

Evolving the OCaml programming language

KC Sivaramakrishnan
kcsrk.info

IITM CSE Bytes
4th November 2025



Tarides

Who am I – KC Sivaramakrishnan

- CS Prof at IIT Madras
 - Programming languages, formal verification and systems
- A core maintainer of the *OCaml* programming language
- CTO at Tarides
 - Building functional systems using *OCaml*
 - Maintainers of the OCaml compiler and platform tools

File Edit Search Run Compile Debug Project Options Window Help

[■] == NONAME00.CPP == 1=[▲]■

1:1 ==

F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

- Turbo C++ IDE
- Learnt to program C here

- Turbo C++ IDE
- Learnt to program C here
- Believed the C language was "perfect & final"
- ...like mountains and oceans

- Turbo C++ IDE
- Learnt to program C here
- Believed the C language was "perfect & final"
- ...like mountains and oceans
- Grew up and realised neither was!

- Turbo C++ IDE
- Learnt to program C here
- Believed the C language was "perfect & final"
- ...like mountains and oceans
- Grew up and realised neither was!
- This talk is about the evolution of programming languages
- Specifically, OCaml





Language

- Functional-first but multi-paradigm (imperative, OO)
- Static-type system with Hindley-Milner type inference
- Advanced features – powerful module system, GADTs, Polymorphic variants
- Multicore support and ***effect handlers***



Language

- Functional-first but multi-paradigm (imperative, OO)
- Static-type system with Hindley-Milner type inference
- Advanced features – powerful module system, GADTs, Polymorphic variants
- Multicore support and ***effect handlers***



Platform

- Fast, native code – x86, ARM, RISC-V, etc.
- JavaScript and WebAssembly (using ***WasmGC***) compilation
- Platform tools – editor (LSP), build system (dune), package manager (opam), docs generator (odoc), etc.



Language

- Functional-first but multi-paradigm (imperative, OO)
- Static-type system with Hindley-Milner type inference
- Advanced features – powerful module system, GADTs, Polymorphic variants
- Multicore support and *effect handlers*

Platform

- Fast, native code – x86, ARM, RISC-V, etc.
- JavaScript and WebAssembly (using *WasmGC*) compilation
- Platform tools – editor (LSP), build system (dune), package manager (opam), docs generator (odoc), etc.

Ecosystem

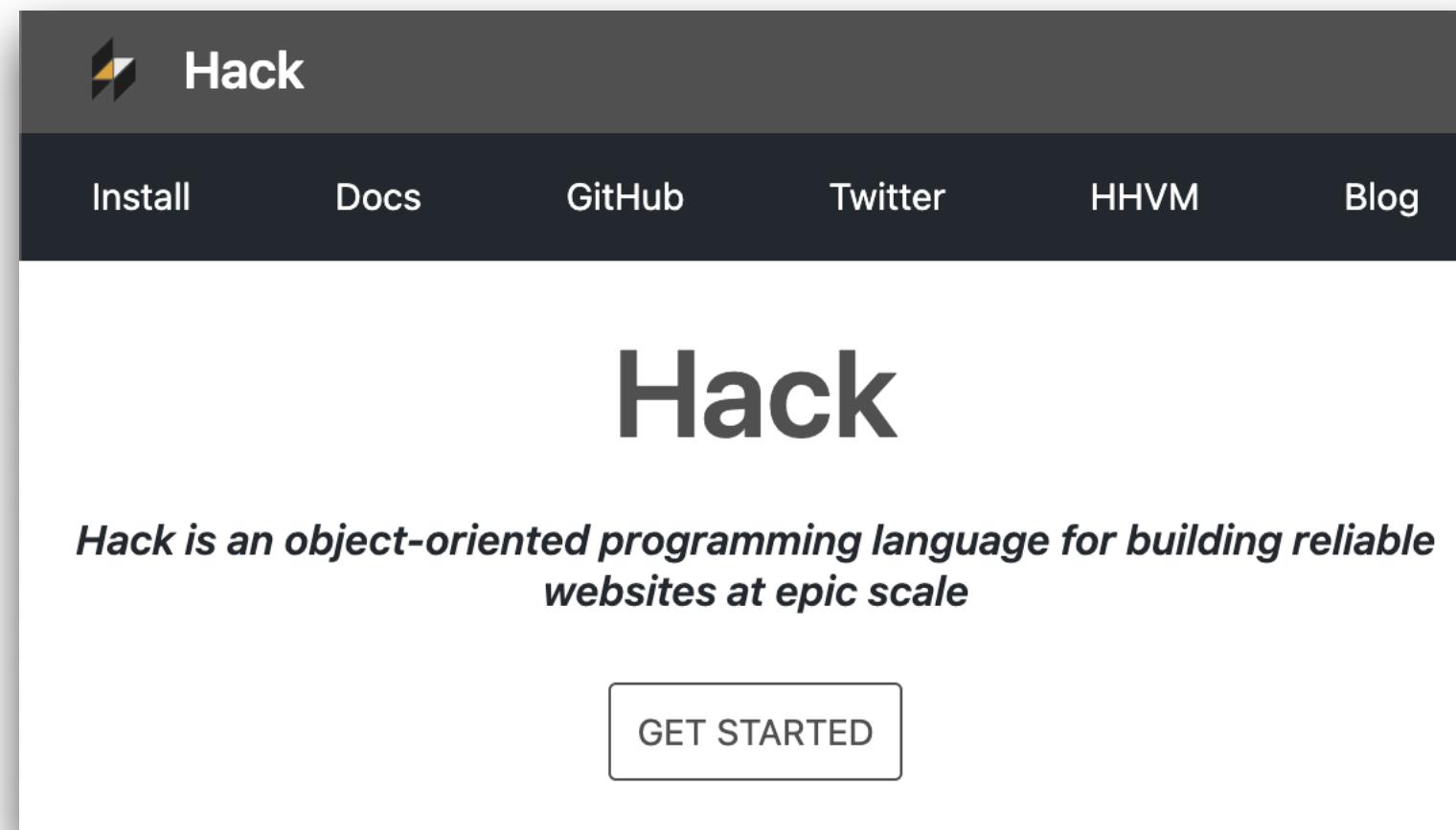
- Opam repository – small but mature package ecosystem
- Notable Industrial users – Jane Street, Meta, Microsoft, Ahrefs, Citrix, Tezos, Bloomberg, Docker

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure



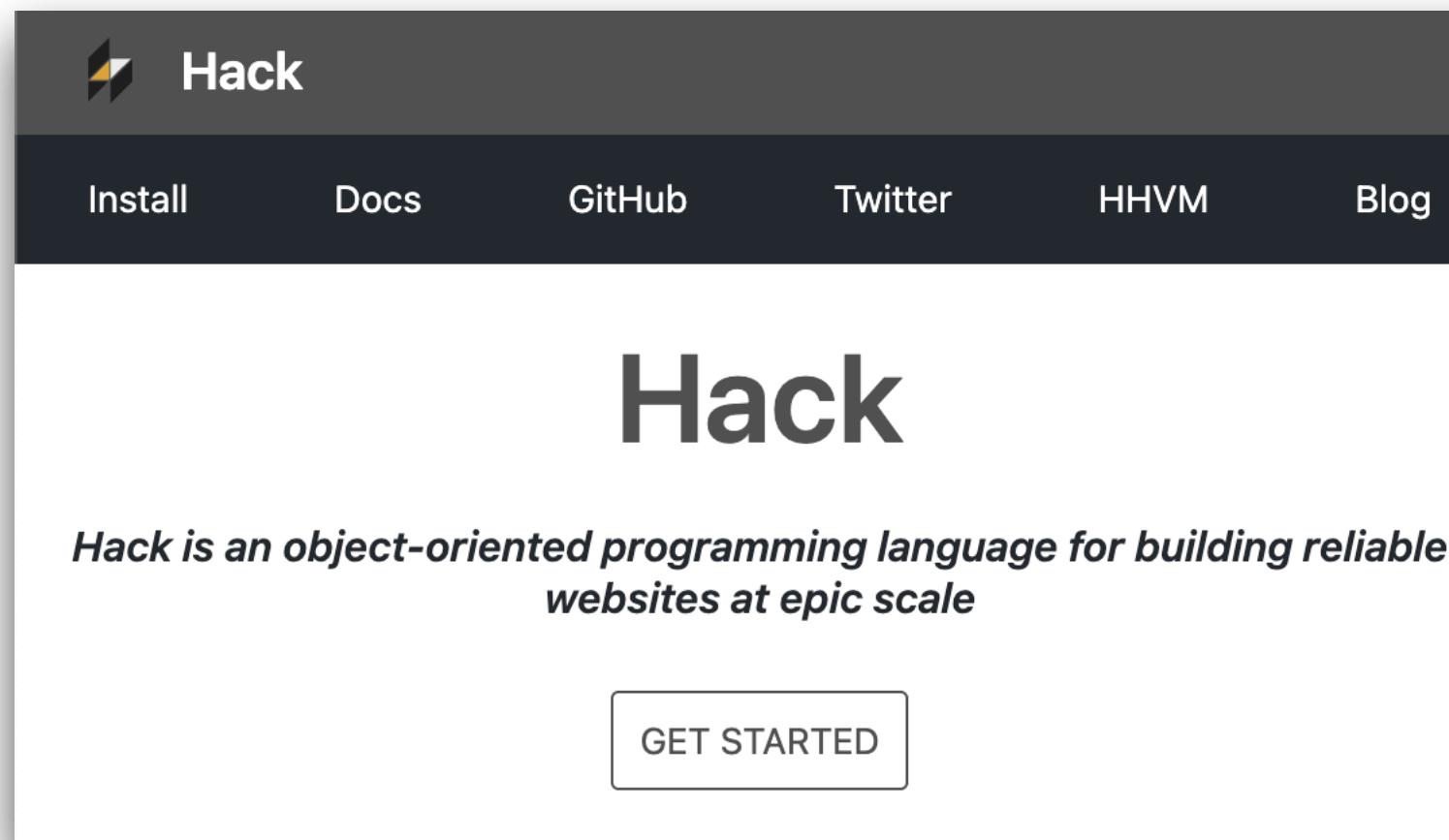
The Rust
Programming
Language



Compilers

High dynamic range

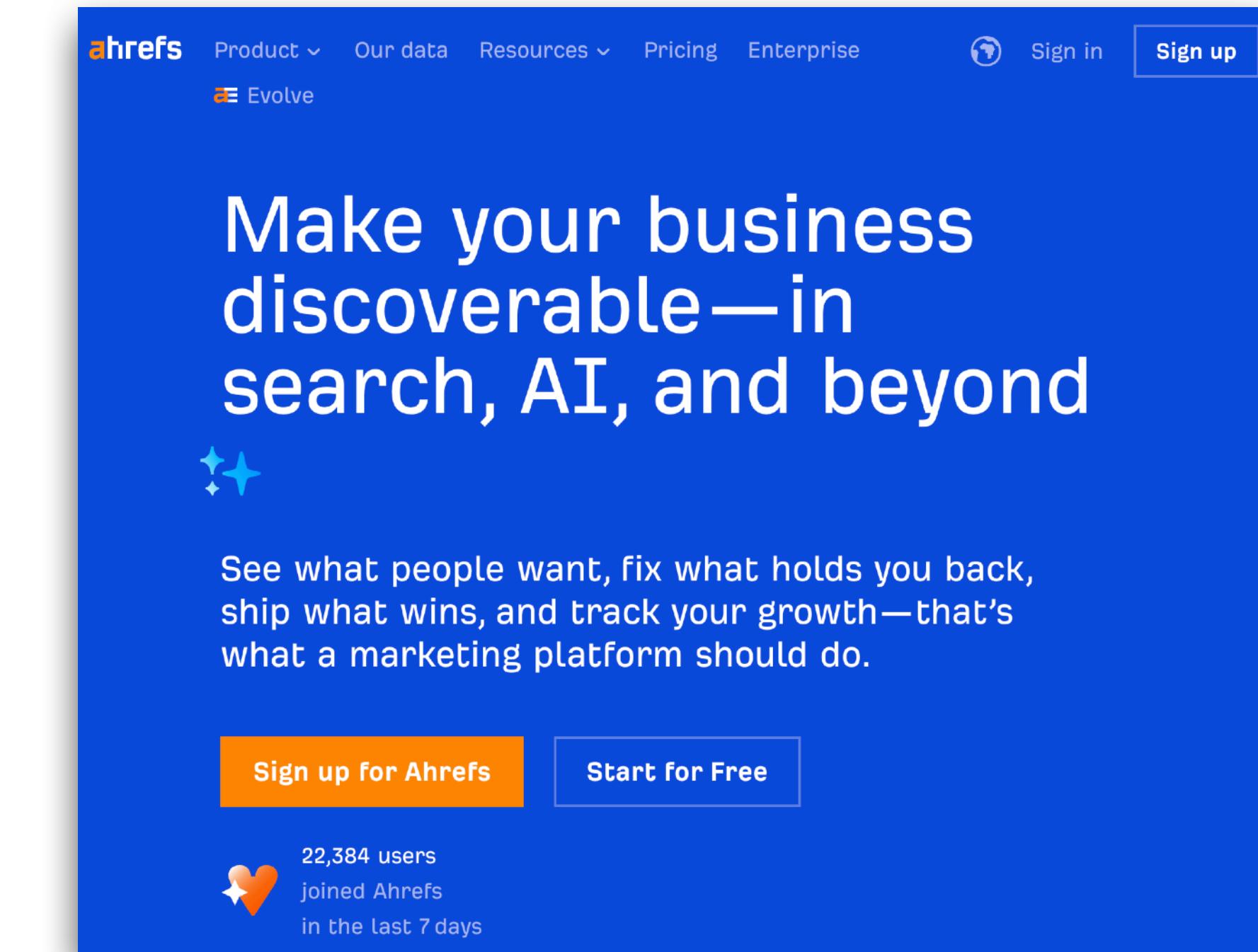
From scripts to scalable systems, research prototypes to production infrastructure



The Rust
Programming
Language



Compilers



Web Frontend

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure

Functional Networking for Millions of Docker Desktops (Experience Report)

ANIL MADHAVAPEDDY, University of Cambridge, United Kingdom

DAVID J. SCOTT, Docker, Inc., United Kingdom

PATRICK FERRIS, University of Cambridge, United Kingdom

RYAN T. GIBB, University of Cambridge, United Kingdom

THOMAS GAZAGNAIRE, Tarides, France



Docker is a developer tool used by millions of developers to build, share and run software stacks. The Docker Desktop clients for Mac and Windows have long used a novel combination of virtualisation and OCaml unikernels to seamlessly run Linux containers on these non-Linux hosts. We reflect on a decade of shipping this functional OCaml code into production across hundreds of millions of developer desktops, and discuss the lessons learnt from our experiences in integrating OCaml deeply into the container architecture that now drives much of the global cloud. We conclude by observing just how good a fit for systems programming that the unikernel approach has been, particularly when combined with the OCaml module and type system.

CCS Concepts: • Software and its engineering → *Software system structures; Functional languages; Computer systems organization* → *Cloud computing*.

Virtualisation and Networking

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure

Functional Networking for Millions of Docker Desktops (Experience Report)

ANIL MADHAVAPEDDY, University of Cambridge, United Kingdom

DAVID J. SCOTT, Docker, Inc., United Kingdom

PATRICK FERRIS, University of Cambridge, United Kingdom

RYAN T. GIBB, University of Cambridge, United Kingdom

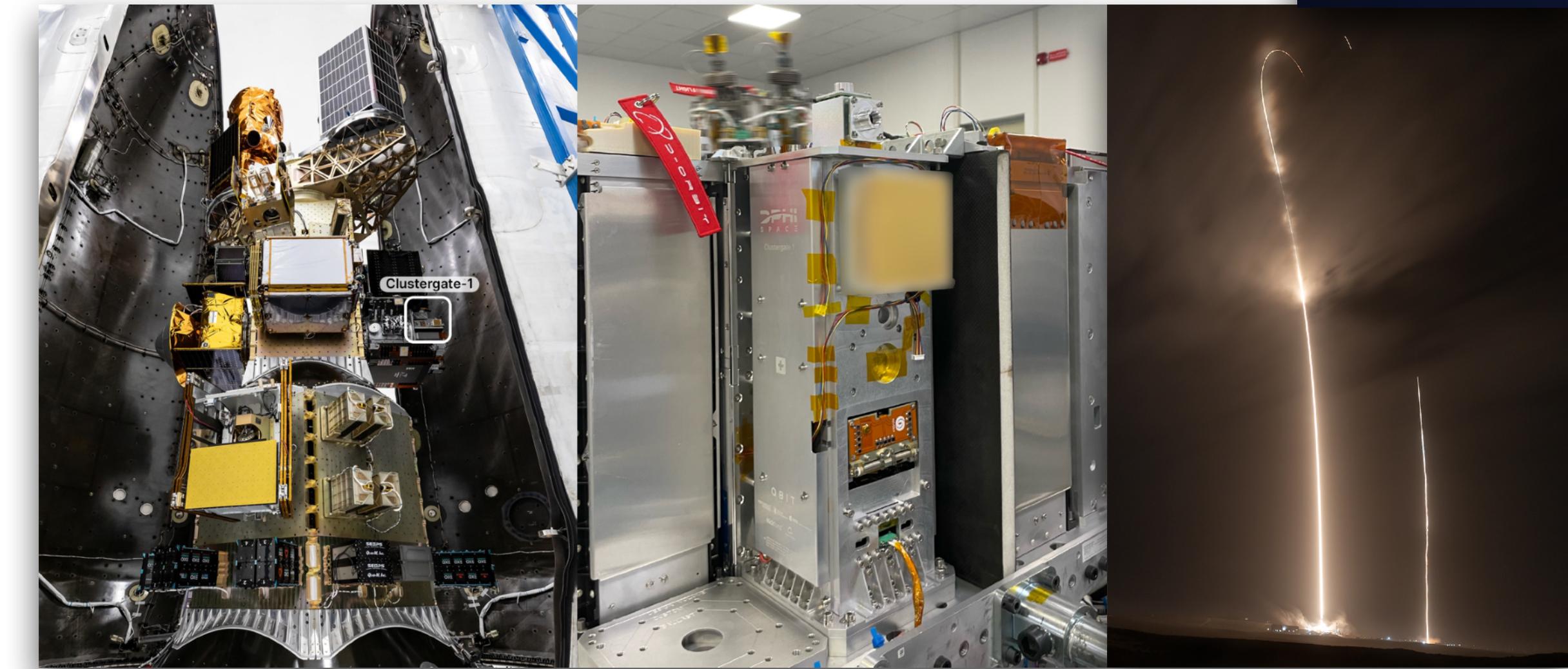
THOMAS GAZAGNAIRE, Tarides, France



Docker is a developer tool used by millions of developers to build, share and run software stacks. The Docker Desktop clients for Mac and Windows have long used a novel combination of virtualisation and OCaml unikernels to **seamlessly run Linux containers on these non-Linux hosts**. We reflect on a decade of shipping this functional OCaml code into production across hundreds of millions of developer desktops, and discuss the lessons learnt from our experiences in integrating OCaml deeply into the container architecture that now drives much of the global cloud. We conclude by observing just how good a fit for systems programming that the unikernel approach has been, particularly when combined with the OCaml module and type system.

CCS Concepts: • Software and its engineering → *Software system structures; Functional languages; Computer systems organization* → *Cloud computing*.

OCaml in Space



Virtualisation and Networking

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure



60+M lines of
OCaml code!

