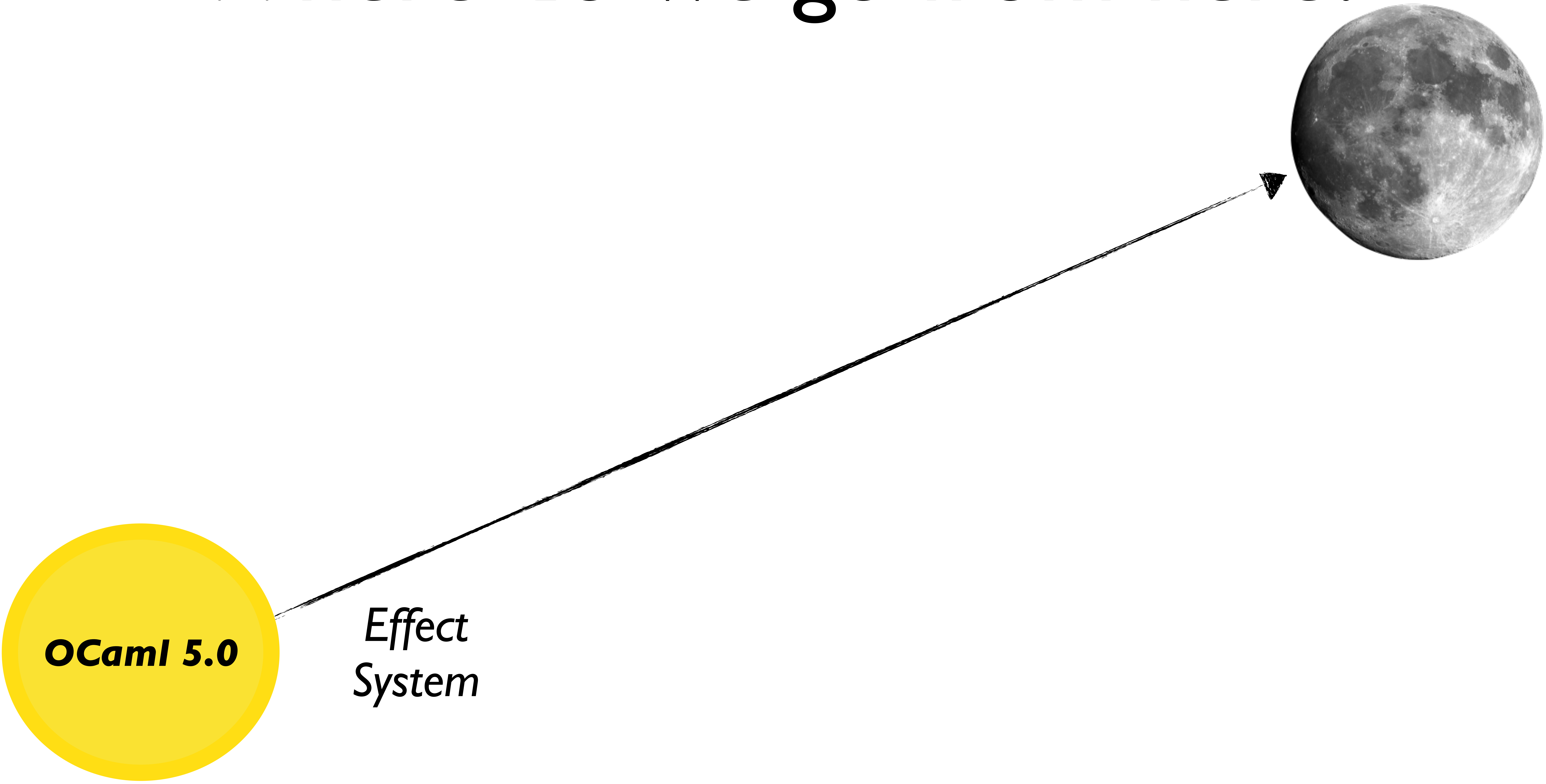
Where do we go from here?



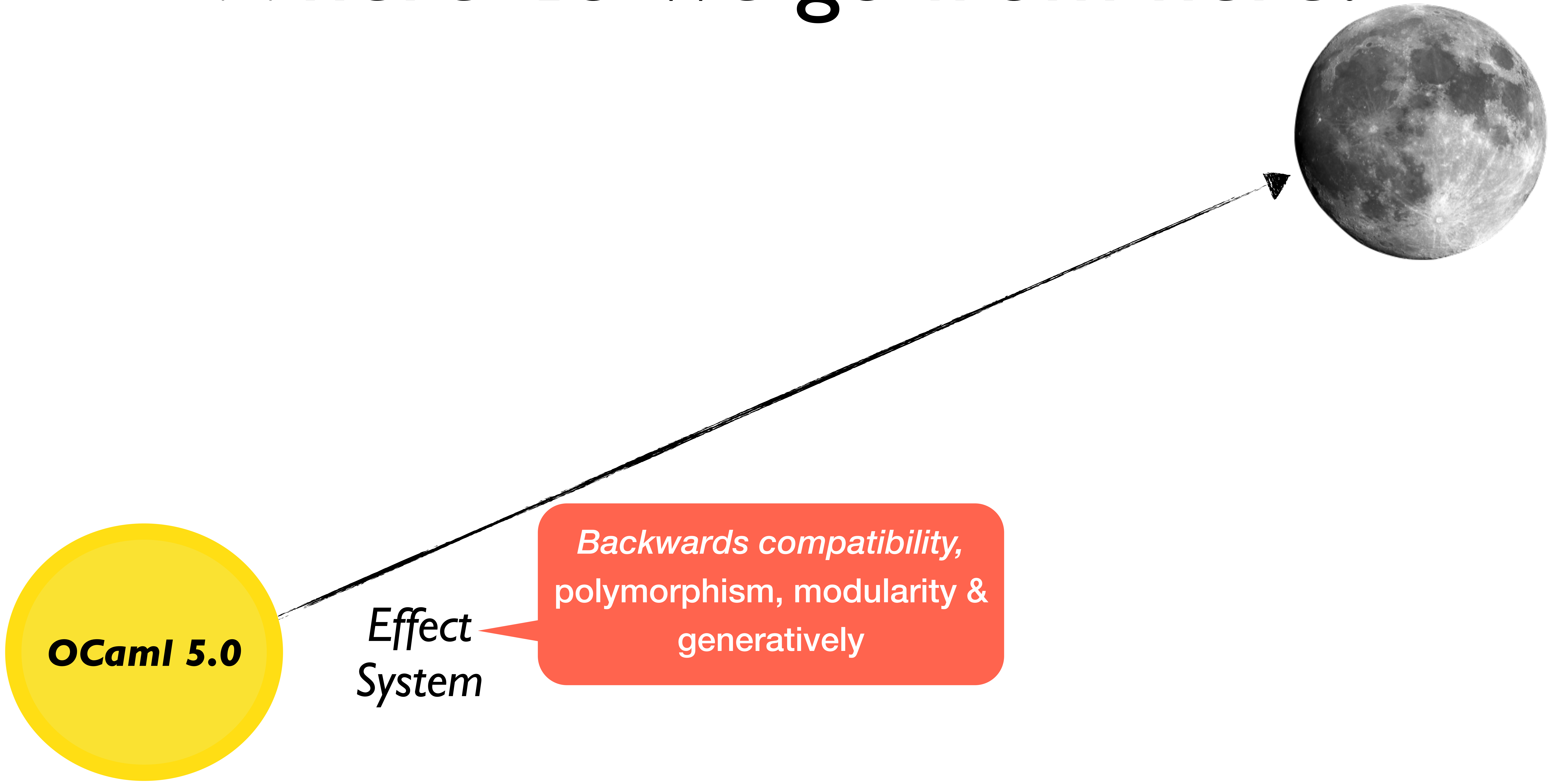
Where do we go from here?