Bloomberg

Finance

High dynamic range

From scripts to scalable systems, research prototypes to production infrastructure

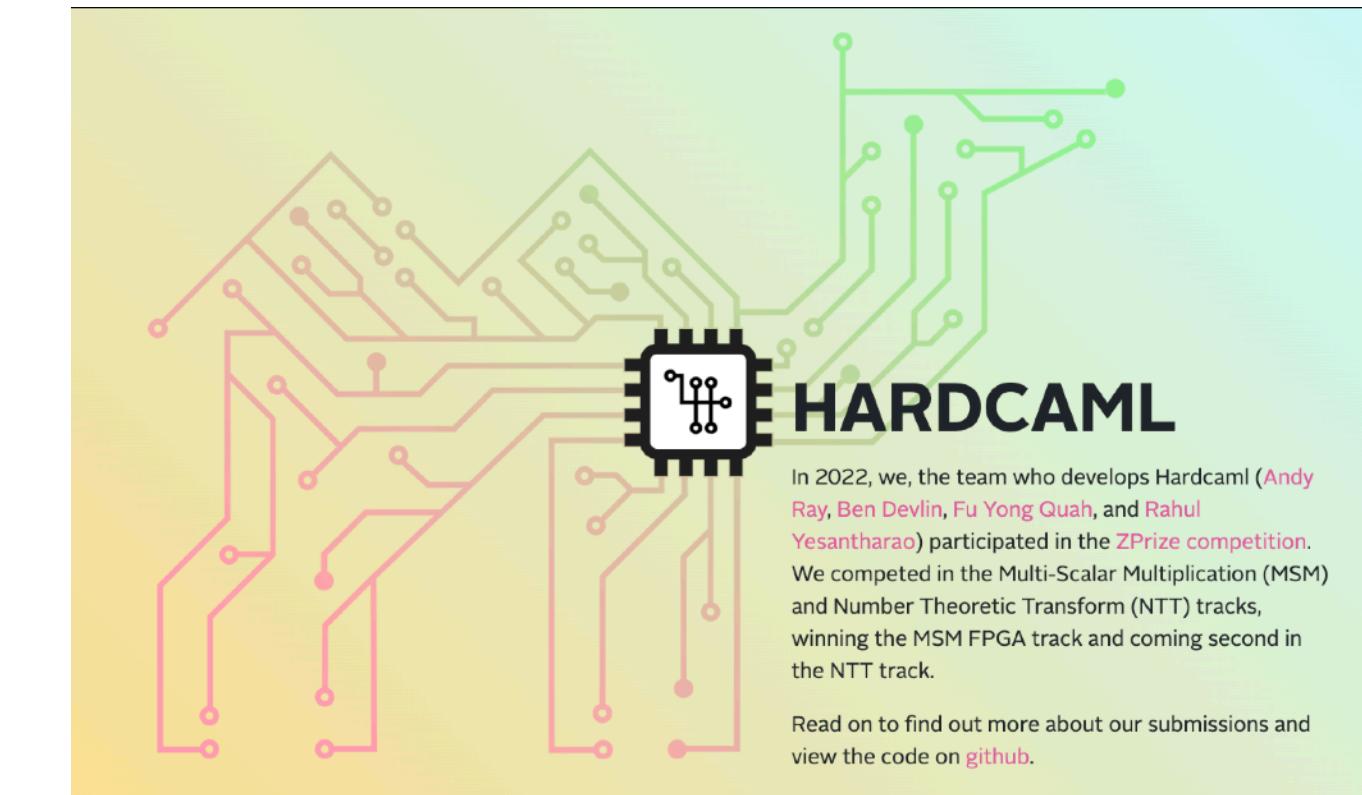


Jane
Street

Bloomberg

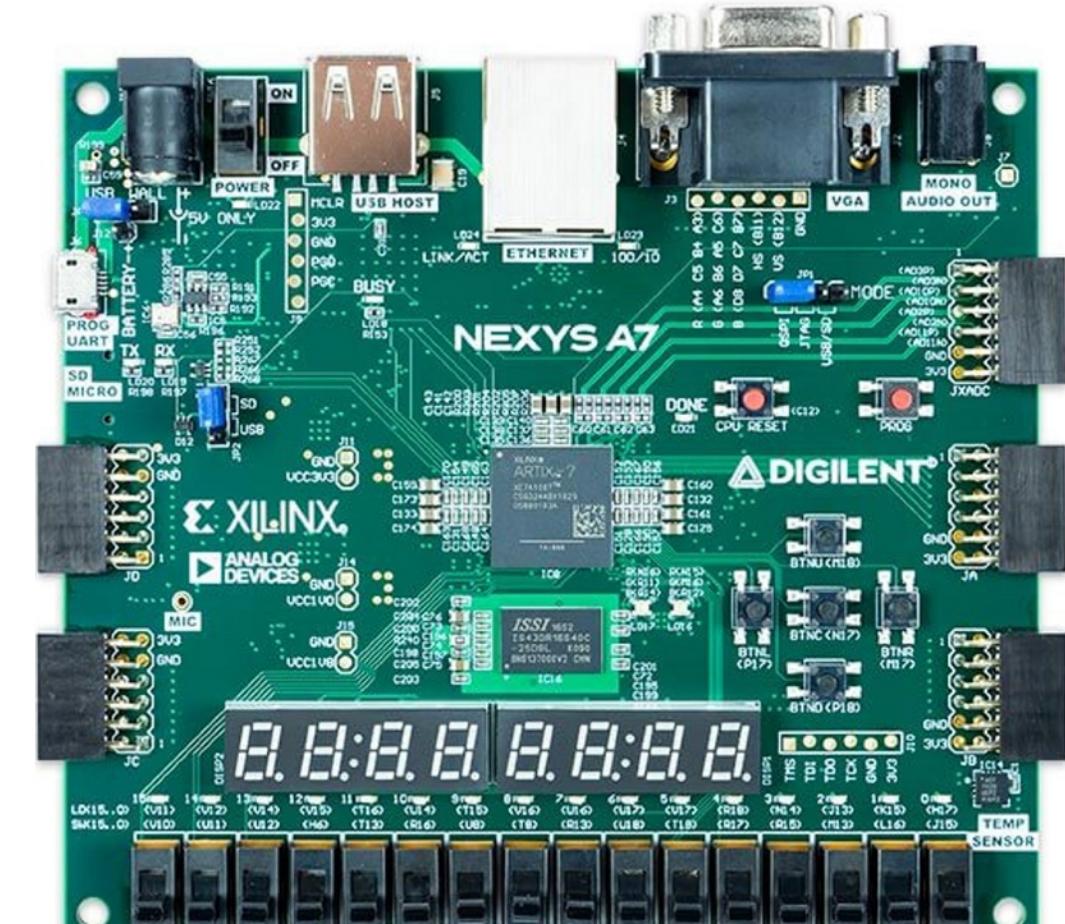
Finance

60+M lines of
OCaml code!



ZPRIZE

*Hardware
design*







29 years old!



29 years old!





29 years old!



1996 – OCaml 1.0

*Object system, low-latency GC, fast
native backend, module system*

1973 – Robin Milner’s “ML” for LCF

Type system, type inference



1985 – Guy Cousineau & co’s CAML

Categorical abstract machine (CAM) as IR



1996 – OCaml 1.0

Object system, low-latency GC, fast native backend, module system

1973 – Robin Milner’s “ML” for LCF

Type system, type inference



1985 – Guy Cousineau & co’s CAML

Categorical abstract machine (CAM) as IR



1996 – OCaml 1.0

Object system, low-latency GC, fast native backend, module system



2012 – OCaml 4.0

Generalized Algebraic Data types (GADTs)

1973 – Robin Milner’s “ML” for LCF

Type system, type inference



1985 – Guy Cousineau & co’s CAML

Categorical abstract machine (CAM) as IR



1996 – OCaml 1.0

Object system, low-latency GC, fast native backend, module system



2012 – OCaml 4.0

Generalized Algebraic Data types (GADTs)



2022 – OCaml 5.0

Multicore parallelism, effect handlers

Steady evolution
over **50+** years

1973 – Robin Milner’s “ML” for LCF

Type system, type inference



1985 – Guy Cousineau & co’s CAML

Categorical abstract machine (CAM) as IR



1996 – OCaml 1.0

Object system, low-latency GC, fast native backend, module system



2012 – OCaml 4.0

Generalized Algebraic Data types (GADTs)



2022 – OCaml 5.0

Multicore parallelism, effect handlers



2025

1973 – Robin Milner’s “ML” for LCF

Type system, type inference



1985 – Guy Cousineau & co’s CAML

Categorical abstract machine (CAM) as IR



1996 – OCaml 1.0

Object system, low-latency GC, fast native backend, module system



2012 – OCaml 4.0

Generalized Algebraic Data types (GADTs)



2022 – OCaml 5.0

Multicore parallelism, effect handlers



Steady evolution
over **50+** years

**How to *thrive* not just
survive after ~30 years?**

Simplicity and *stability*

Simplicity and *stability*

Xavier Leroy, 2023 SIGPLAN programming languages software award! 

What made that possible? Not just fancy types and nice modules – even though systems programmers value type safety and modularity highly – but also basic properties of OCaml:

- a language with a simple cost model, where it's easy to track how much time and how much space is used;
- a compiler that produces efficient code that looks like the source code, with only predictable optimizations;
- a low-latency garbage collector, usable for soft real-time applications.

Simplicity and *stability*

Xavier Leroy, 2023 SIGPLAN programming languages software award! 

What made that possible? Not just fancy types and nice modules – even though systems programmers value type safety and modularity highly – but also basic properties of OCaml:

- a language with a simple cost model, where it's easy to track how much time and how much space is used;
- a compiler that produces efficient code that looks like the source code, with only predictable optimizations;
- a low-latency garbage collector, usable for soft real-time applications.

Simplicity and *stability*

Xavier Leroy, 2023 SIGPLAN programming languages software award! 

What made that possible? Not just fancy types and nice modules – even though systems programmers value type safety and modularity highly – but also basic properties of OCaml:

- a language with a simple cost model, where it's easy to track how much time and how much space is used;
- a compiler that produces efficient code that looks like the source code, with only predictable optimizations;
- a low-latency garbage collector, usable for soft real-time applications.

Simplicity and *stability*

Xavier Leroy, 2023 SIGPLAN programming languages software award! 

What made that possible? Not just fancy types and nice modules – even though systems programmers value type safety and modularity highly – but also basic properties of OCaml:

- a language with a simple cost model, where it's easy to track how much time and how much space is used;
- a compiler that produces efficient code that looks like the source code, with only predictable optimizations;
- a low-latency garbage collector, usable for soft real-time applications.

- If you take OCaml from 20 years ago, the code will likely **continue to work!**

Simplicity and *stability*

Xavier Leroy, 2023 SIGPLAN programming languages software award! 

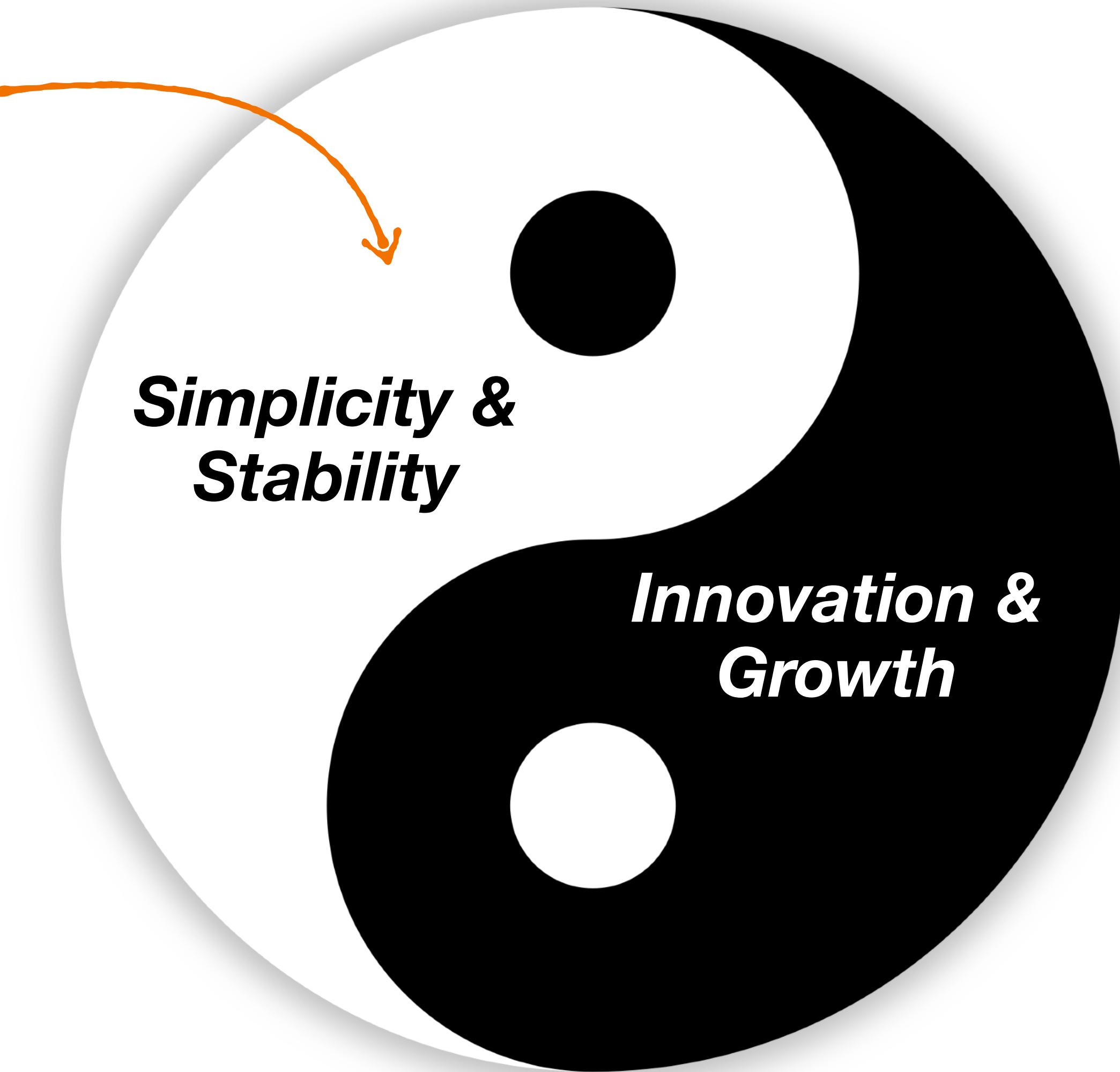
What made that possible? Not just fancy types and nice modules – even though systems programmers value type safety and modularity highly – but also basic properties of OCaml:

- a language with a simple cost model, where it's easy to track how much time and how much space is used;
- a compiler that produces efficient code that looks like the source code, with only predictable optimizations;
- a low-latency garbage collector, usable for soft real-time applications.

- If you take OCaml from 20 years ago, the code will likely **continue to work!**
- No recent releases for some popular packages
 - They are **good enough**, and continue to be so.
 - Nothing to be done to keep it working!



OCaml



OCaml Maintainers

Abigael

Alain Frisch

Armaël Guéneau

Anil Madhavapeddy

Pierre Chambart

Damien Doligez

David Allsopp

Jacques Garrigue

Gabriel Scherer

Richard Eisenberg

Jacques-Henri Jourdan

KC Sivaramakrishnan

Frédéric Bour

Leo White

Vincent Lviron

Luc Maranget

Mark Shinwell

Nick Barnes

Nicolás Ojeda Bär

Florian Angeletti

Olivier Nicole

Sadiq Jaffer

Sébastien Hinderer

Stephen Dolan

Thomas Refis

Xavier Leroy

Jeremy Yallop

- 27 maintainers from France, UK, Japan, India and USA, across industry and academia.

OCaml Maintainers

Abigael

Alain Frisch

Armaël Guéneau

Anil Madhavapeddy

Pierre Chambart

Damien Doligez

David Allsopp

Jacques Garrigue

Gabriel Scherer

Richard Eisenberg

Jacques-Henri Jourdan

KC Sivaramakrishnan

Frédéric Bour

Leo White

Vincent Lviron

Luc Maranget

Mark Shinwell

Nick Barnes

Nicolás Ojeda Bär

Florian Angeletti

Olivier Nicole

Sadiq Jaffer

Sébastien Hinderer

Stephen Dolan

Thomas Refis

Xavier Leroy

Jeremy Yallop

- 27 maintainers from France, UK, Japan, India and USA, across industry and academia.
- Custodians of the compiler
 - *Not the ones deciding how the language should evolve!*

Who decides how OCaml evolves?

Who decides how OCaml evolves?



Who decides how OCaml evolves?

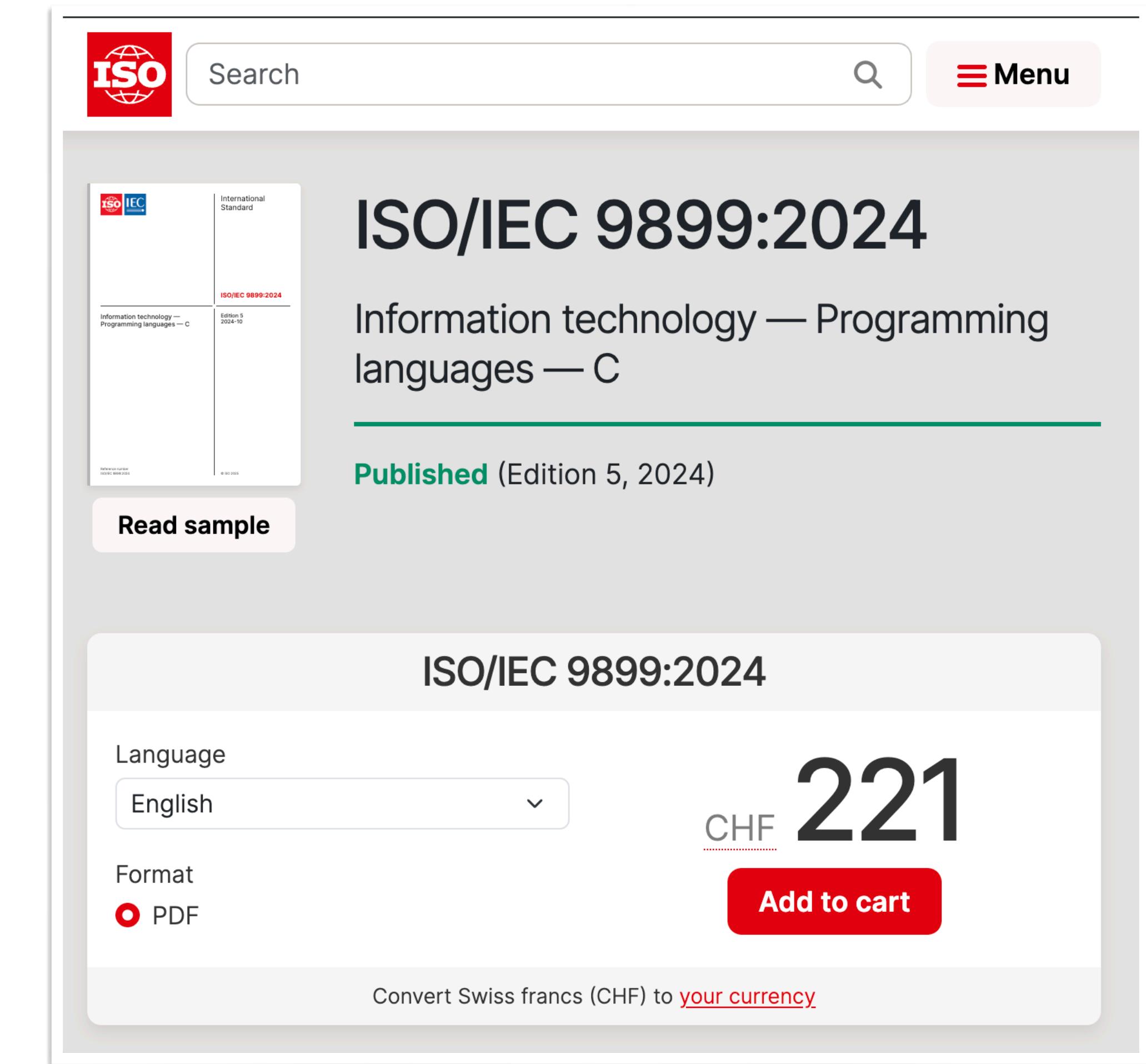
- Evolution
 - **User-driven:** OCaml, Python
 - **Committee-driven:** ISO/IEC evolving C and C++
 - **Vendor-driven consensus:** WebAssembly

Who decides how OCaml evolves?

- Evolution
 - **User-driven:** OCaml, Python
 - **Committee-driven:** ISO/IEC evolving C and C++
 - **Vendor-driven consensus:** WebAssembly
- **Language** and **compiler** aren't distinct
 - OCaml compiler implementation **IS** the language.

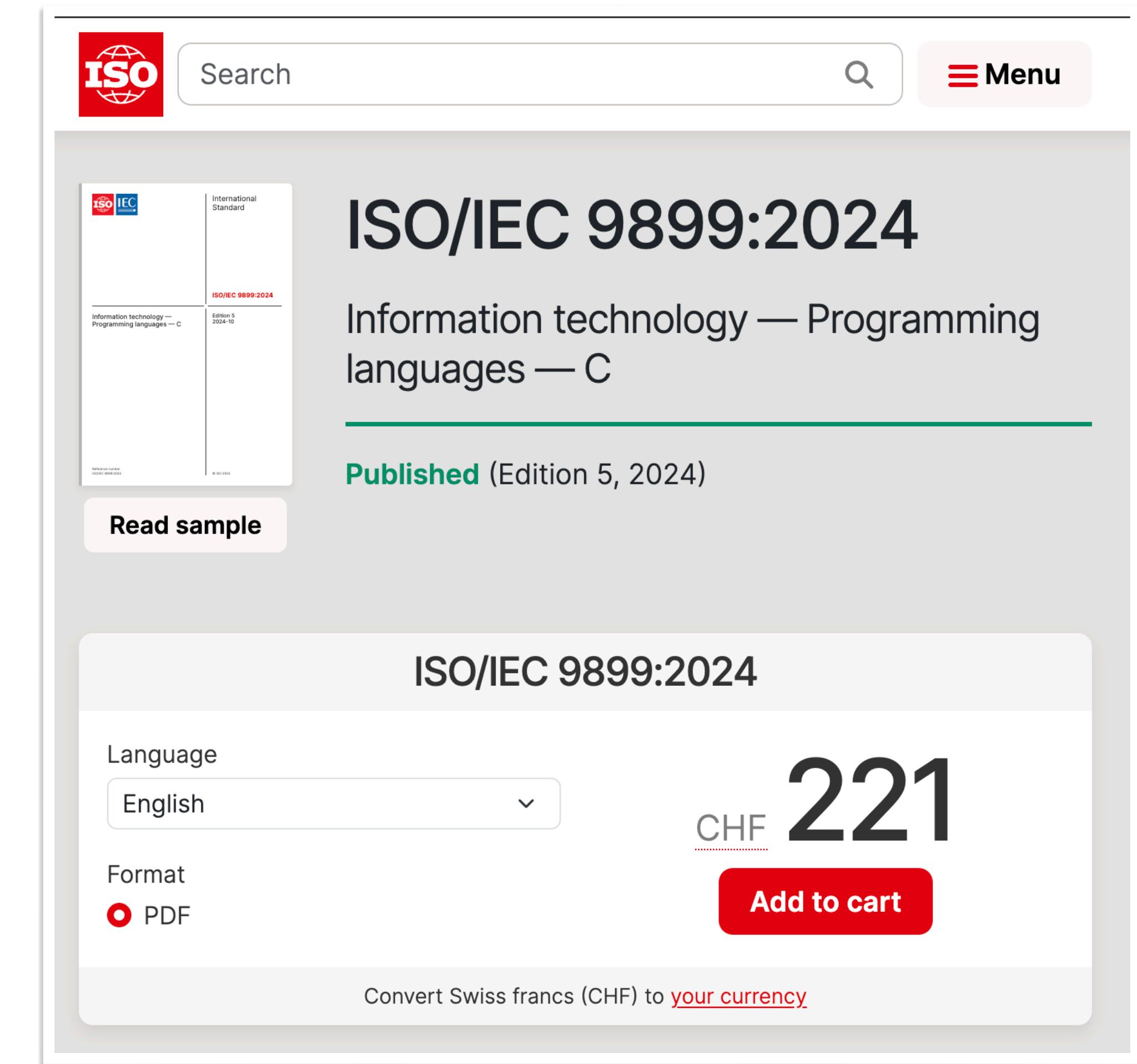
Who decides how OCaml evolves?

- Evolution
 - **User-driven:** OCaml, Python
 - **Committee-driven:** ISO/IEC evolving C and C++
 - **Vendor-driven consensus:** WebAssembly
- **Language and compiler** aren't distinct
 - OCaml compiler implementation *IS* the language.
- Unlike C, Wasm, JavaScript



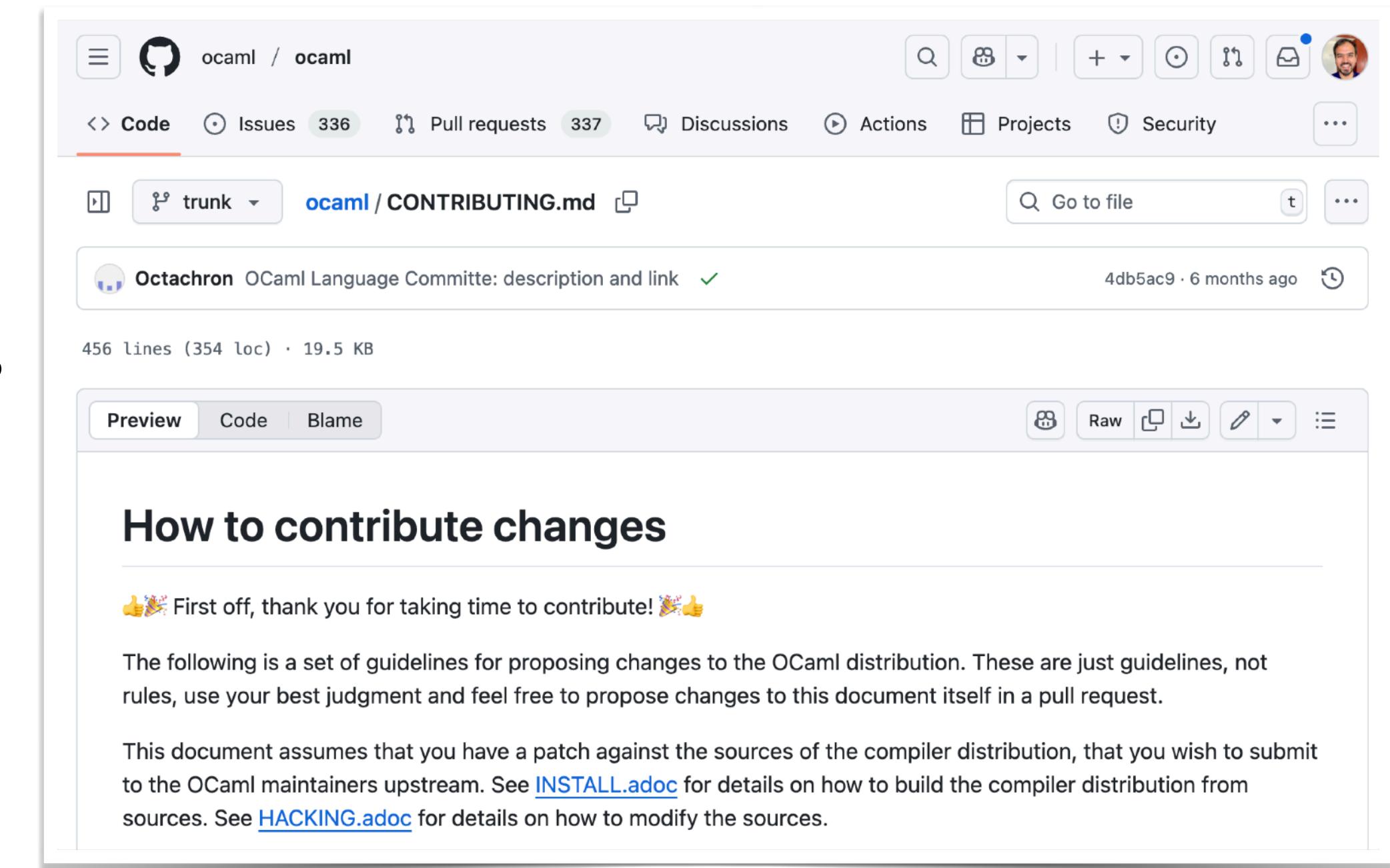
Who decides how OCaml evolves?

- Evolution
 - **User-driven:** OCaml, Python
 - **Committee-driven:** ISO/IEC evolving C and C++
 - **Vendor-driven consensus:** WebAssembly
- **Language and compiler** aren't distinct
 - OCaml compiler implementation **IS** the language.
- Unlike C, Wasm, JavaScript
- *The bar is lower to change the language*



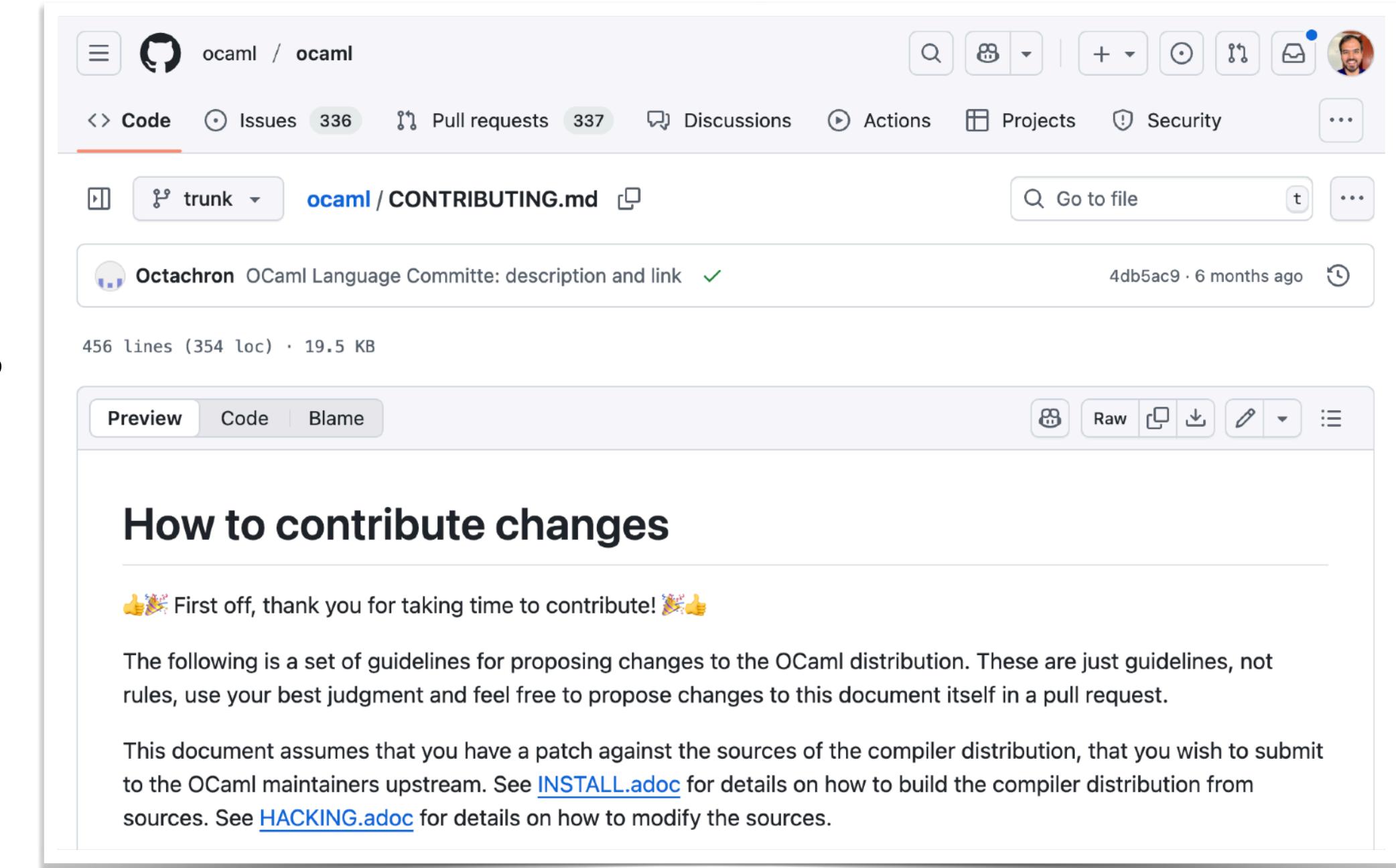
Mechanics of evolution

- Open process
 - OCaml compiler is maintained on GitHub
 - All discussions are public in the PRs, Issues and RFCs on GitHub



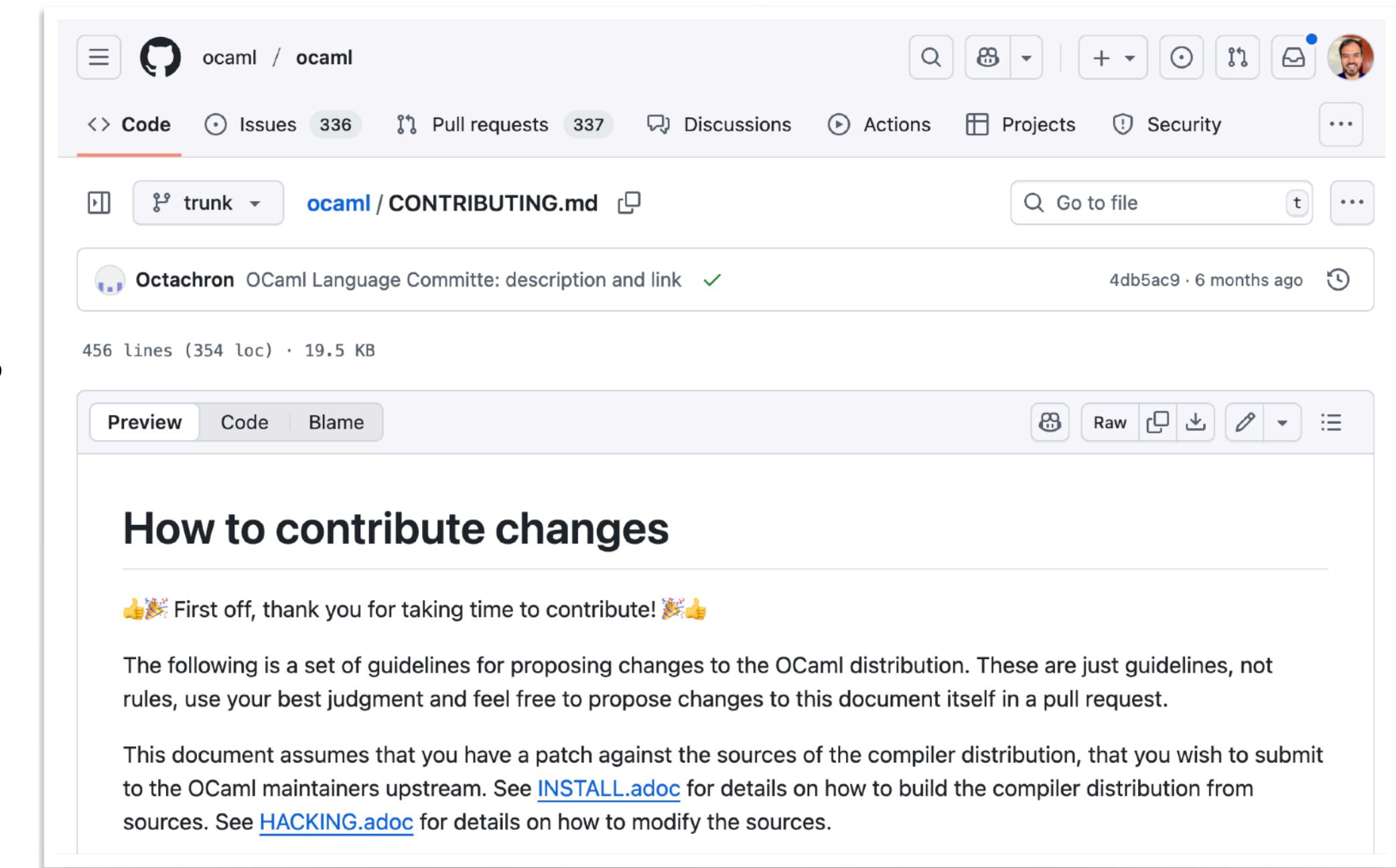
Mechanics of evolution

- Open process
 - OCaml compiler is maintained on GitHub
 - All discussions are public in the PRs, Issues and RFCs on GitHub
- Multi-speed model



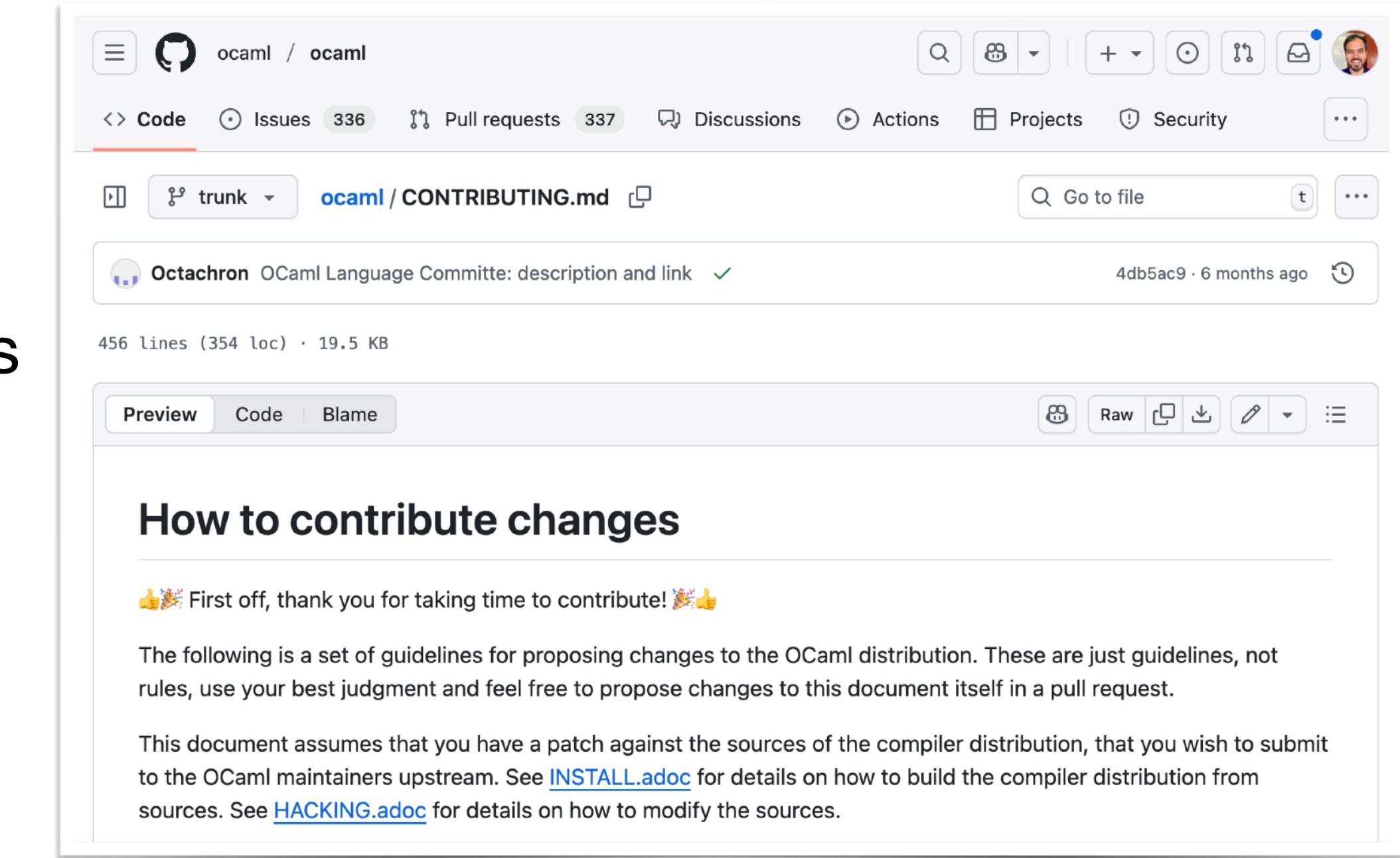
Mechanics of evolution

- Open process
 - OCaml compiler is maintained on GitHub
 - All discussions are public in the PRs, Issues and RFCs on GitHub
- Multi-speed model
 - **Small fixes/features** → Make an issue (“feature request”), open a PR, discuss and get that merged
 - Every PR needs a maintainer's approval before merging



Mechanics of evolution

- Open process
 - OCaml compiler is maintained on GitHub
 - All discussions are public in the PRs, Issues and RFCs on GitHub
- Multi-speed model
 - **Small fixes/features** → Make an issue (“feature request”), open a PR, discuss and get that merged
 - Every PR needs a maintainer's approval before merging
 - **Large features** → Bespoke based on the features
 - May need publishing papers, extensive performance evaluation, formalised/mechanised soundness results, etc.
- ***Often, presumably small feature requests take a life of their own!***



A small(?) change – Dynamic Arrays

Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk`

Conversation 12 Commits 1 Checks 0 Files changed 7

Mathilde411 commented on Nov 15, 2019

Created DynArray Module which implements amortized time complexity dynamic arrays.

32b86a8 Assignees

Opened: Nov 15, 2019, Closed: Nov 15 2019

Implementation rather naive, room for improvements

A small(?) change – Dynamic Arrays

Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk` 

Conversation 12 Commits 1 Checks 0 Files changed 7 +198 -5

add Dynarray to the stdlib. #11563

Closed c-cube wants to merge 29 commits into `ocaml:trunk` from `c-cube:dynamic-array` 

Conversation 92 Commits 29 Checks 0 Files changed 10 +792 -1

 c-cube commented on Sep 25, 2022 Member ...

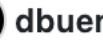
Overview

This is a (work in progress) PR to add dynamic arrays ("vectors") to the stdlib. The module name is `Dyn_array`, which, as some people pointed out, is more correct than `vector`. For now the implementation is pure OCaml. I discussed with [@Octachron](#) about ways to implement some filling functions in C, but I now think it might not be worth it after he pointed out some design constraints newly imposed by multicore.

A lot of the API mimics `Array`, when it does not change the length of the dynamic array.

Reviewers

 **gasche**
+5 more reviewers

 **bluddy**
 **dbuenzli**
 **hhugo**
 **silene**
 **gadmm**

Assignees

No one—[assign yourself](#)

Opened: Nov 15, 2019, Closed: Nov 15 2019

Implementation rather naive, room for improvements

Opened: Sep 25, 2022, Closed: Jan 18, 2023

Clean API, *but* multicore safety, performance

A small(?) change – Dynamic Arrays

Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk`

Conversation 12 Commits 1 Checks 0 Files changed 7 +198 -5

add Dynarray to the stdlib. #11563

Closed c-cube wants to merge 29 commits into `ocaml:trunk` from `c-cube:dyn-array`

Conversation 92 Commits 29 Checks 0 Files changed 10 +792 -1

Dynarrays, boxed #11882

Merged gasche merged 51 commits into `ocaml:trunk` from `gasche:dyn-array-boxed` on Oct 21, 2023

Conversation 342 Commits 51 Checks 0 Files changed 18 +2,108 -173

gasche commented on Jan 11, 2023 · edited

Current status of this PR

(last updated: September 27th 2023)

Reviewers

- alainfrisch
- damiendoligez
- Octachron
- +6 more reviewers
- dbuenzli

Opened: Nov 15, 2019, Closed: Nov 15 2019

Implementation rather naive, room for improvements

Opened: Sep 25, 2022, Closed: Jan 18, 2023

Clean API, **but** multicore safety, performance

Opened: Jan 11, 2023, Merged: Oct 21, 2023

Clean API **and** simple implementation

A small(?) change – Dynamic Arrays

Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk`

Conversation 12 Commits 1 Checks 0 Files changed 7 +198 -5

add Dynarray to the stdlib. #11563

Closed c-cube wants to merge 29 commits into `ocaml:trunk` from `c-cube:dyn-array`

Conversation 92 Commits 29 Checks 0 Files changed 10 +792 -1

Dynarrays, boxed #11882

Merged gasche merged 51 commits into `ocaml:trunk` from `gasche:dyn-array-boxed` on Oct 21, 2023

Conversation 92 Commits 29 Checks 0 Files changed 10 +792 -1

Dynarrays, unboxed (with local dummies) #12885

Merged gasche merged 9 commits into `ocaml:trunk` from `gasche:dynarray-unboxed-dummy` on May 2, 2024

Conversation 53 Commits 9 Checks 0 Files changed 6 +455 -235

gasche commented on Jan 5, 2024 · edited

Member ... Reviewers

Opened: Nov 15, 2019, Closed: Nov 15 2019

Implementation rather naive, room for improvements

Opened: Sep 25, 2022, Closed: Jan 18, 2023

Clean API, **but** multicore safety, performance

Opened: Jan 11, 2023, Merged: Oct 21, 2023

Clean API **and** simple implementation

Opened: Jan 5, 2024, Merged: May 2, 2024

Clean API and **optimised** implementation

Dynamic Arrays

Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk`

Conversation 12, Commits 1, Checks 0, Files changed 7, +198 -5

Mathilde411 commented on Nov 15, 2019

Created DynArray Module which implements amortized time complexity dynamic arrays.