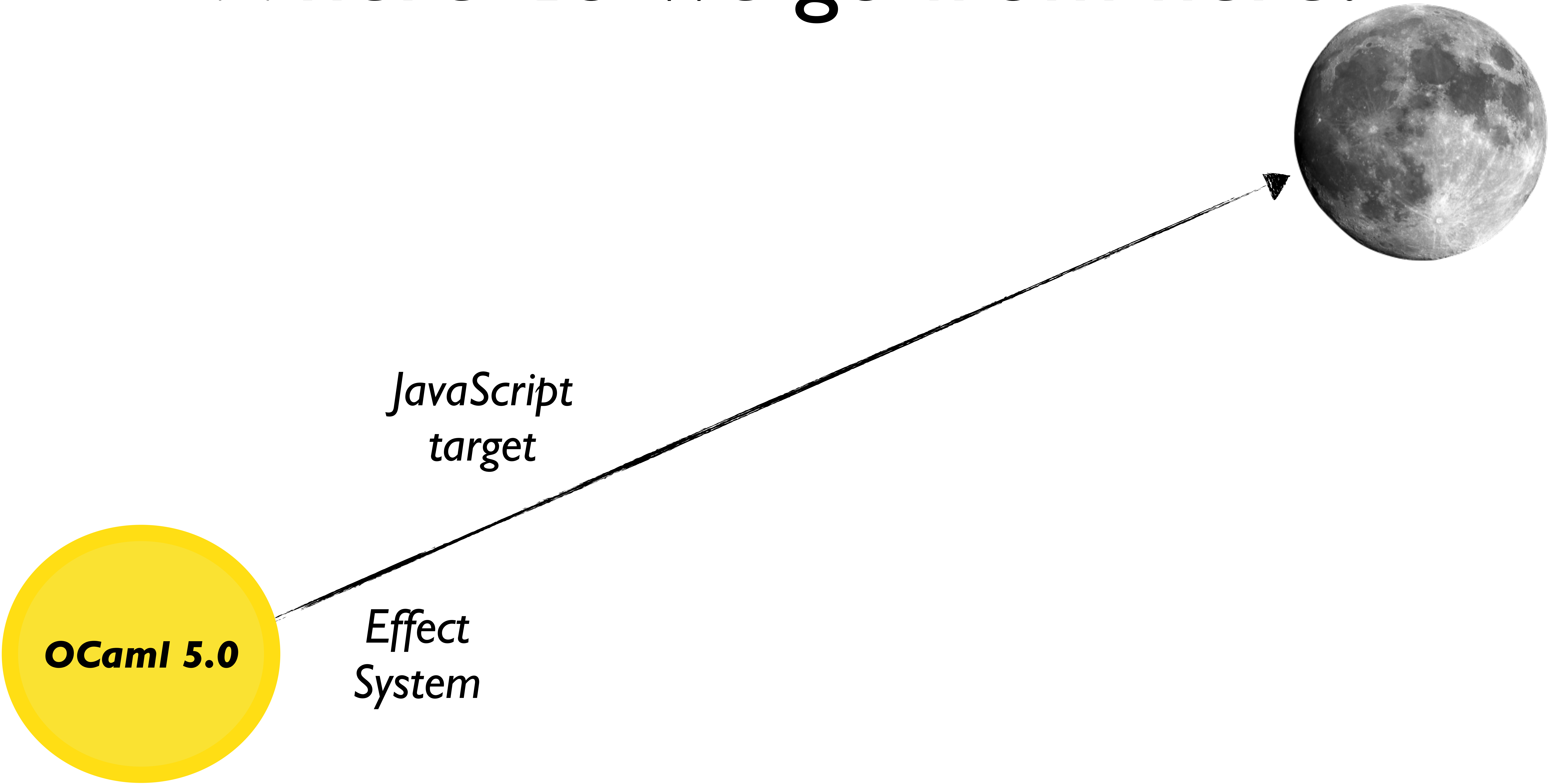
Where do we go from here?



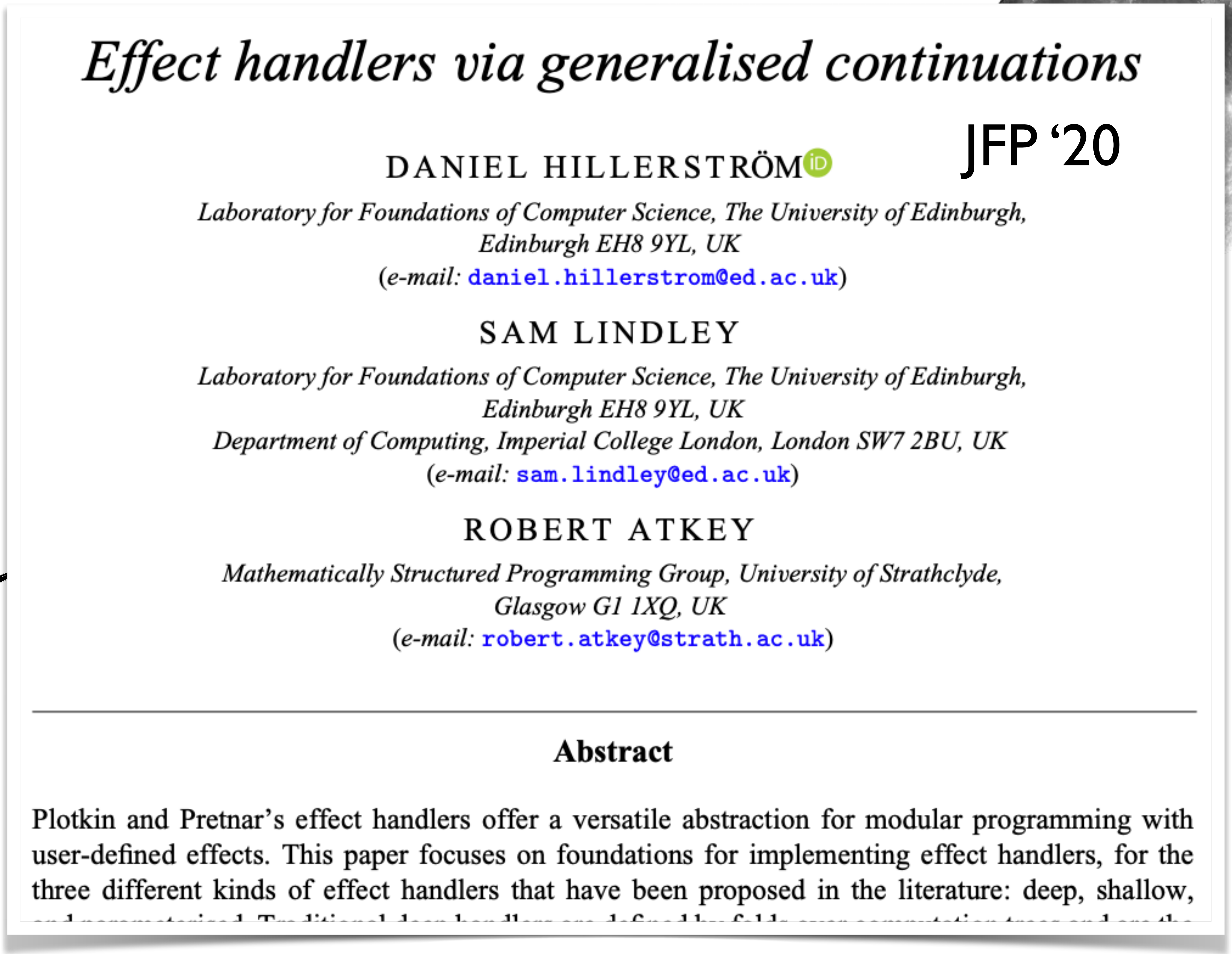
Where do we go from here?



Where do we go from here?



Where do we go from here?



Effect handlers via generalised continuations

DANIEL HILLERSTRÖM

Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH8 9YL, UK

(*e-mail:* daniel.hillerstrom@ed.ac.uk)

SAM LINDLEY

Laboratory for Foundations of Computer Science, The University of Edinburgh, Edinburgh EH8 9YL, UK

Department of Computing, Imperial College London, London SW7 2BU, UK

(*e-mail:* sam.lindley@ed.ac.uk)

ROBERT ATKEY

Mathematically Structured Programming Group, University of Strathclyde, Glasgow G1 1XQ, UK

(*e-mail:* robert.atkey@strath.ac.uk)

Abstract

Plotkin and Pretnar's effect handlers offer a versatile abstraction for modular programming with user-defined effects. This paper focuses on foundations for implementing effect handlers, for the three different kinds of effect handlers that have been proposed in the literature: deep, shallow,