Octachron, +2 more reviewers, Drup, thomasblanc

Dynarrays, unboxed (with local dummies) #12885

Merged gasche merged 9 commits into `ocaml:trunk` from `gasche:dynamic-unboxed-dummy` on May 2, 2024

Conversation 53, Commits 9, Checks 0, Files changed 6, +455 -235

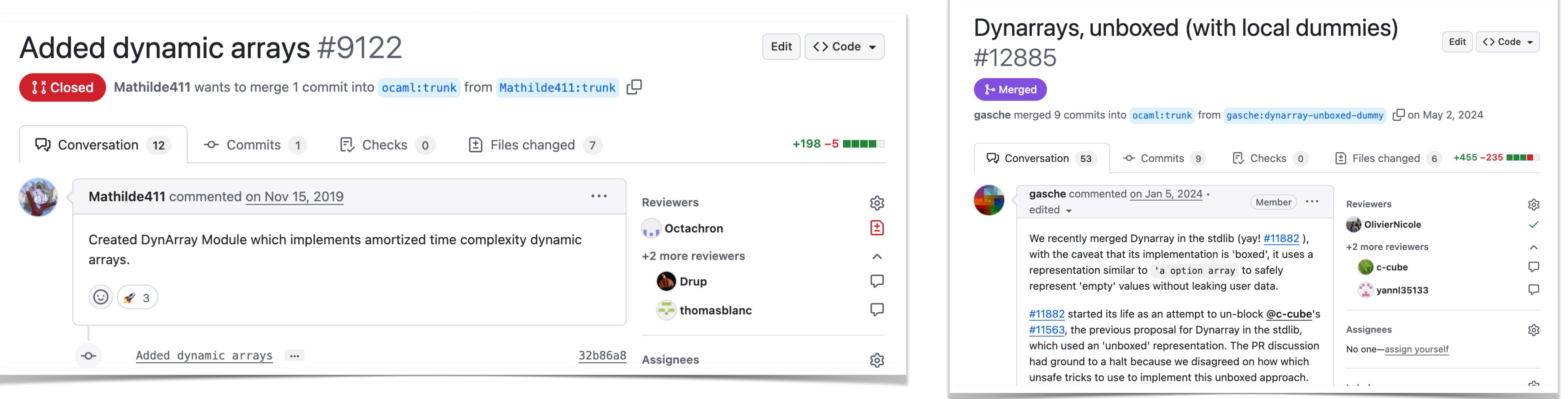
gasche commented on Jan 5, 2024

We recently merged Dynarray in the stdlib (yay! #11882), with the caveat that its implementation is 'boxed', it uses a representation similar to 'a option array to safely represent 'empty' values without leaking user data.

#11882 started its life as an attempt to un-block @c-cube's #11563, the previous proposal for Dynarray in the stdlib, which used an 'unboxed' representation. The PR discussion had ground to a halt because we disagreed on how which unsafe tricks to use to implement this unboxed approach.

- **Summary**
 - Proposed – Nov 2019, Merged – (PR#1) Jan 2024; (PR#2) May 2024
 - Initially – 198 loc, finally – ~2500 loc
 - 500+ comments in the various PRs

Dynamic Arrays



Added dynamic arrays #9122

Closed Mathilde411 wants to merge 1 commit into `ocaml:trunk` from `Mathilde411:trunk`

Conversation 12, Commits 1, Checks 0, Files changed 7, +198 -5

Mathilde411 commented on Nov 15, 2019

Created DynArray Module which implements amortized time complexity dynamic arrays.

Octachron, +2 more reviewers, Drup, thomasblanc

Dynarrays, unboxed (with local dummies) #12885

Merged gasche merged 9 commits into `ocaml:trunk` from `gasche:dynamic-unboxed-dummy` on May 2, 2024

Conversation 53, Commits 9, Checks 0, Files changed 6, +455 -235

gasche commented on Jan 5, 2024

We recently merged Dynarray in the stdlib (yay! #11882), with the caveat that its implementation is 'boxed', it uses a representation similar to 'a option array to safely represent 'empty' values without leaking user data.

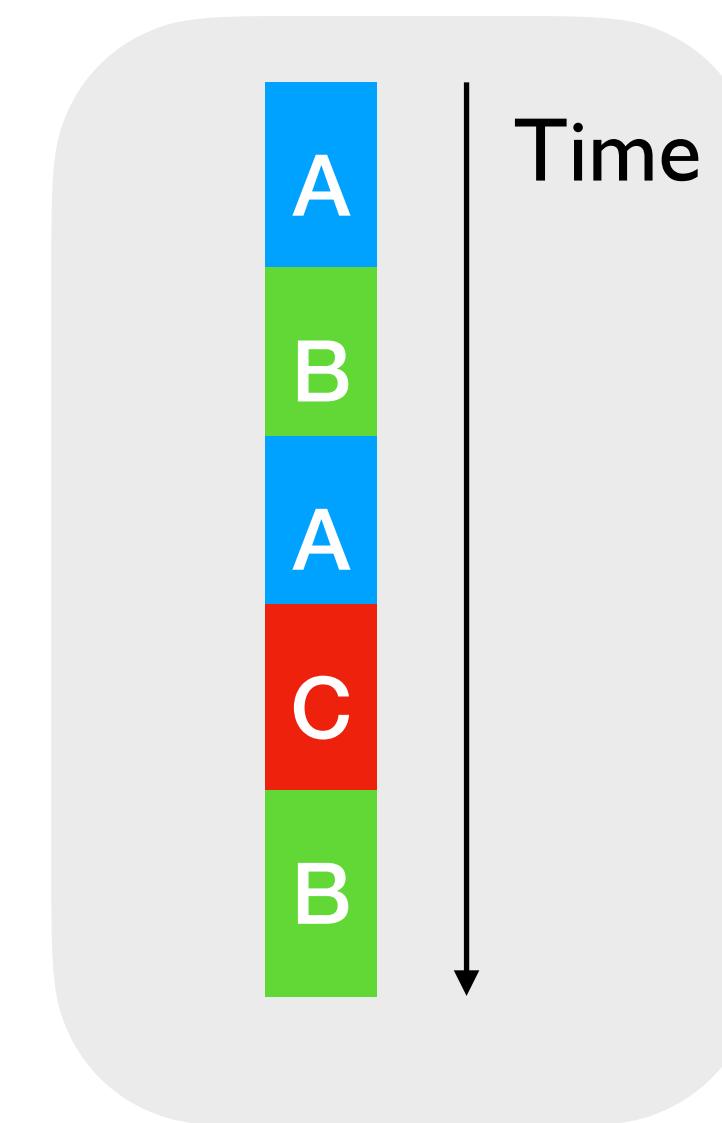
#11882 started its life as an attempt to un-block @c-cube's #11563, the previous proposal for Dynarray in the stdlib, which used an 'unboxed' representation. The PR discussion had ground to a halt because we disagreed on how which unsafe tricks to use to implement this unboxed approach.

- **Summary**
 - Proposed – Nov 2019, Merged – (PR#1) Jan 2024; (PR#2) May 2024
 - Initially – 198 loc, finally – ~2500 loc
 - 500+ comments in the various PRs
- **Worth it?**
 - **Yes!** Should work for the next couple of decades.
 - Harder to undo changes after the release.

A large change – Multicore OCaml

- Native support for concurrency and parallelism to OCaml

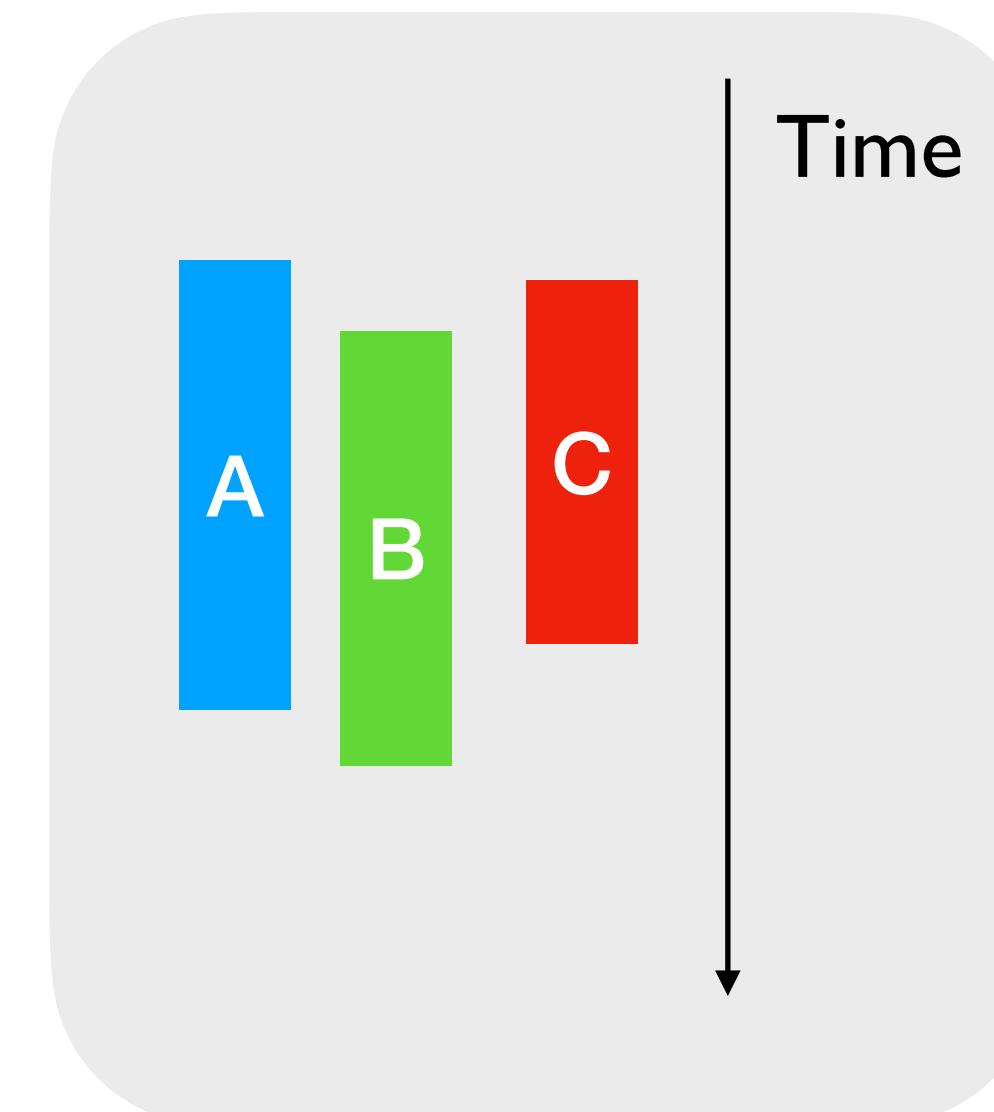
Concurrency



Interleaved execution

Effect Handlers

Parallelism



Simultaneous execution

Domains

Challenges

Challenges

- A new multicore garbage collector and multicore runtime system
 - *Replacing a car engine with a new one!*

Challenges

- A new multicore garbage collector and multicore runtime system
 - *Replacing a car engine with a new one!*
- Make the language itself thread-safe
 - OCaml is a safe language! (Unlike C/C++, Go)

Challenges

- A new multicore garbage collector and multicore runtime system
 - *Replacing a car engine with a new one!*
- Make the language itself thread-safe
 - OCaml is a safe language! (Unlike C/C++, Go)
- Maintain feature and performance backwards compatibility!
 - Most OCaml programs will continue to remain single-threaded

Challenges

- A new multicore garbage collector and multicore runtime system
 - *Replacing a car engine with a new one!*
- Make the language itself thread-safe
 - OCaml is a safe language! (Unlike C/C++, Go)
- Maintain feature and performance backwards compatibility!
 - Most OCaml programs will continue to remain single-threaded

Build credibility by ***publishing key results*** and ***rigorous evaluation***

Starting out

Multicore OCaml

Stephen Dolan

Leo White

Anil Madhavapeddy

Currently, threading is supported in OCaml only by means of a global lock, allowing at most one 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

Starting out

Upstream
OCaml



Multicore OCaml

Stephen Dolan

Leo White

Anil Madhavapeddy

Currently, threading is supported in OCaml only by means of a global lock, allowing at most one 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

OCaml Workshop 2014

Starting out

Multicore OCaml

Stephen Dolan Leo White Anil Madhavapeddy

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

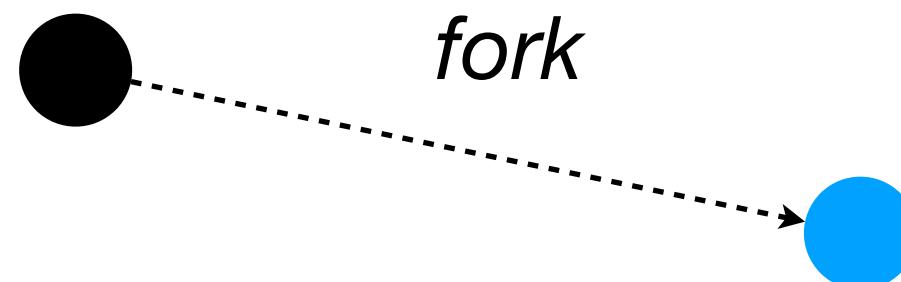
1 Introduction

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

Upstream
OCaml



Multicore
OCaml

Starting out

Multicore OCaml

Stephen Dolan

Leo White

Anil Madhavapeddy

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

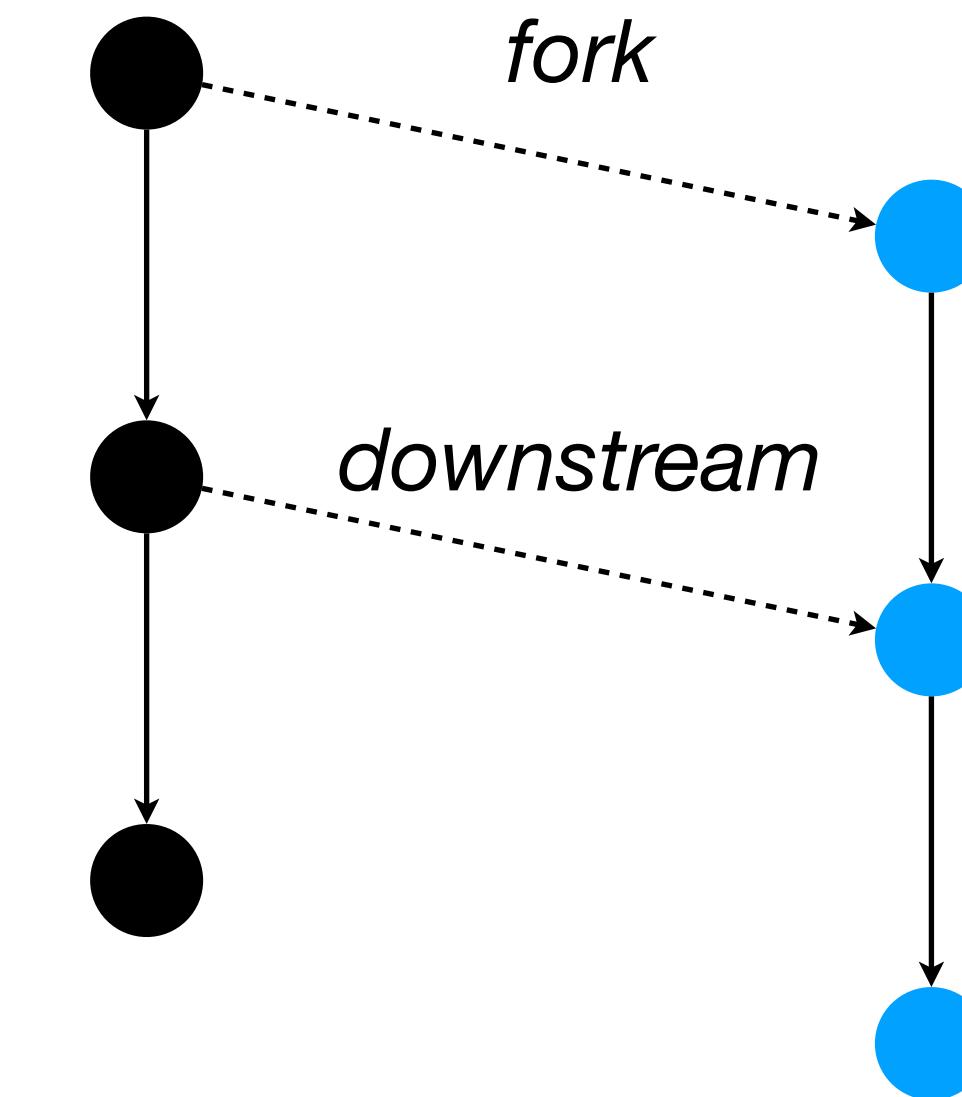
1 Introduction

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

Upstream OCaml



Multicore OCaml

Starting out

Multicore OCaml

Stephen Dolan

Leo White

Anil Madhavapeddy

Currently, threading is supported in OCaml only by means of a global lock, allowing at most one 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-

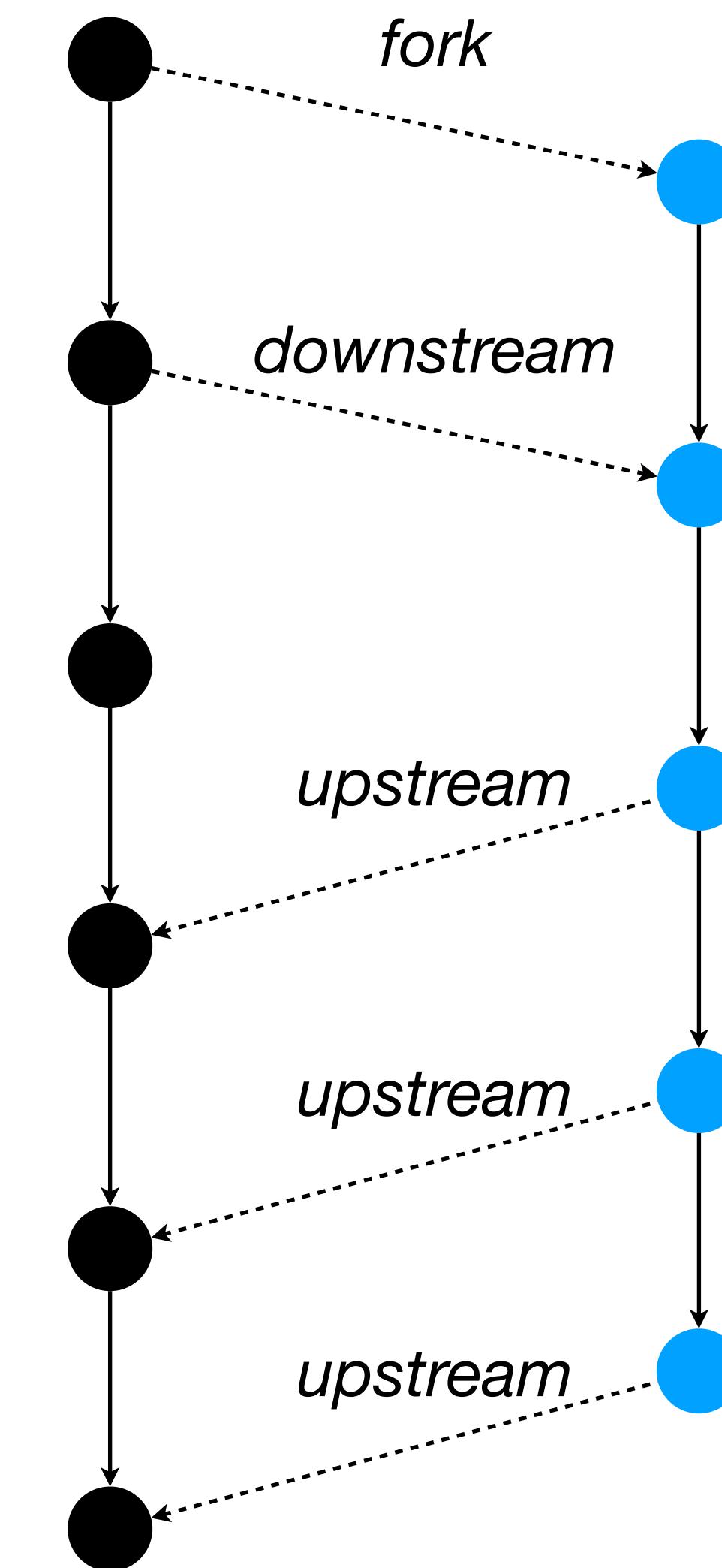
all objects reachable from it to be promoted to the shared heap en masse. Unfortunately this eagerly promotes many objects that were never really shared: just because an object is pointed to by a shared object does not mean another thread is actually going to attempt to access it.

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

OCaml Workshop 2014

Upstream OCaml

Multicore OCaml



Building confidence – Papers

Multicore GC and runtime system

Retrofitting Parallelism onto OCaml

Bounding Data Races in Space and Time

(Extended version, with appendices

Retrofitting Effect Handlers onto OCaml

Stephen Dolan

OCaml Labs

Cambridge, UK

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

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

Relaxed Memory Model

Concurrency story

Abstract

We propose programs that of data race guarantees that memory parallel pro

KC Sivaramakrishnan

IIT Madras

Chennai, India

kcsrk@cse.iitm.ac.in

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

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

Diving deeper – Concurrency

Retrofitting Effect Handlers onto OCaml

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



Concurrent Programming

- Computations may be *suspended* and *resumed* later

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
 - ◆ **async/await** – JavaScript, Python, Rust, C# 5.0, F#, Swift, ...
 - ◆ **generators** – Python, Javascript, ...
 - ◆ **coroutines** – C++, Kotlin, Lua, ...
 - ◆ **futures & promises** – JavaScript, Swift, ...
 - ◆ **Lightweight threads/processes** – Haskell, Go, Erlang

Concurrent Programming

- Computations may be *suspended* and *resumed* later
- Many languages provide concurrent programming mechanisms as *primitives*
 - ◆ **async/await** – JavaScript, Python, Rust, C# 5.0, F#, Swift, ...
 - ◆ **generators** – Python, Javascript, ...
 - ◆ **coroutines** – C++, Kotlin, Lua, ...
 - ◆ **futures & promises** – JavaScript, Swift, ...
 - ◆ **Lightweight threads/processes** – Haskell, Go, Erlang
- *Often include many different primitives in the same language!*
 - ◆ JavaScript has async/await, generators, promises, and callbacks

Don't want a **zoo** of primitives but
want **expressivity**

Don't want a **zoo** of primitives but
want **expressivity**

What's the **smallest** primitive that
expresses **many** concurrency patterns?