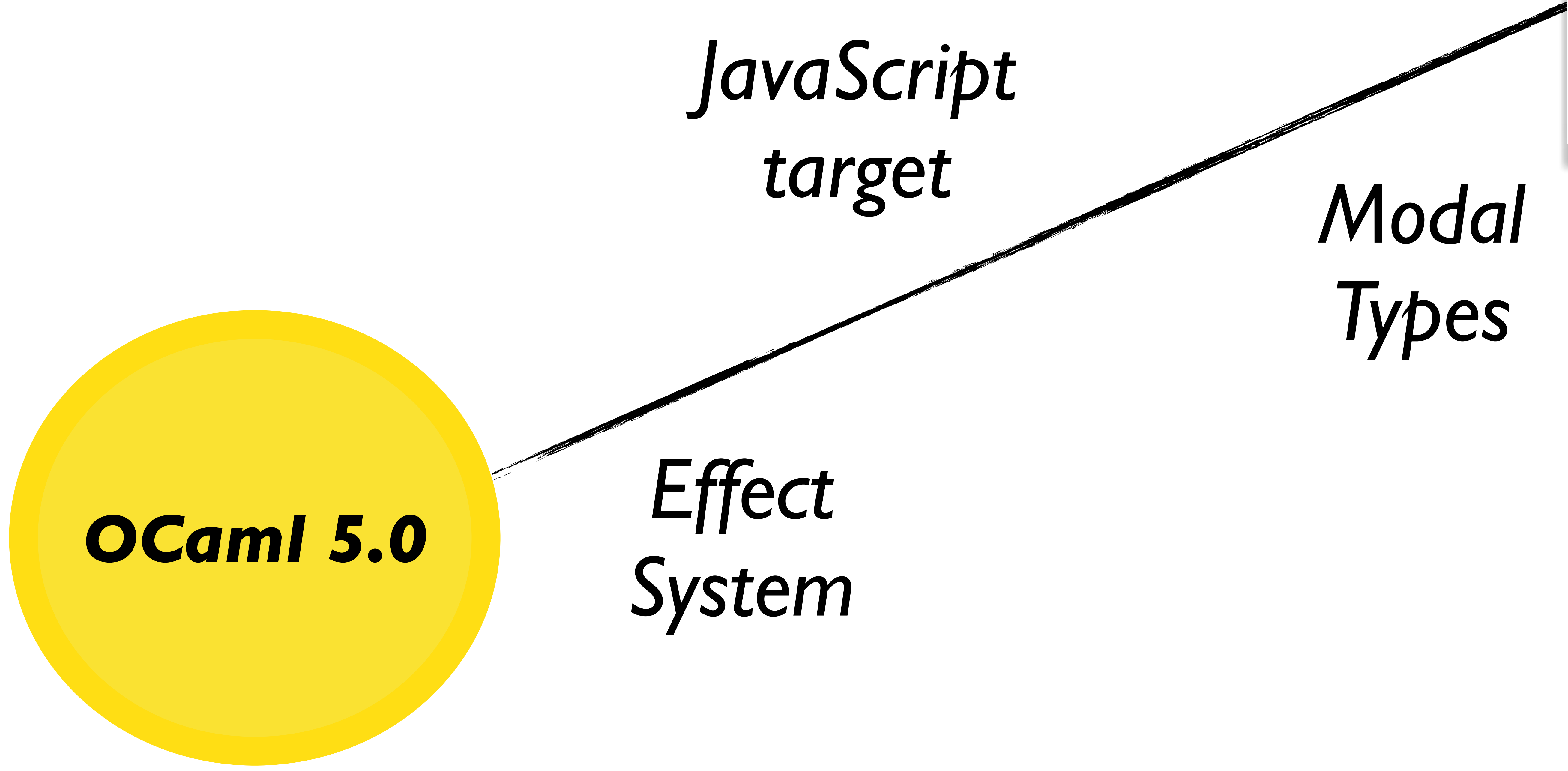
JFP '20

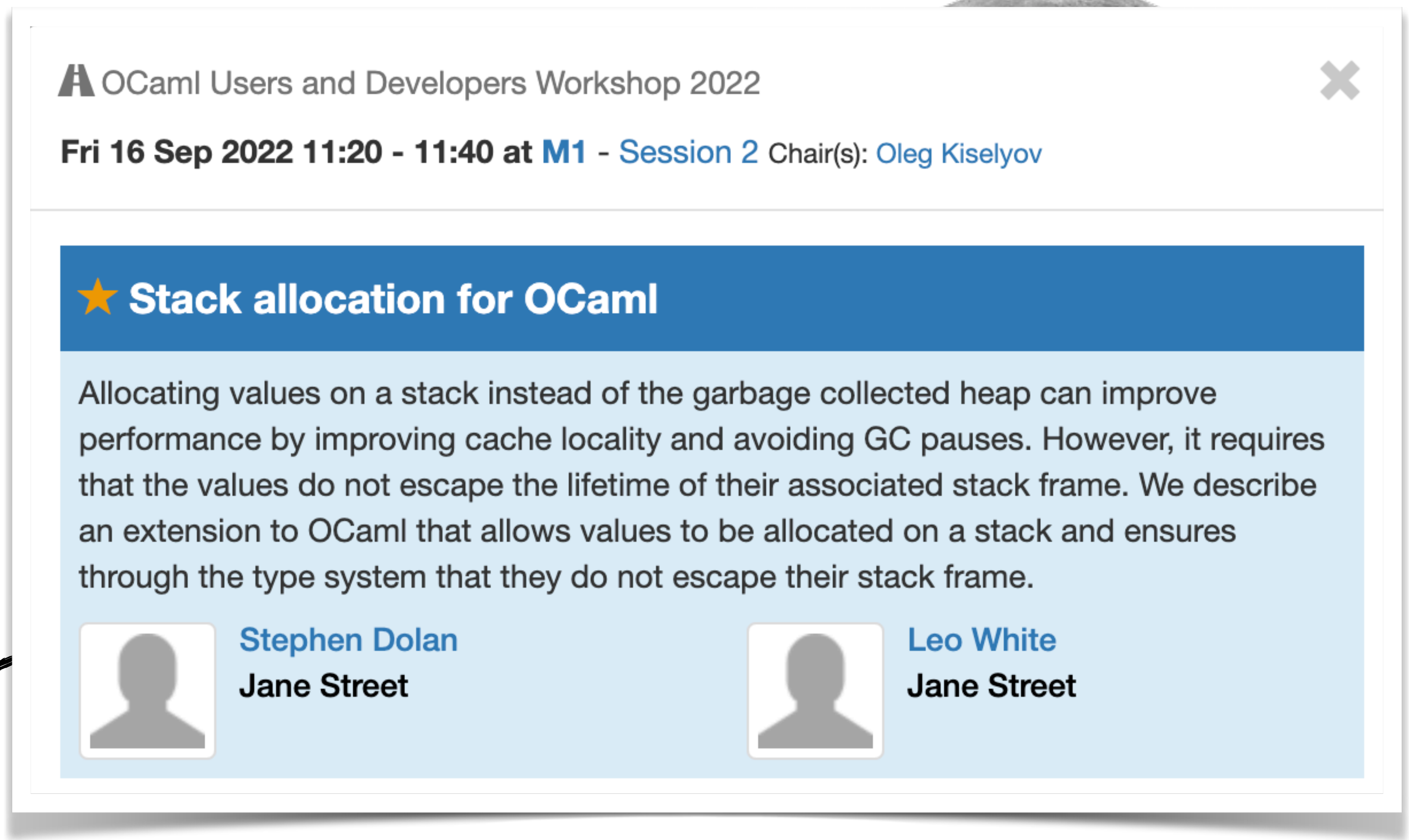
JavaScript target

OCaml 5.0

Effect System

Where do we go from here?




OCaml Users and Developers Workshop 2022

Fri 16 Sep 2022 11:20 - 11:40 at M1 - Session 2 Chair(s): Oleg Kiselyov

Stack allocation for OCaml

Allocating values on a stack instead of the garbage collected heap can improve performance by improving cache locality and avoiding GC pauses. However, it requires that the values do not escape the lifetime of their associated stack frame. We describe an extension to OCaml that allows values to be allocated on a stack and ensures through the type system that they do not escape their stack frame.

Stephen Dolan
Jane Street

Leo White
Jane Street

Where do we go from here?