Effect handlers

- A mechanism for programming with *user-defined effects*

Effect handlers

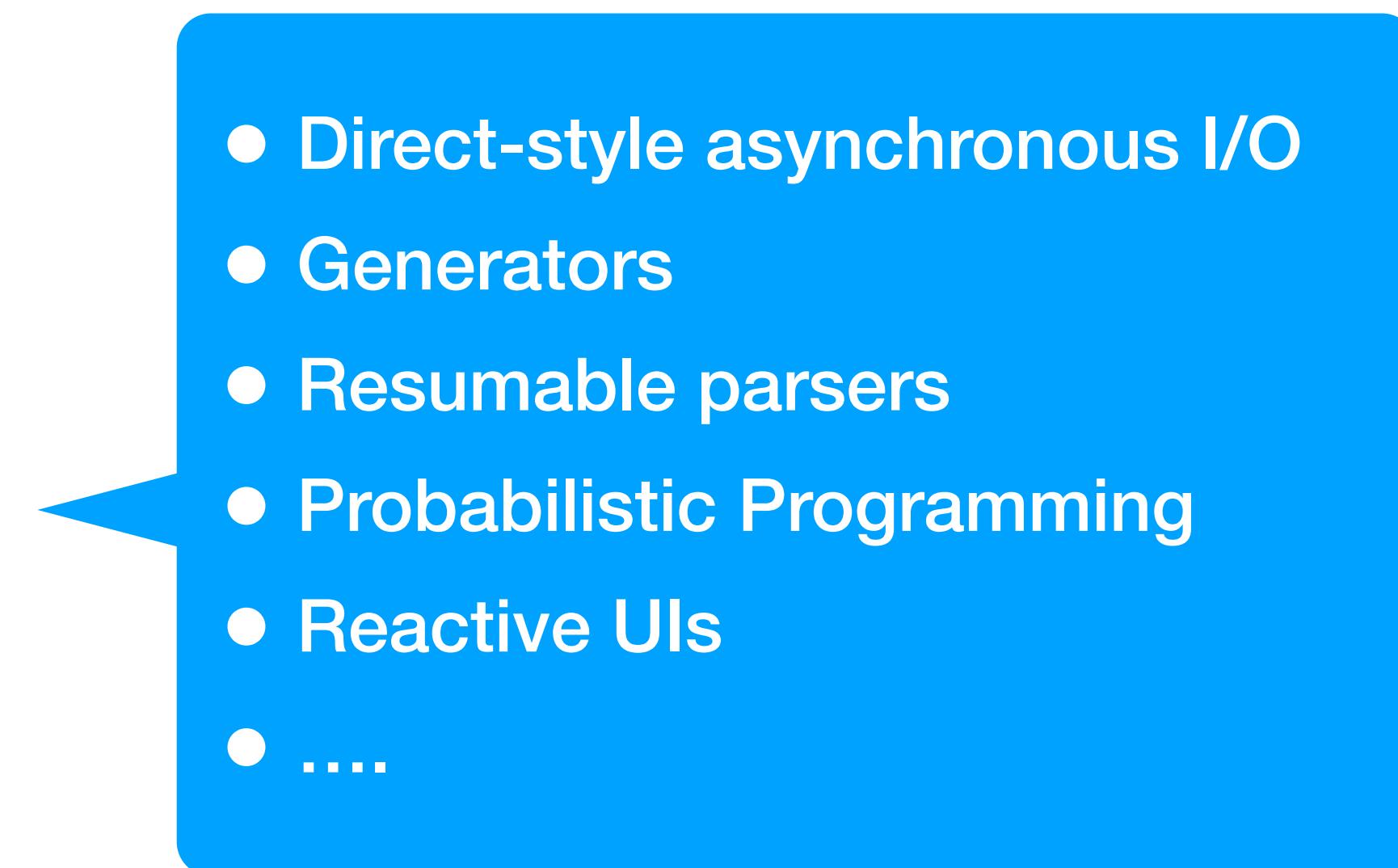
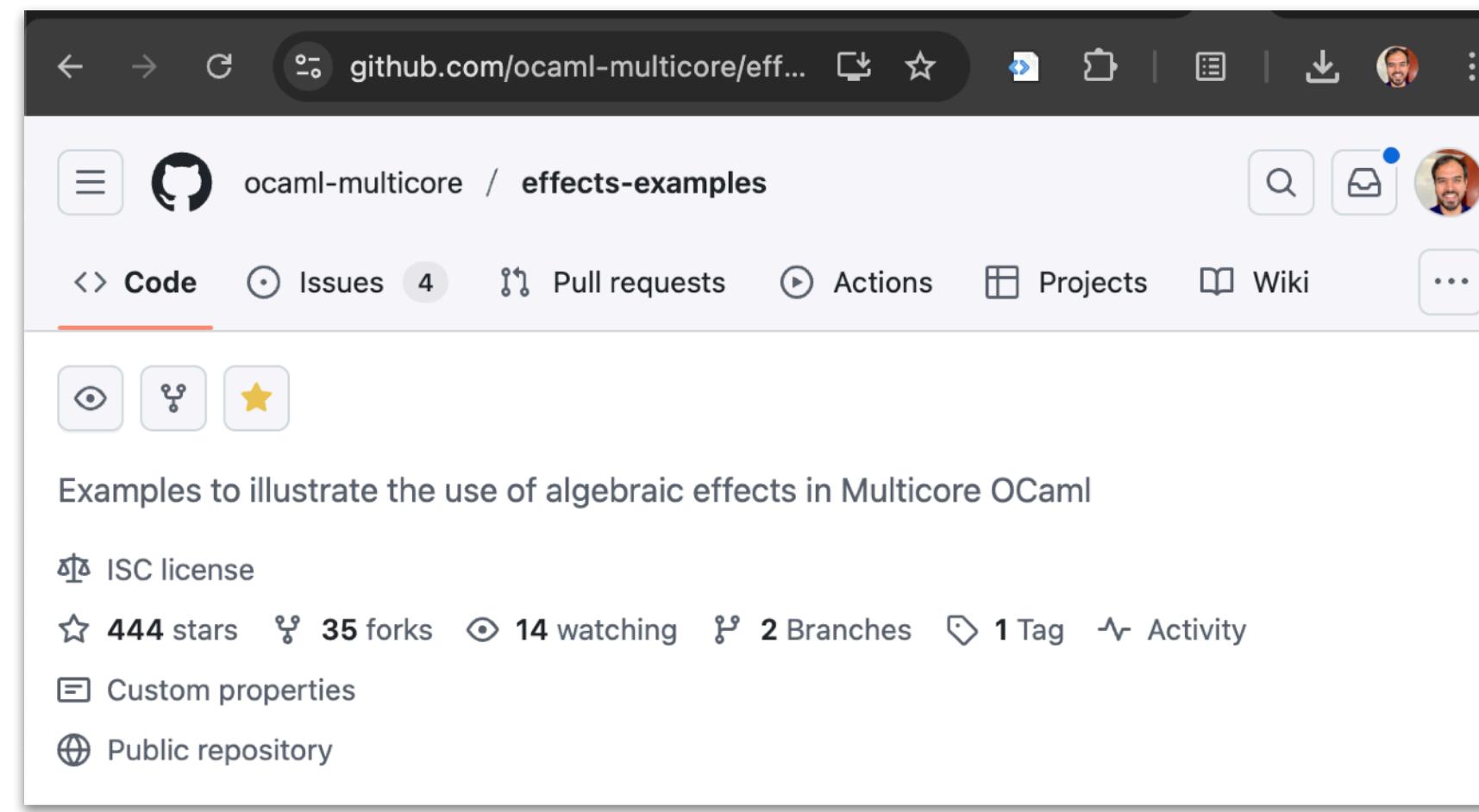
- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
 - ◆ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*

Effect handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
 - ◆ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*
- Effect handlers $\sim=$ *first-class, restartable exceptions*
 - ◆ Structured programming with *delimited continuations*

Effect handlers

- A mechanism for programming with *user-defined effects*
- *Modular* and *composable* basis of non-local control-flow mechanisms
 - ◆ Exceptions, generators, lightweight threads, promises, asynchronous IO, coroutines as *libraries*
- Effect handlers $\sim=$ *first-class, restartable exceptions*
 - ◆ Structured programming with *delimited continuations*



Effect handlers

```
type _ eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

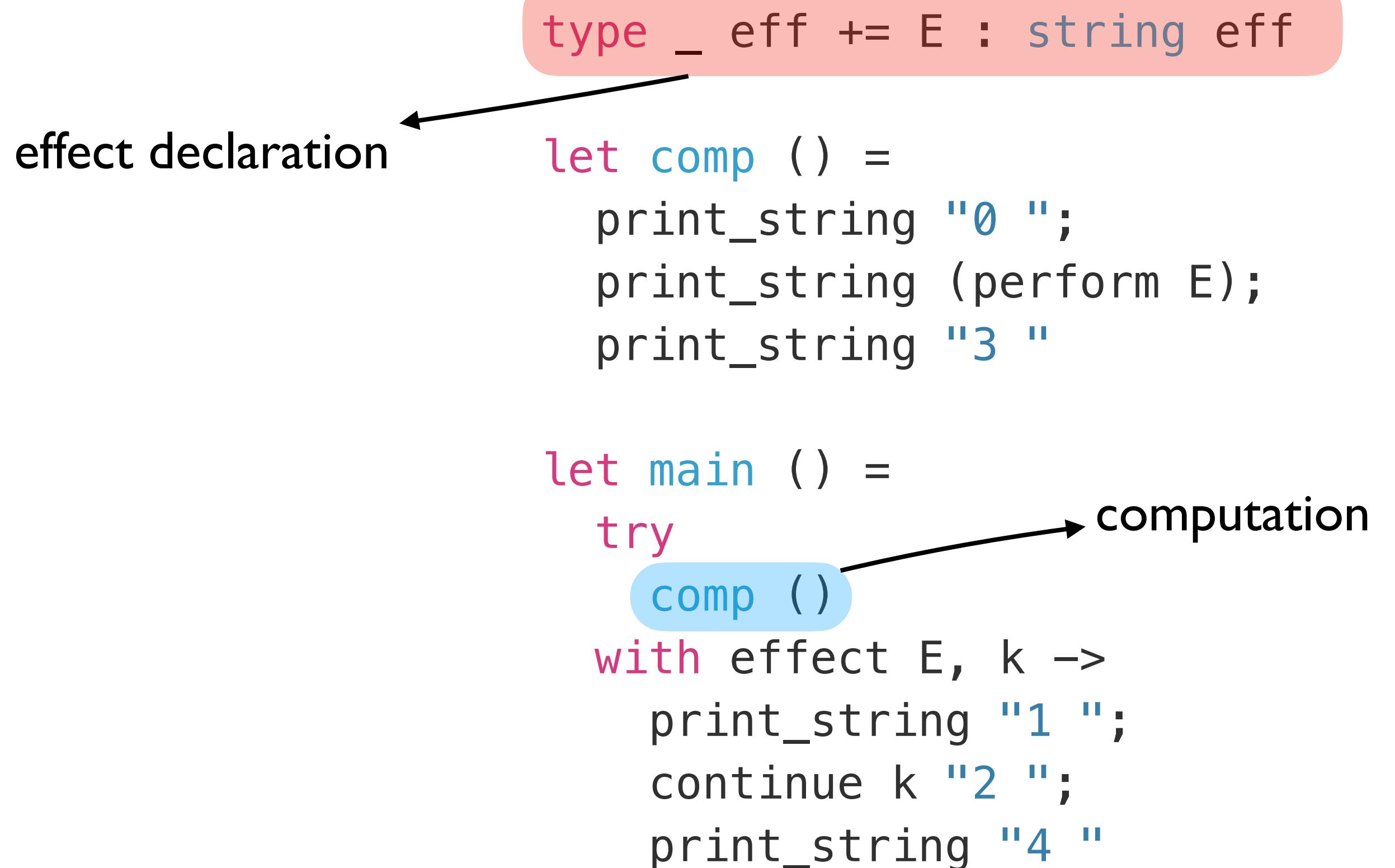
let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

Effect handlers

```
type _ eff += E : string eff  
effect declaration  
let comp () =  
  print_string "0 ";  
  print_string (perform E);  
  print_string "3 "  
  
let main () =  
  try  
    comp ()  
  with effect E, k ->  
    print_string "1 ";  
    continue k "2 ";  
    print_string "4 "
```

Effect handlers

```
type _ eff += E : string eff  
effect declaration  
let comp () =  
  print_string "0 ";  
  print_string (perform E);  
  print_string "3 "  
  
let main () =  
  try  
    comp ()  
  with effect E, k ->  
    print_string "1 ";  
    continue k "2 ";  
    print_string "4 "
```



Effect handlers

```
type _ eff += E : string eff  
effect declaration  
let comp () =  
  print_string "0 ";  
  print_string (perform E);  
  print_string "3 "  
  
let main () =  
  try  
    comp ()  
    computation  
  with effect E, k ->  
    print_string "1 ";  
    continue k "2 ";  
    print_string "4 "  
    handler
```

Effect handlers

```
type _ eff += E : string eff
effect declaration
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "
suspends current
computation

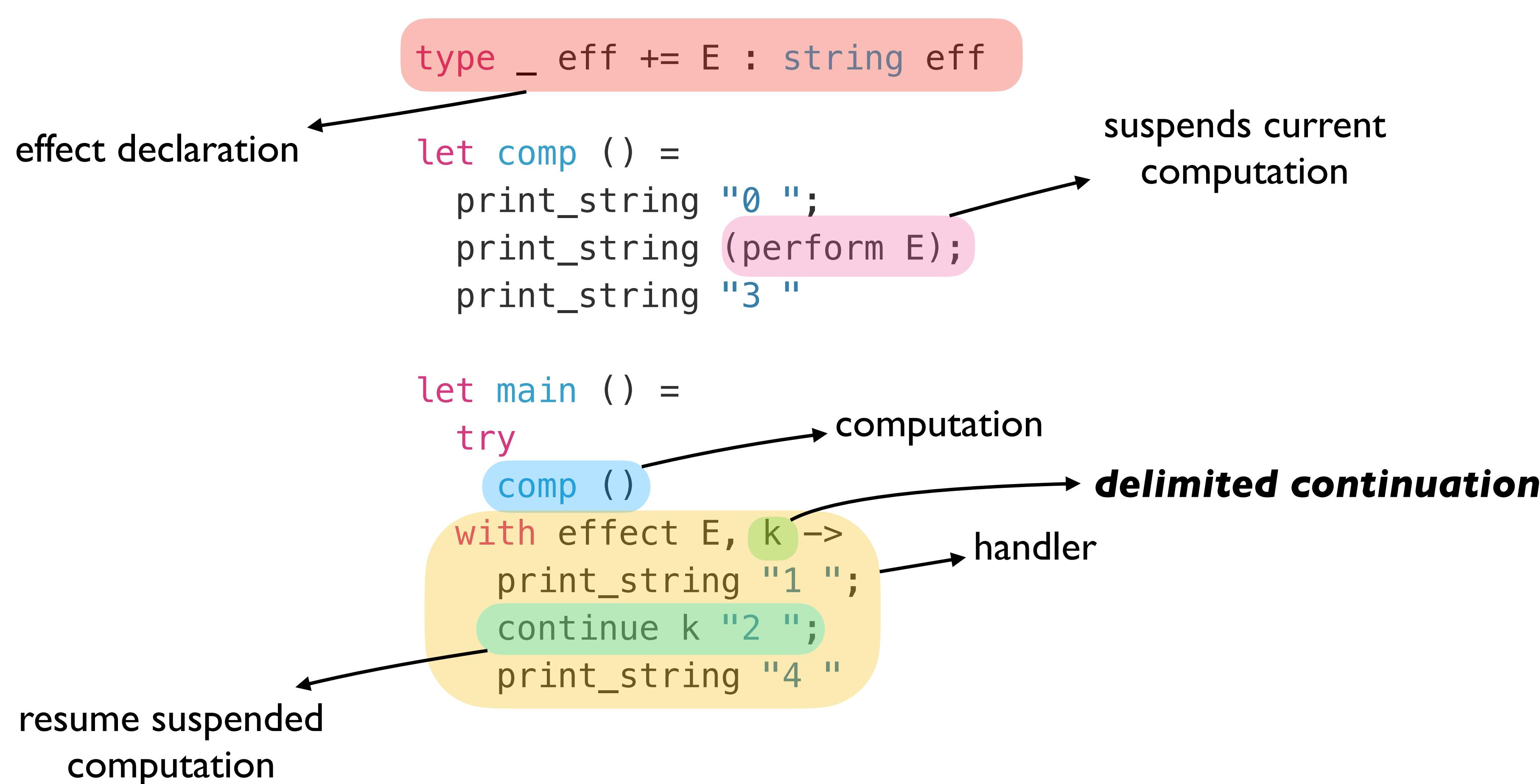
let main () =
  try
    comp ()
    computation
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
handler
```

Effect handlers

```
type _ eff += E : string eff
effect declaration
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "
suspends current
computation

let main () =
  try
    comp ()
    computation
    with effect E, k ->
      print_string "1 ";
      continue k "2 ";
      print_string "4 "
      handler
delimited continuation
```

Effect handlers



Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →



Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

pc→
```



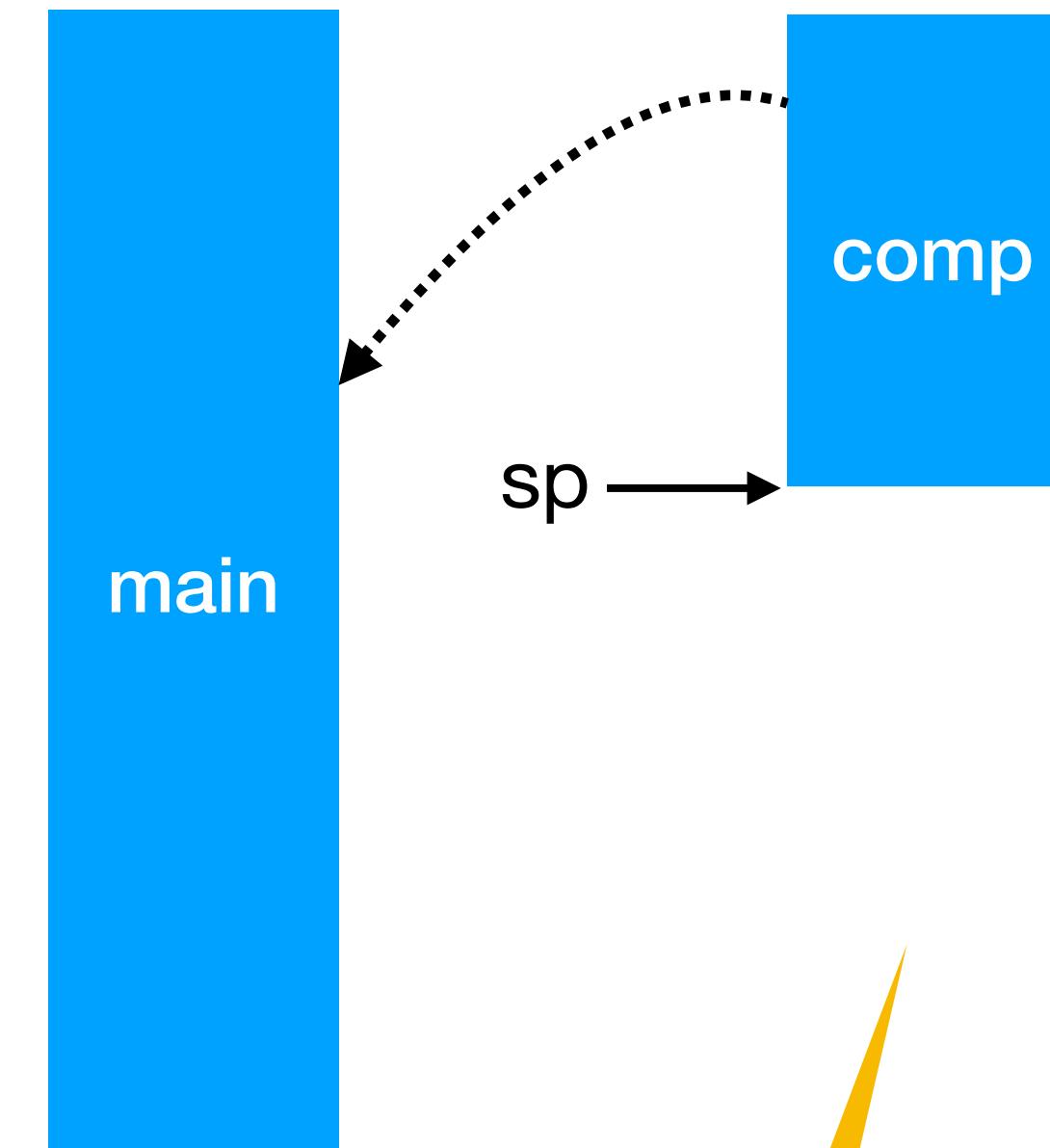
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
pc→
```

parent



Fiber: A piece of stack
+ effect handler

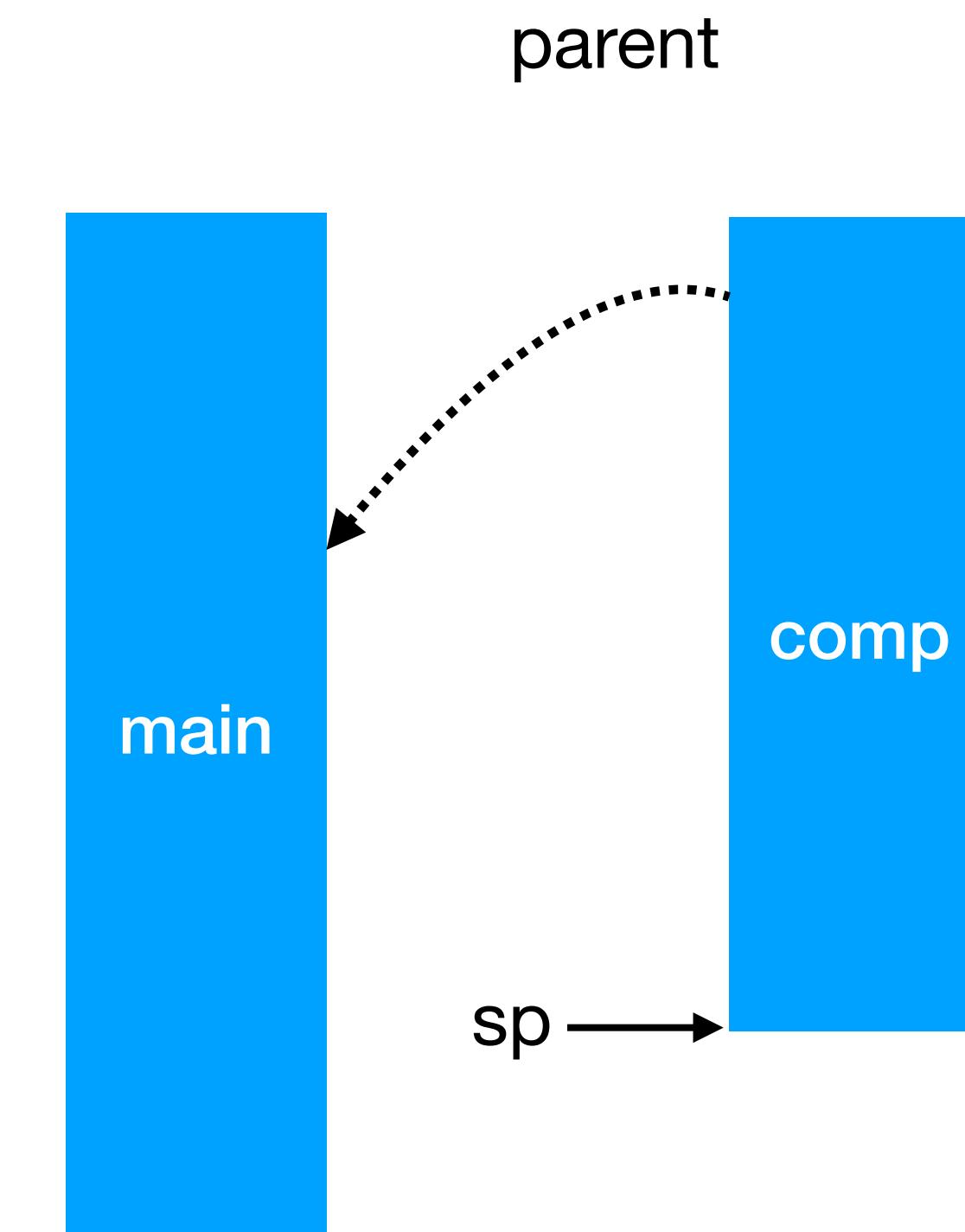
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

0



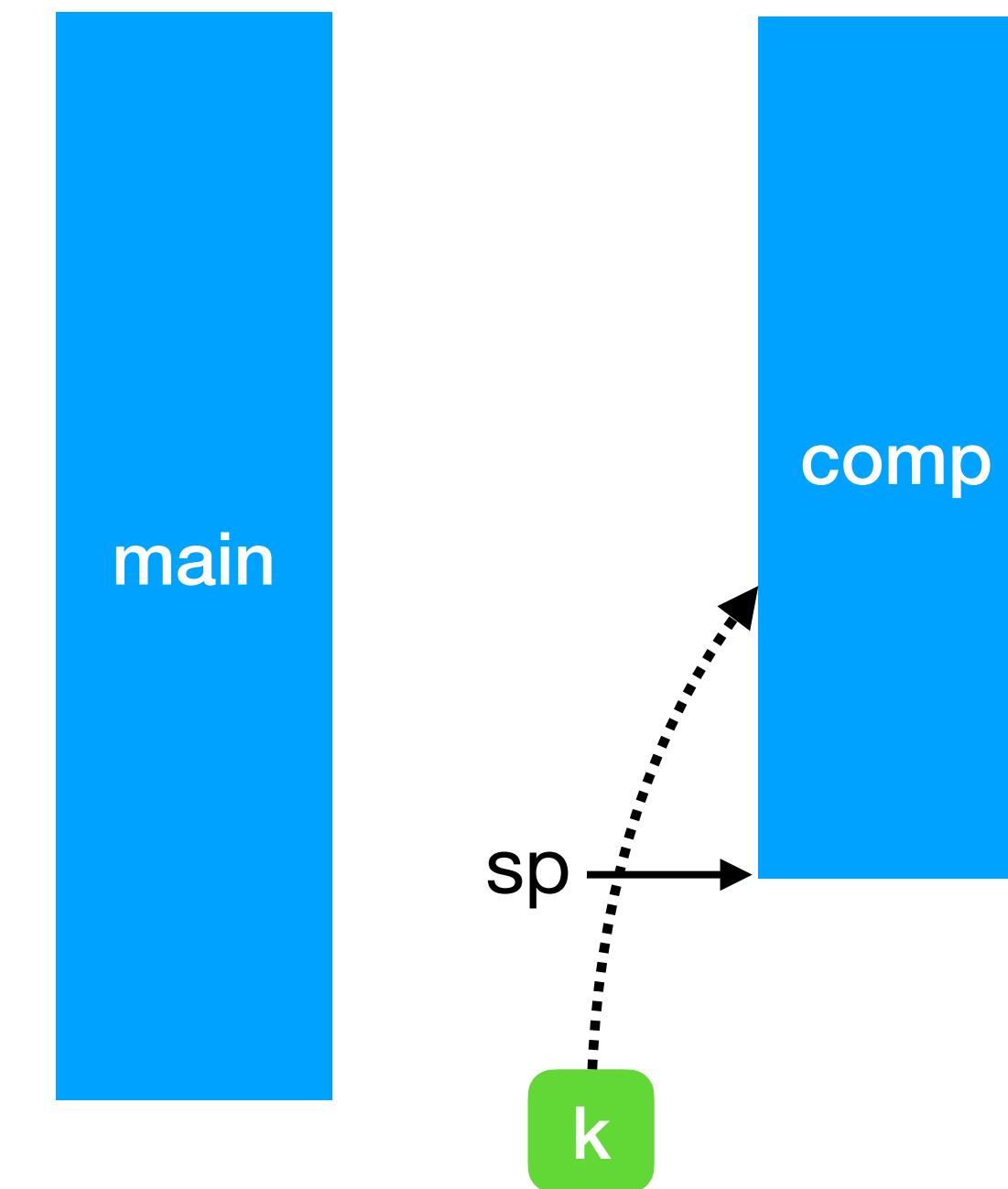
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

pc→ let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

0

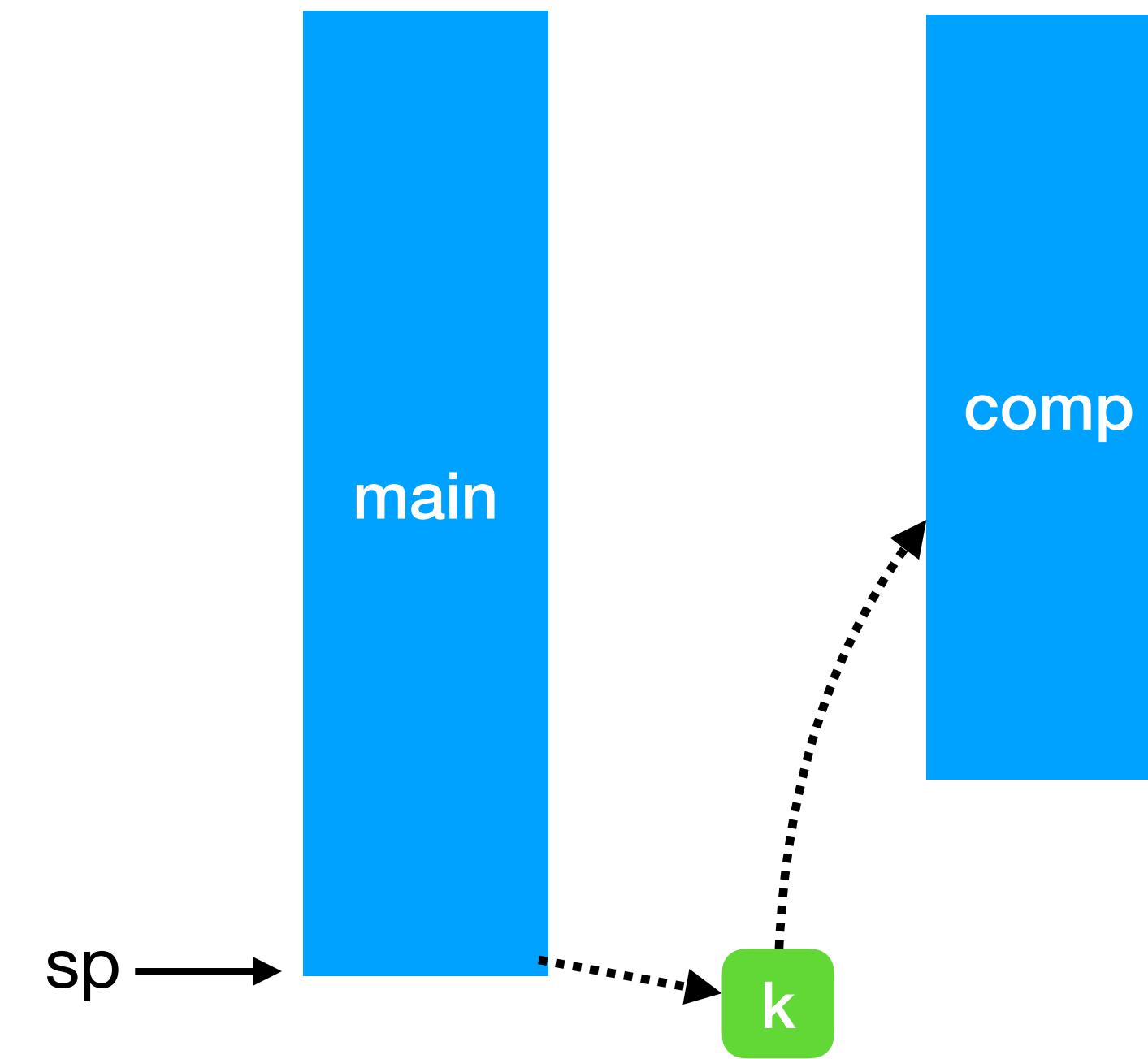


Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

pc→ let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```



0

Stepping through the example

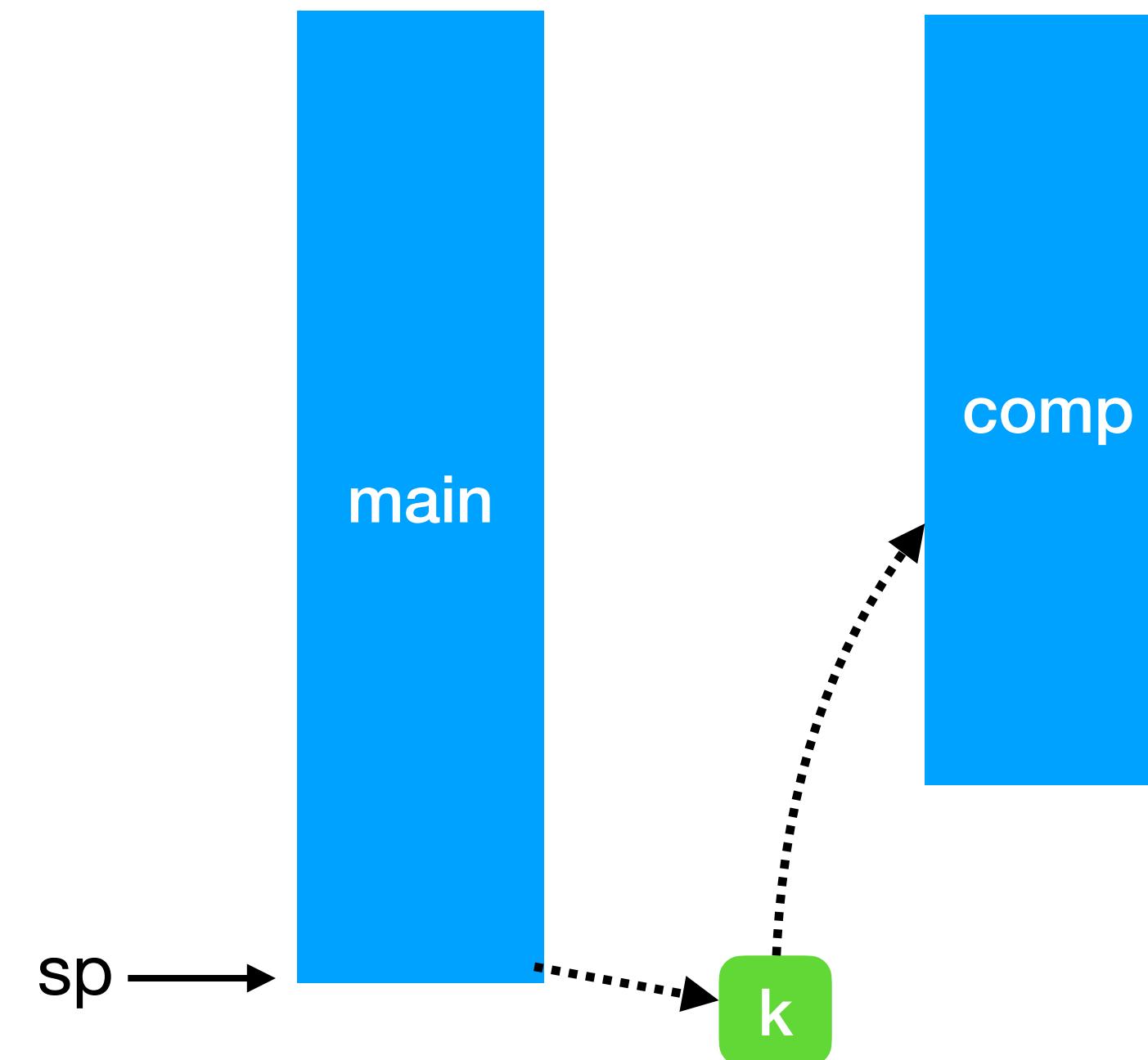
```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

pc →
```

0



Stepping through the example

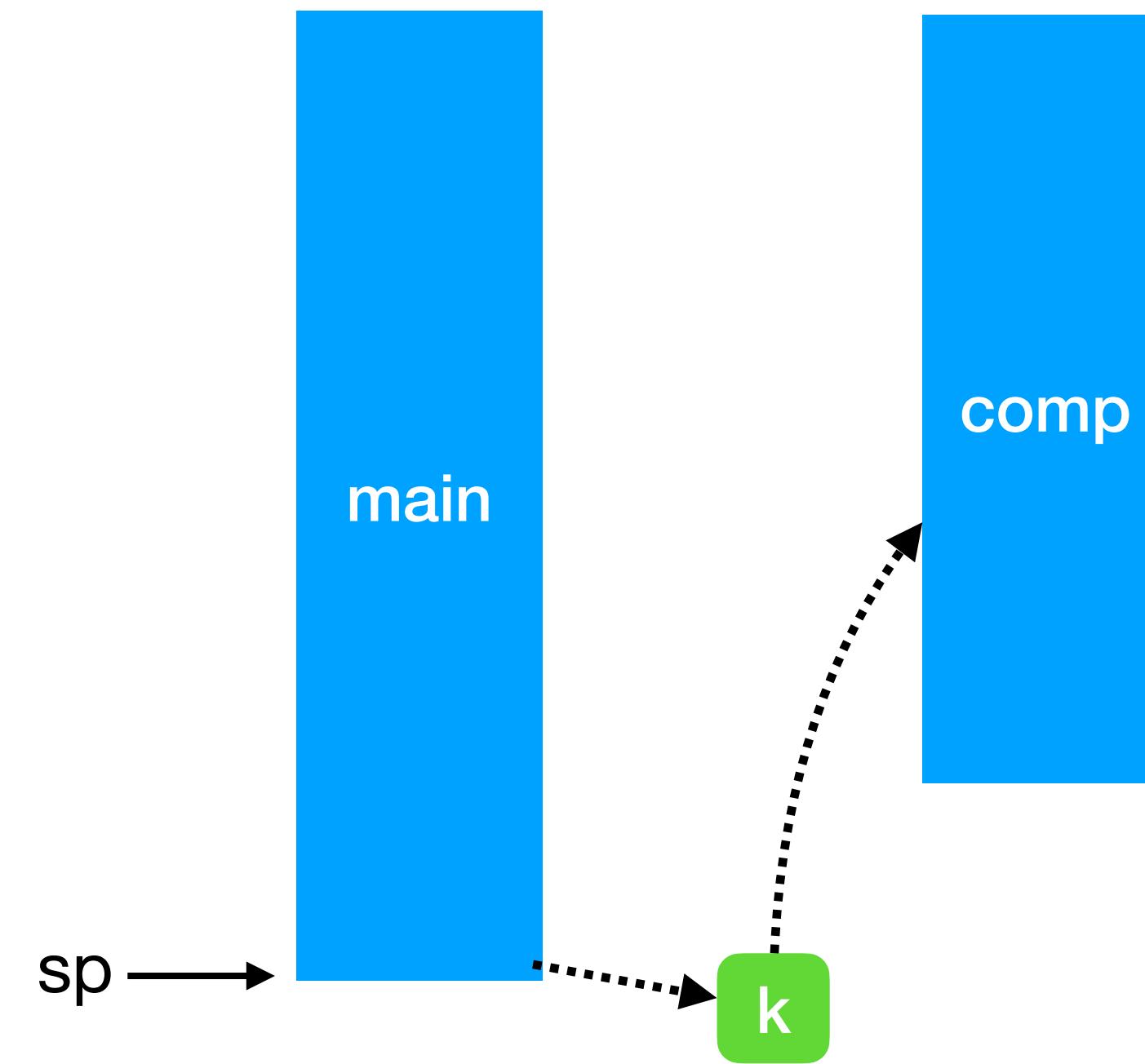
```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

pc →
```

0 1



Stepping through the example

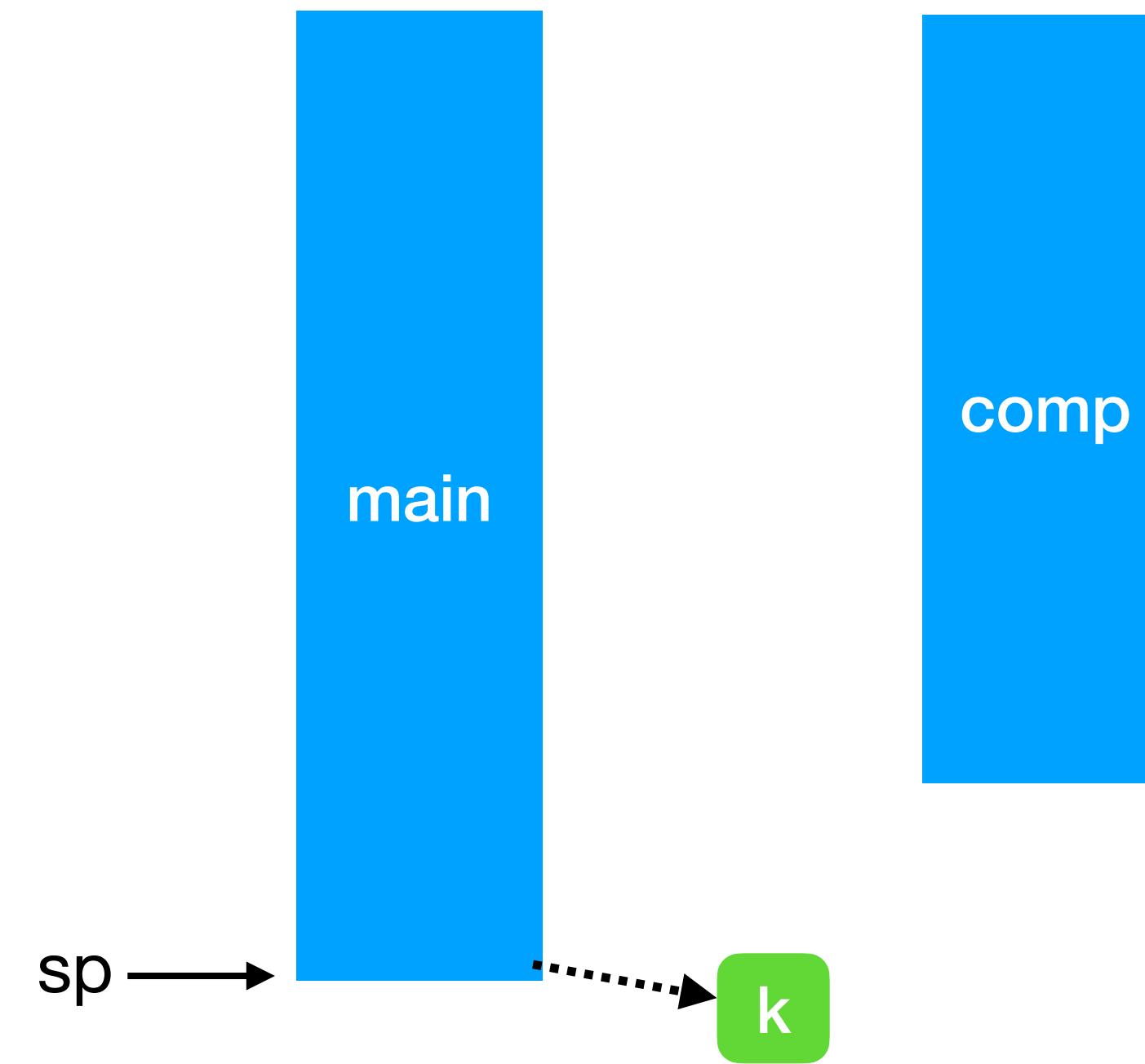
```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

pc →
```

0 1



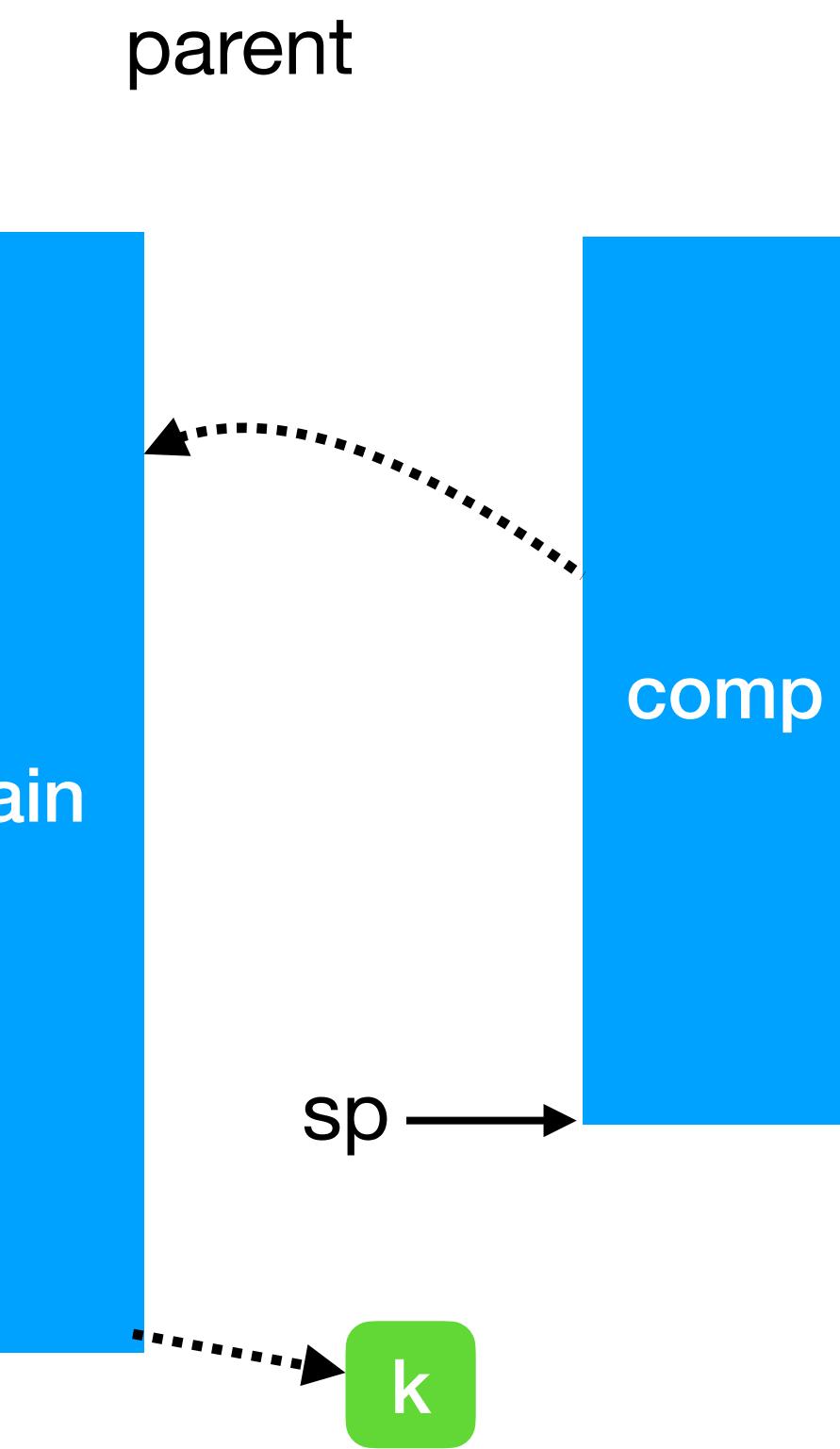
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "

pc →
```