OCaml Users and Developers Workshop 2022

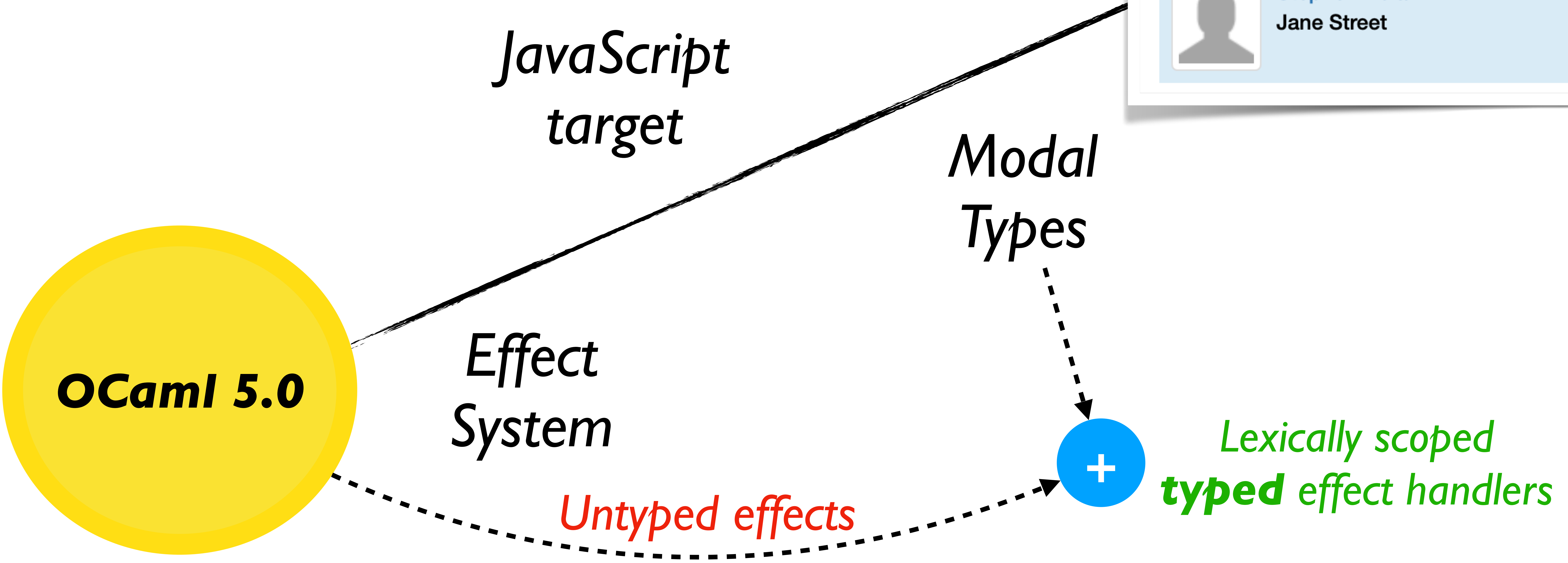
Fri 16 Sep 2022 11:20 - 11:40 at M1 - Session 2 Chair(s): Oleg Kiselyov

Stack allocation for OCaml

Allocating values on a stack instead of the garbage collected heap can improve performance by improving cache locality and avoiding GC pauses. However, it requires that the values do not escape the lifetime of their associated stack frame. We describe an extension to OCaml that allows values to be allocated on a stack and ensures through the type system that they do not escape their stack frame.

Stephen Dolan
Jane Street

Leo White
Jane Street



Where do we go from here?