0 1

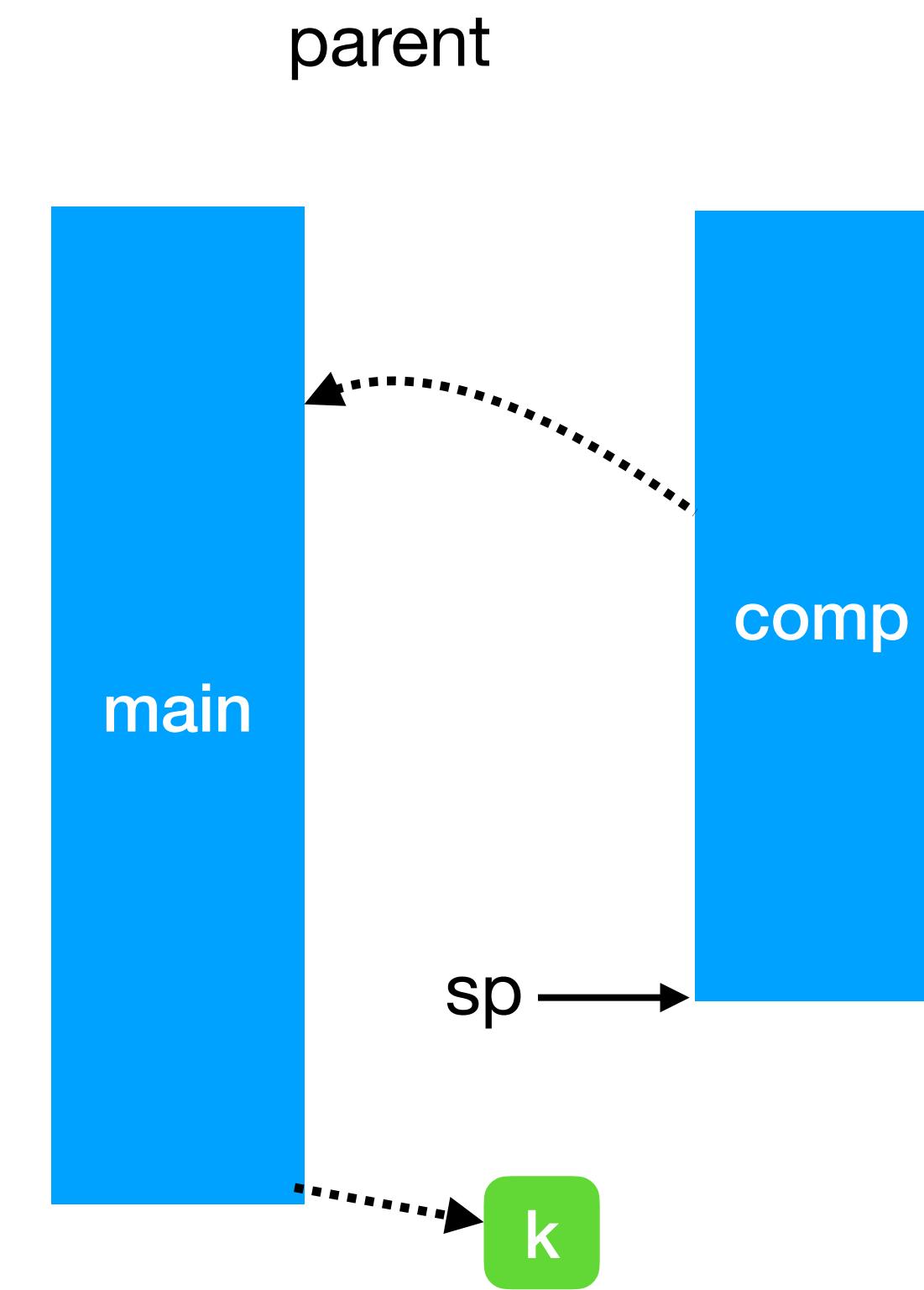
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

pc---> let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

0 1 2



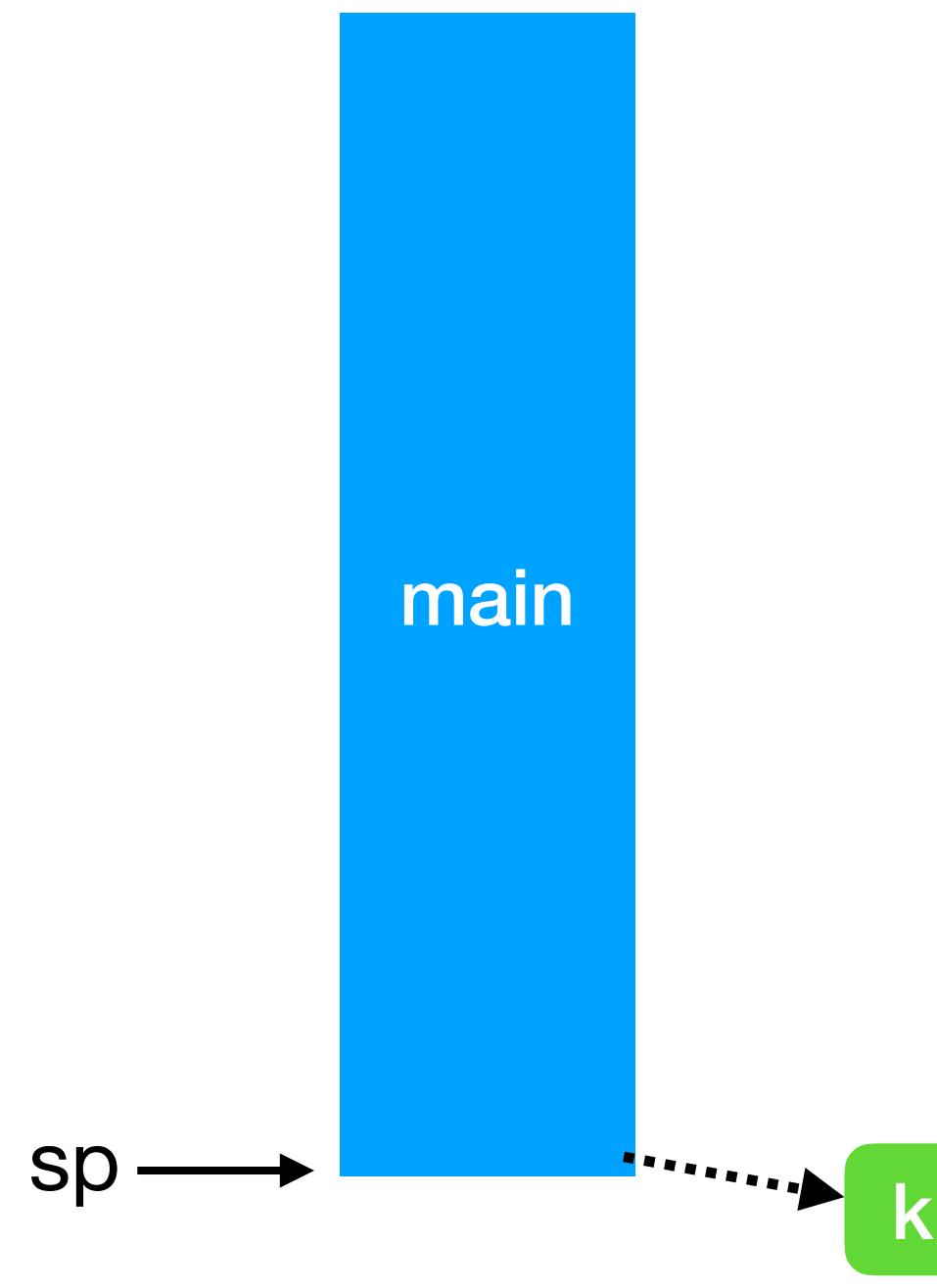
Stepping through the example

```
type 'a eff += E : string eff

let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "

let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
pc→
```

0 1 2 3



Stepping through the example

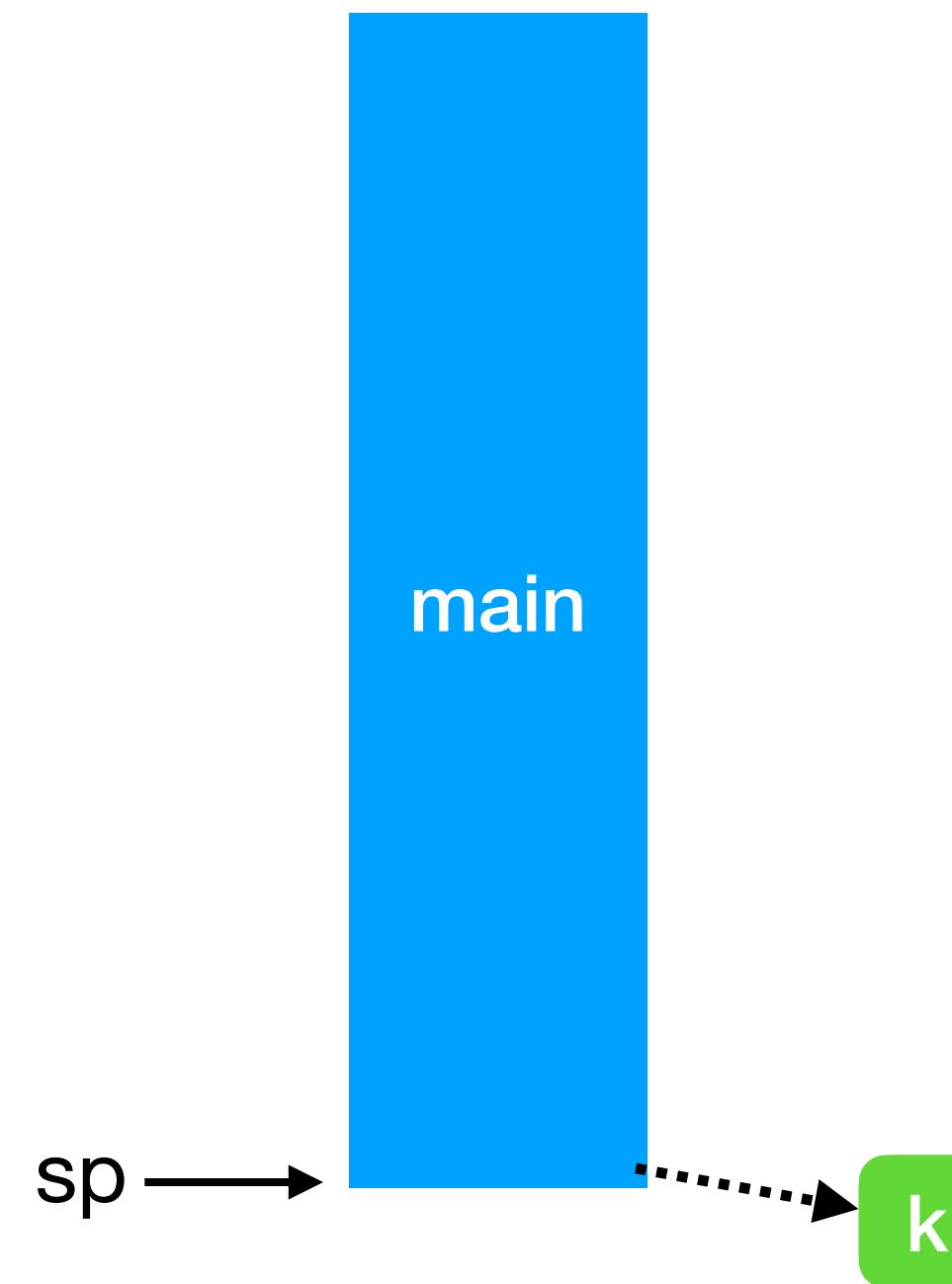
```
type 'a eff += E : string eff
```

```
let comp () =
  print_string "0 ";
  print_string (perform E);
  print_string "3 "
```

```
let main () =
  try
    comp ()
  with effect E, k ->
    print_string "1 ";
    continue k "2 ";
    print_string "4 "
```

pc →

0 1 2 3 4



Lightweight threading

```
type _ eff += Fork : (unit -> unit) -> unit eff
             | Yield : unit eff
```

Lightweight threading

```
type _ eff += Fork : (unit -> unit) -> unit eff
             | Yield : unit eff

let run main =
  ... (* assume queue of continuations *)
let run_next () =
  match dequeue () with
  | Some k -> continue k ()
  | None -> ()
in
let rec spawn f =
  match f () with
  | () -> run_next () (* value case *)
  | effect Yield, k -> enqueue k; run_next ()
  | effect (Fork f), k -> enqueue k; spawn f
in
spawn main
```

Effect Handler {

Lightweight threading

```
type _ eff += Fork : (unit -> unit) -> unit eff
             | Yield : unit eff

let run main =
  ... (* assume queue of continuations *)
let run_next () =
  match dequeue () with
  | Some k -> continue k ()
  | None -> ()
in
let rec spawn f =
  match f () with
  | () -> run_next () (* value case *)
  | effect Yield, k -> enqueue k; run_next ()
  | effect (Fork f), k -> enqueue k; spawn f
in
spawn main

let fork f = perform (Fork f)
let yield () = perform Yield
```

Effect Handler {

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline "2.b")
;;
run main
```

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline "2.b")
;;
run main
```

```
1.a
2.a
1.b
2.b
```

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline "2.b")
;;
run main
```

1.a
2.a
1.b
2.b

User-code need not be
aware of effects

Lightweight threading