Control memory layout

Unboxed Types

Parallelism

JavaScript target

Flambda2

Aggressive compiler optimisations

OCaml 5.0

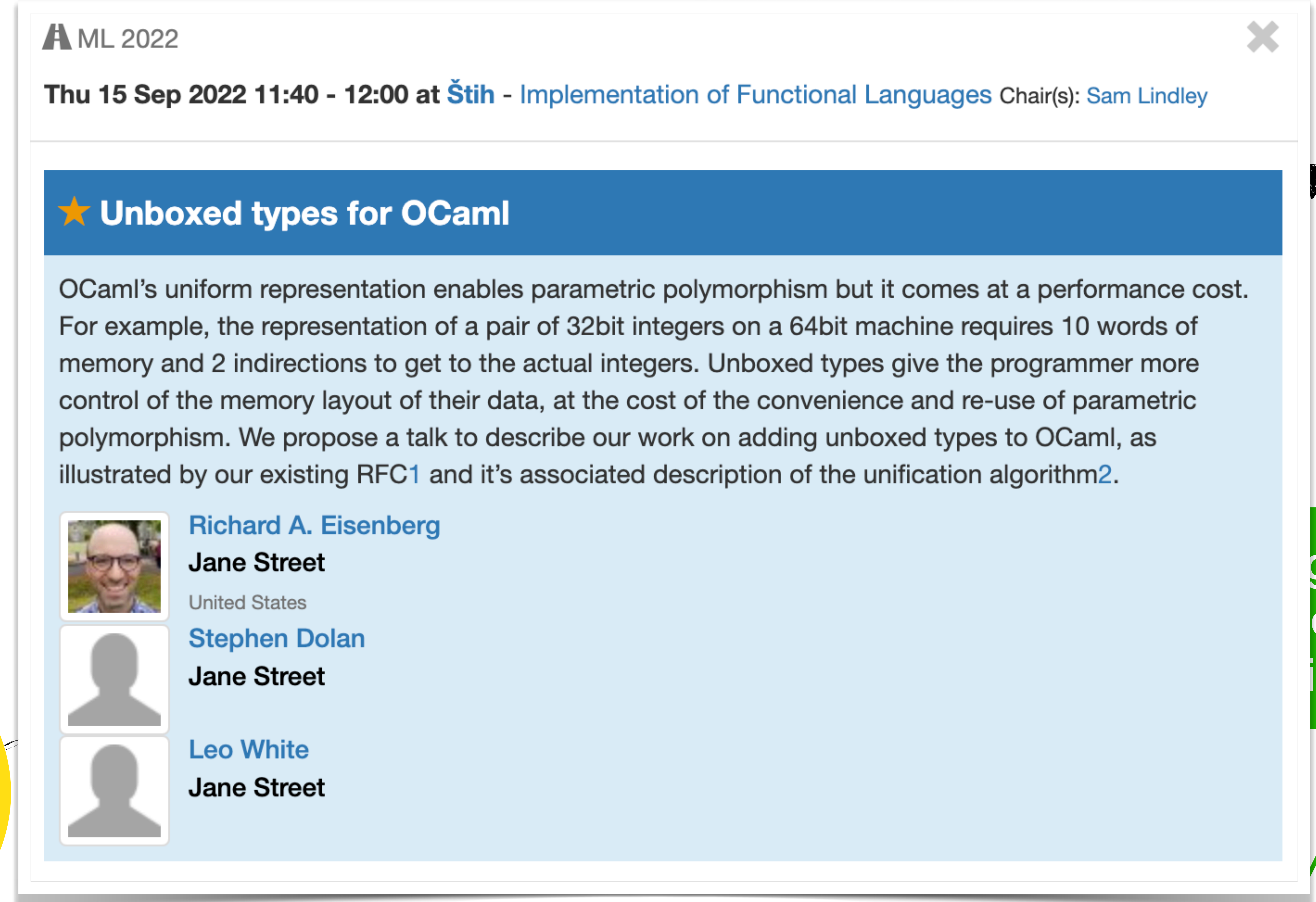
Effect System

Modal Types

Avoid heap allocations

Rust/C-like performance (on demand), with GC as default, and the ergonomics and safety of classic ML

Where do we go from here?



ML 2022

Thu 15 Sep 2022 11:40 - 12:00 at Štih - Implementation of Functional Languages Chair(s): Sam Lindley

★ Unboxed types for OCaml

OCaml's uniform representation enables parametric polymorphism but it comes at a performance cost. For example, the representation of a pair of 32bit integers on a 64bit machine requires 10 words of memory and 2 indirections to get to the actual integers. Unboxed types give the programmer more control of the memory layout of their data, at the cost of the convenience and re-use of parametric polymorphism. We propose a talk to describe our work on adding unboxed types to OCaml, as illustrated by our existing RFC1 and it's associated description of the unification algorithm2.

Richard A. Eisenberg
Jane Street
United States

Stephen Dolan
Jane Street

Leo White
Jane Street

Parallelism

OCaml 5.0

gressive
ompiler
imisations

ith GC as default,
and the ergonomics and safety of classic ML

Enjoy OCaml 5!

Top Secret