```
let main () =
  fork (fun _ ->
    print_endline "1.a";
    yield ();
    print_endline "1.b");
  fork (fun _ ->
    print_endline "2.a";
    yield ();
    print_endline "2.b")
;;
run main
```

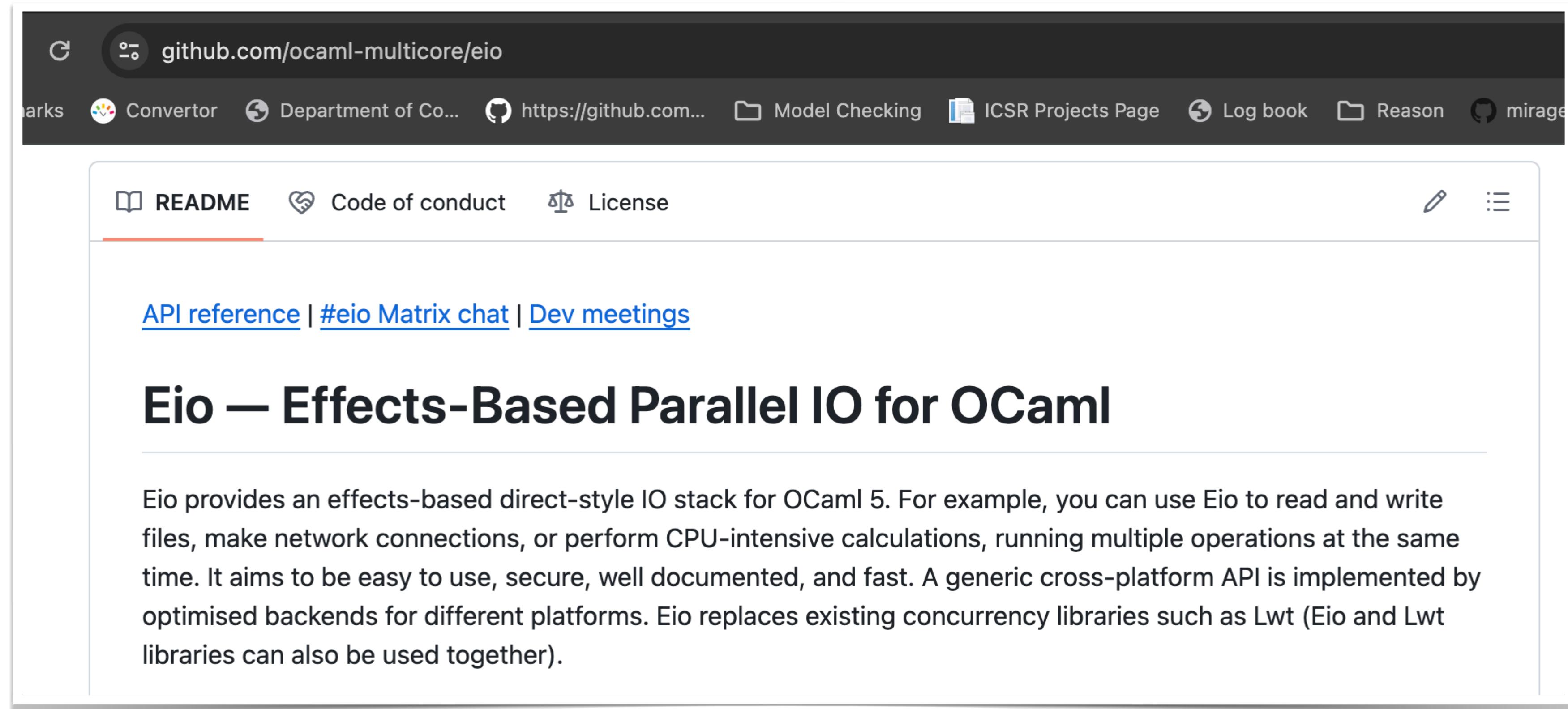
Ability to specialise scheduler
unlike GHC Haskell / Go

User-code need not be
aware of effects

1.a
2.a
1.b
2.b

Industrial-strength concurrency

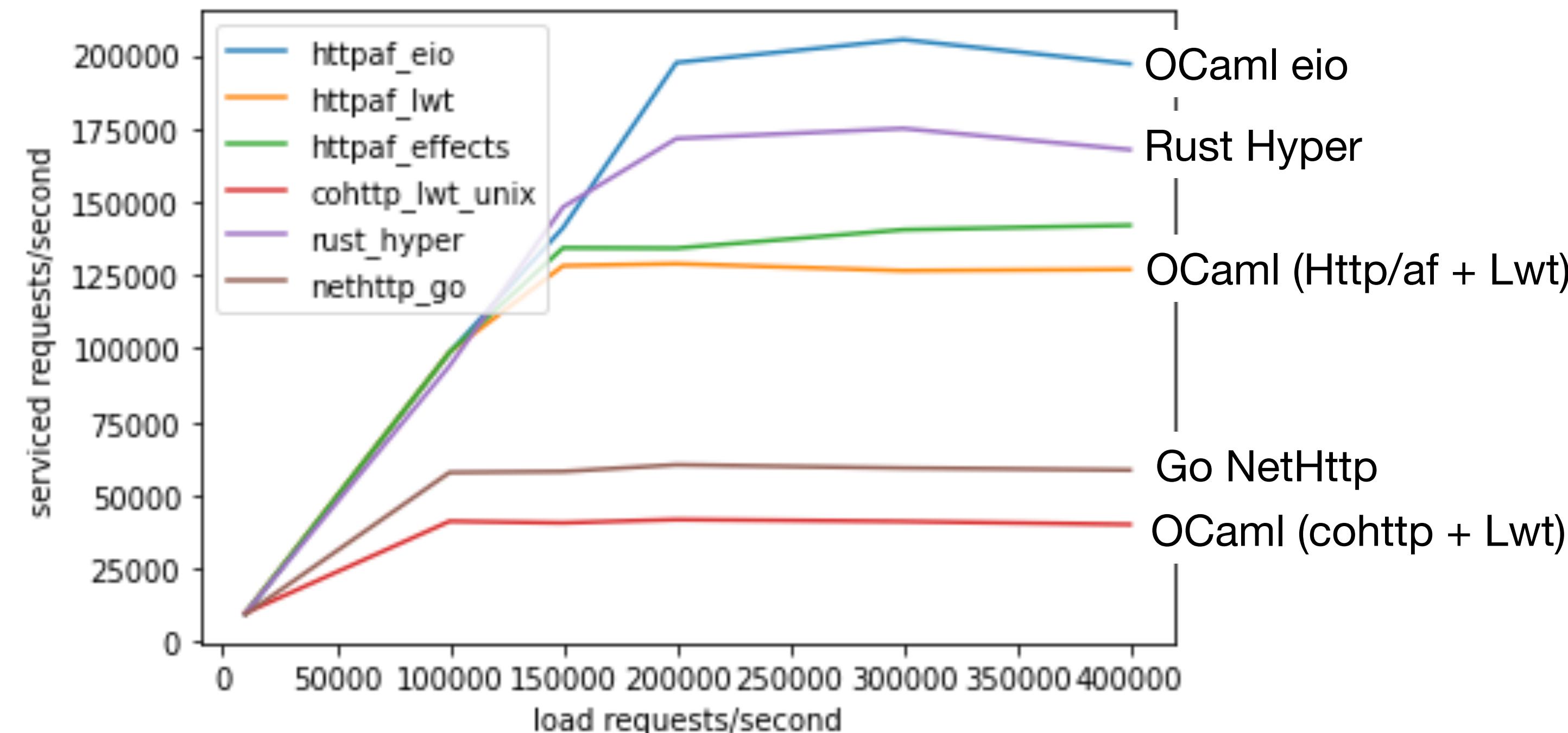
- **eio**: effects-based direct-style I/O
 - ◆ Multiple backends – epoll, select, *io_uring* (*new async io in Linux kernel*)



<https://github.com/ocaml-multicore/eio>

Industrial-strength concurrency

- **eio**: effects-based direct-style I/O
 - ◆ Multiple backends – epoll, select, *io_uring* (*new async io in Linux kernel*)

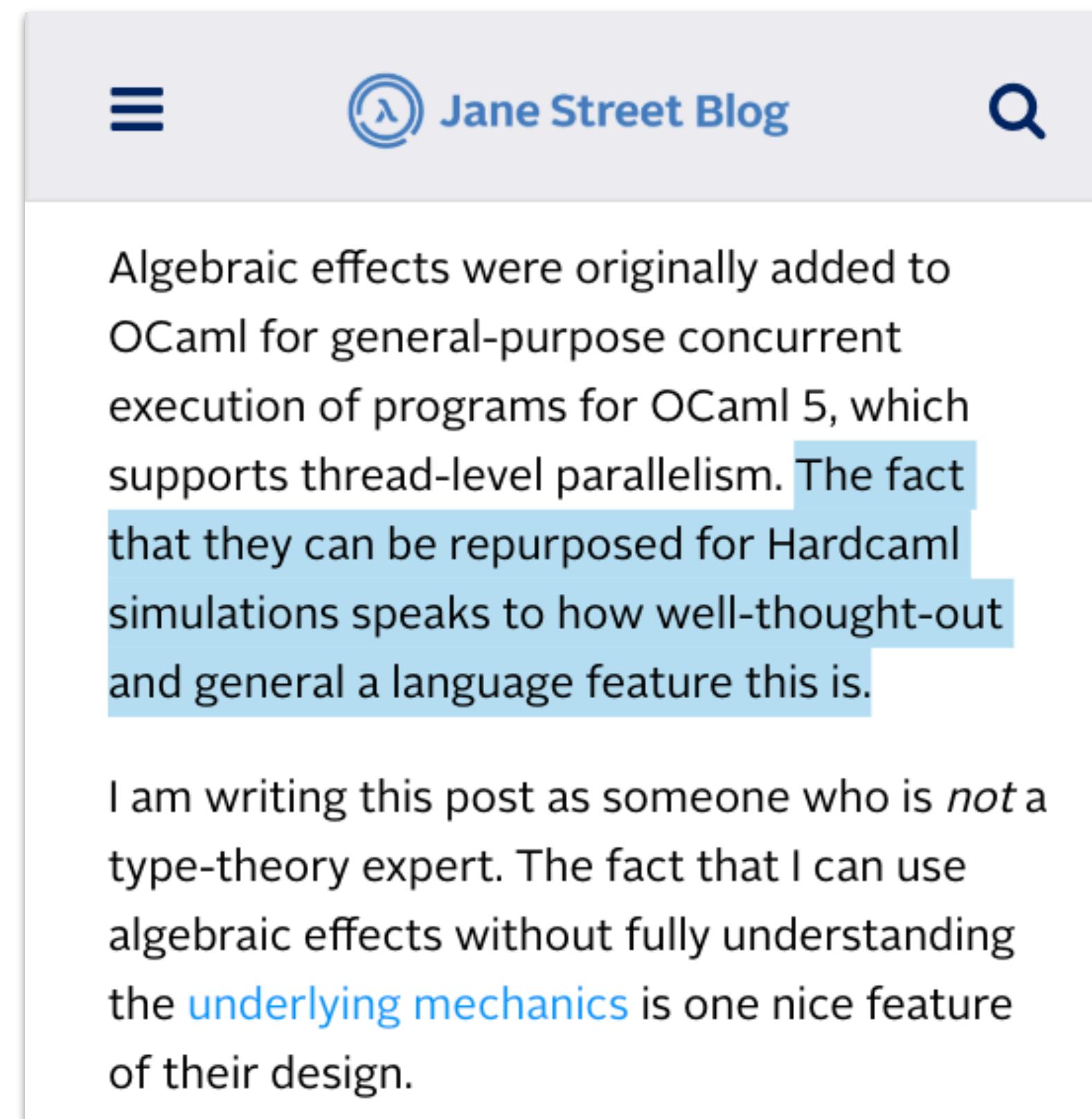


100 open connections, 60 seconds w/ *io_uring*

<https://github.com/ocaml-multicore/eio>

Unexpected uses

- Hardware simulations for HardCaml



The screenshot shows a blog post from the Jane Street Blog. The header includes a menu icon, the blog logo (a stylized lambda symbol), the text 'Jane Street Blog', and a search icon. The main content of the post discusses the use of algebraic effects in OCaml for concurrent execution and how they can be repurposed for HardCaml simulations. The author, who is not a type-theory expert, appreciates the well-thought-out design of the language. The URL of the post is visible at the bottom of the screenshot.

Algebraic effects were originally added to OCaml for general-purpose concurrent execution of programs for OCaml 5, which supports thread-level parallelism. The fact that they can be repurposed for Hardcaml simulations speaks to how well-thought-out and general a language feature this is.

I am writing this post as someone who is *not* a type-theory expert. The fact that I can use algebraic effects without fully understanding the [underlying mechanics](#) is one nice feature of their design.

<https://blog.janestreet.com/fun-with-algebraic-effects-hardcaml/>

Further reading

Control structures in programming languages: from goto to algebraic effects

Xavier Leroy

This book is a journey through the design space and history of programming languages from the perspective of control structures: the language mechanisms that enable programs to control their execution flows. Starting with the “goto” jumps of early programming languages and the emergence of structured programming in the 1960s, the book explores advanced control structures for imperative languages such as generators and coroutines, then develops alternate views of control in functional languages, first as continuations and their control operators, then as algebraic effects and effect handlers. Blending history, code examples, and theory, the book offers an original, comparative perspective on programming languages, as well as an extensive introduction to algebraic effects and other contemporary research topics in P.L.

Publication history

To be published by Cambridge University Press.

Book preview

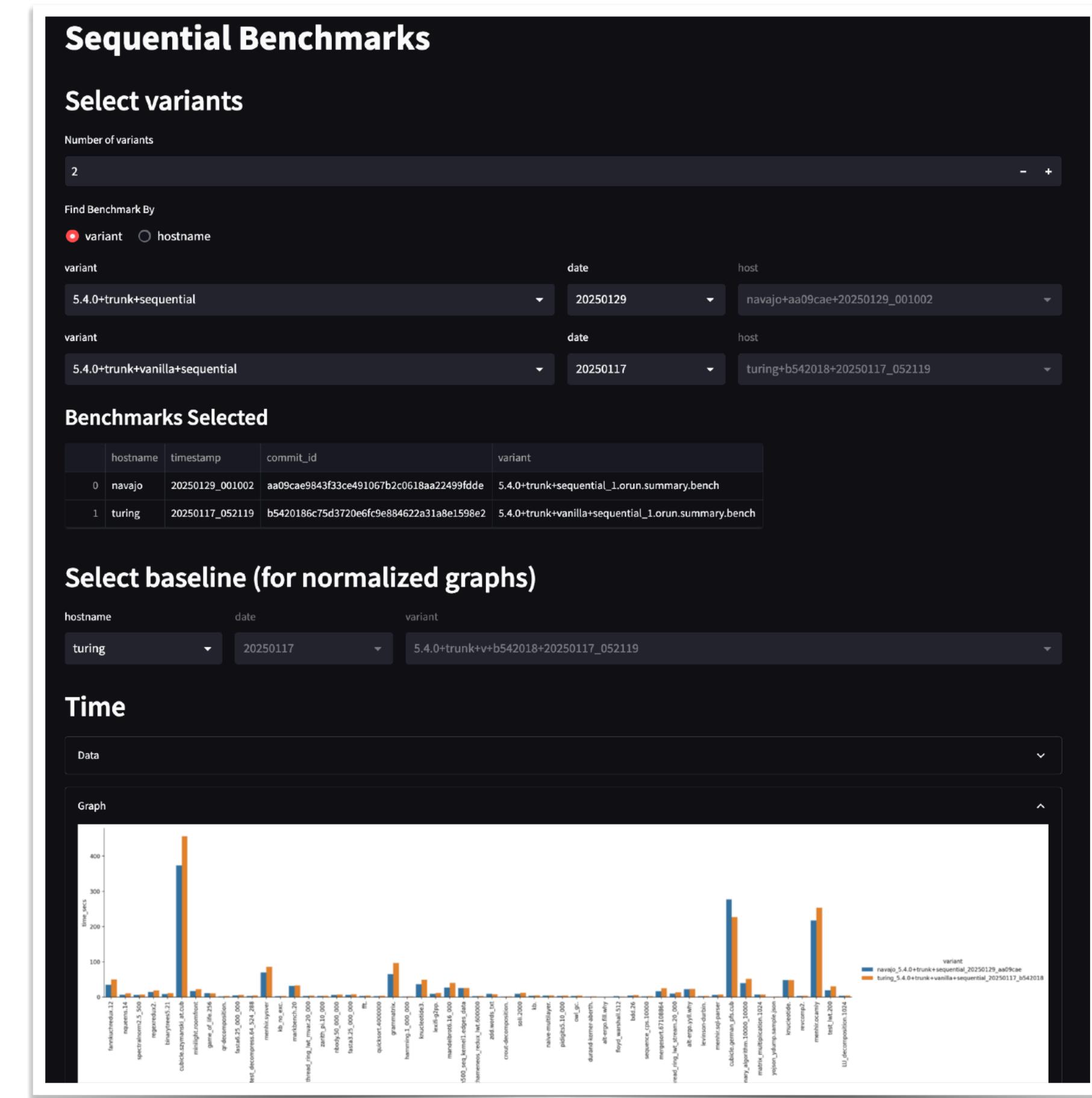
This is an HTML preview of the book, generated with [Hevea](#). License: [CC-BY-NC-ND 4.0](#).

- [Table of contents](#)
- [Introduction](#)

<https://xavierleroy.org/control-structures/>

Building confidence – Benchmarking

- *Rigorous, continuous* benchmarking on *real-world programs*
- sandmark.tarides.com – Benchmark suite, Infra and runners



Building confidence – CI for package universe

- ***Can the new compiler build the existing universe?***
 - Build the OPAM universe of packages against ***upstream*** and ***multicore*** compilers

Building confidence – CI for package universe

- *Can the new compiler build the existing universe?*
 - Build the OPAM universe of packages against *upstream* and *multicore* compilers

	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

Building confidence – CI for package universe

- ***Can the new compiler build the existing universe?***
 - Build the OPAM universe of packages against ***upstream*** and ***multicore*** compilers

	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

You can contribute to the compiler development without hacking on the compiler

Release and Long Tail

- **Opened – Dec 2021, Merged – Jan 2022**
 - *....A few months of iteration to fix design issues and bugs....*

Multicore OCaml #10831

Merged xavierleroy merged 4,103 commits into [ocaml:trunk](#) from [ocaml-multicore:multicore-pr](#) on Jan 10, 2022

Conversation 393 Commits 250 Checks 0 Files changed 300+ +22,955 -14,062

kayceesrk commented on Dec 21, 2021 · edited Member

This PR adds support for shared-memory parallelism through domains and direct-style concurrency through effect handlers (without syntactic support). It intends to have backwards compatibility in terms of language features, C API, and also the performance of single-threaded code.

For users

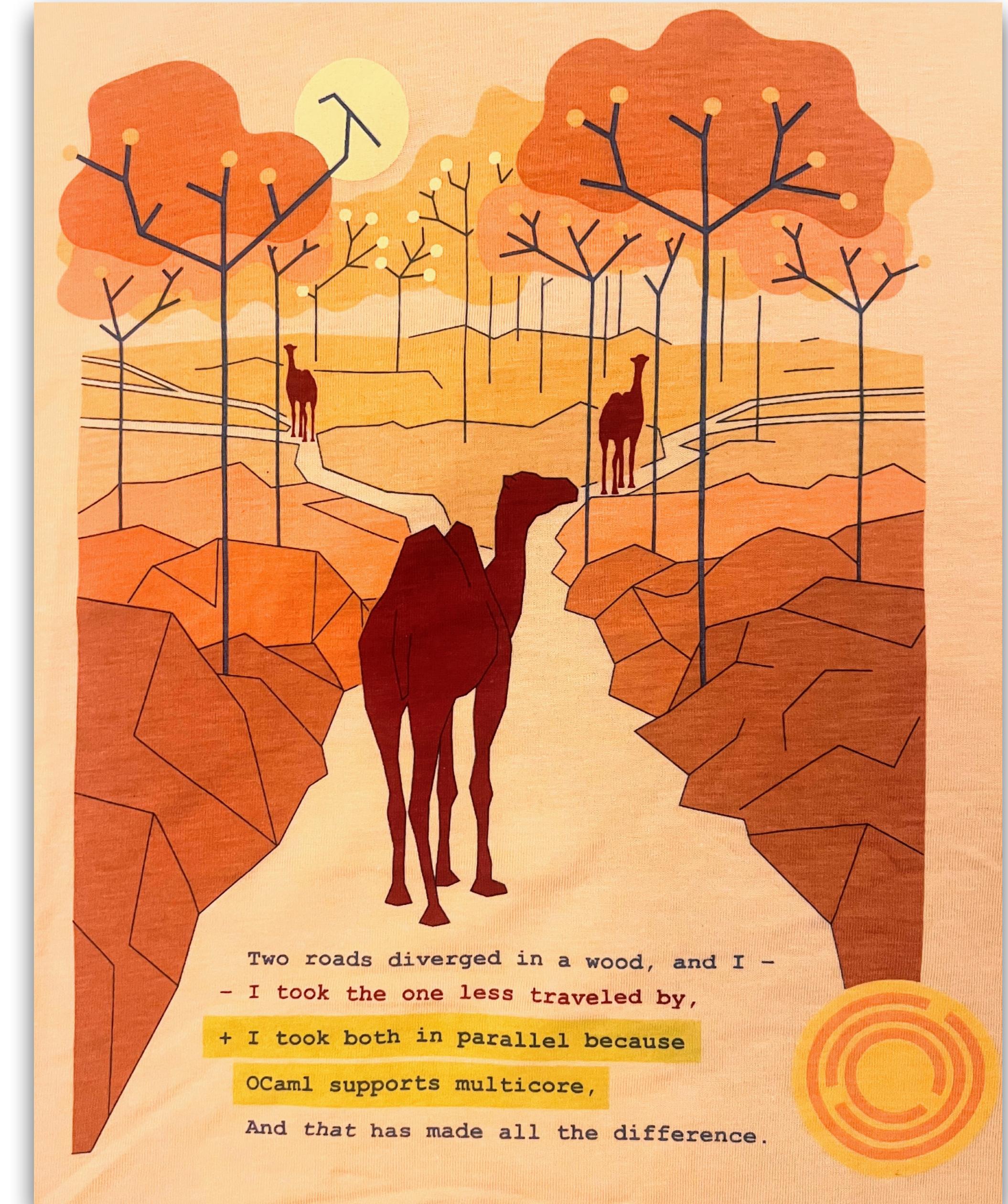
If you want to learn more about Multicore OCaml, please have a look at the [multicore](#)

Reviewers

- abysmal
- gasche
- sadiqj
- avsm
- xavierleroy
- damiendoligez
- dra27

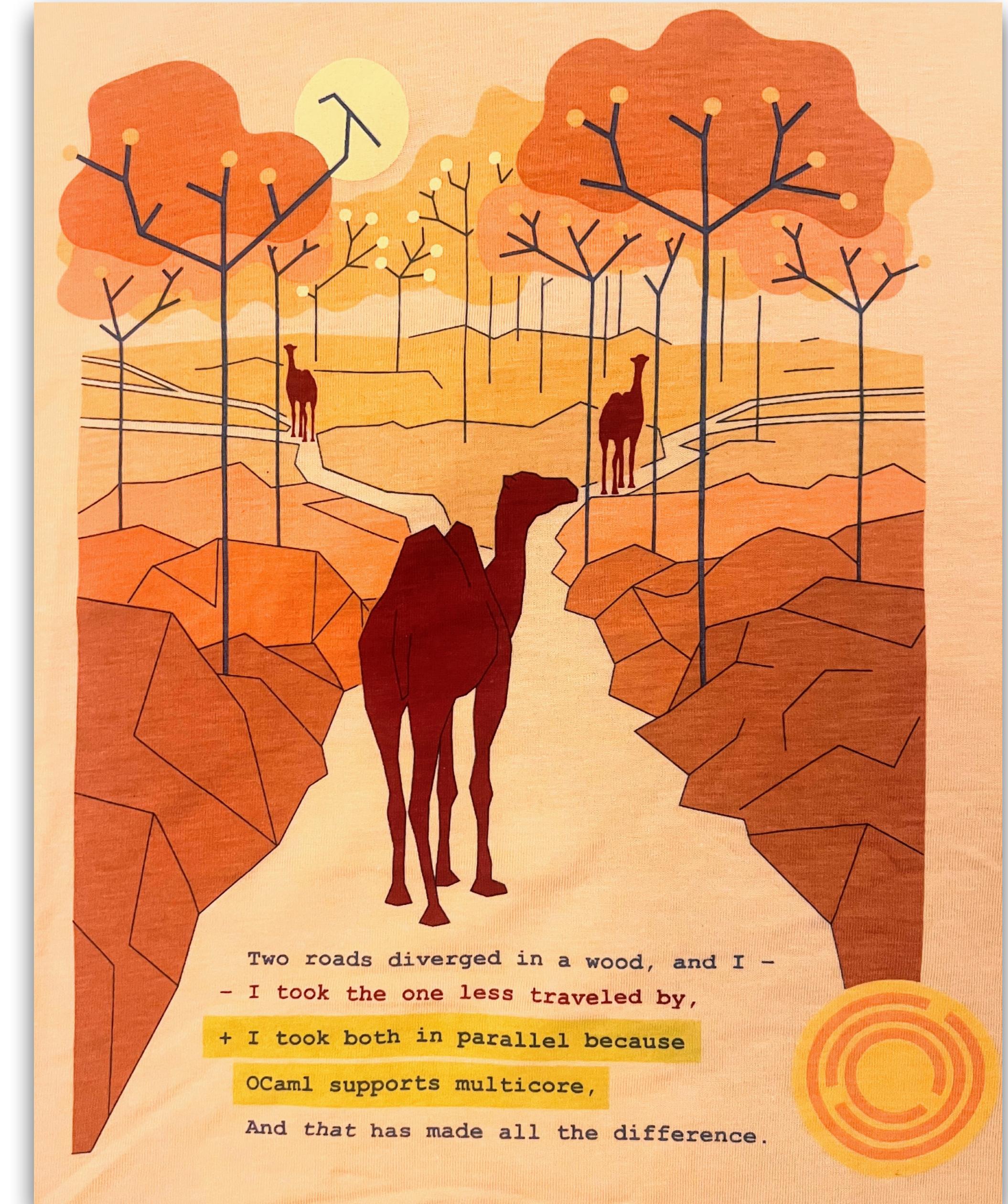
Release and Long Tail

- **Opened** – Dec 2021, **Merged** – Jan 2022
 -A few months of iteration to fix design issues and bugs....
- **Released** – Dec 16 2022, as OCaml 5.0



Release and Long Tail

- **Opened** – Dec 2021, **Merged** – Jan 2022
 -A few months of iteration to fix design issues and bugs....
- **Released** – Dec 16 2022, as OCaml 5.0
- **Long tail** of adding missing features, bug fixes and performance improvements
 - 5.1 – Sep 2023
 - 5.2 – May 2024
 - 5.3 – Jan 2025
 - 5.4 – Sep 2025



What's next for OCaml?

- **OxCaml** – Bridging the performance and safety gap between OCaml and Rust
 - *Data-race-free parallelism* through *modes*
 - Better control over object layout, allocations and GC



The screenshot shows the homepage of the OxCaml website. The header features the Jane Street logo and the text "OxCaml". Below the header is a navigation bar with links for "About", "Documentation", and "Get OxCaml". The main content area has a blue background with a white ox logo and the text "OCaml, Oxidized!". A grey sidebar on the right contains the text "OxCaml" and a description: "OxCaml is a fast-moving set of extensions to the OCaml programming language. It is both Jane Street's production compiler, as well as a laboratory for experiments focused towards making OCaml better for performance-oriented programming. Our hope is that these extensions can over time be contributed to upstream OCaml." At the bottom of the sidebar is a link: <https://oxcaml.org>.

<https://oxcaml.org>

What's next for OCaml?

- **OxCaml** – Bridging the performance and safety gap between OCaml and Rust
 - *Data-race-free parallelism* through *modes*
 - Better control over object layout, allocations and GC
- Draws lessons from Multicore OCaml execution
 - Several award-winning papers at POPL, ICFP, OOPSLA
 - CI for the external universe – <https://oxcaml.check.ci.dev/>



<https://oxcaml.org>

What's next for OCaml?

- **OxCaml** – Bridging the performance and safety gap between OCaml and Rust
 - *Data-race-free parallelism* through *modes*
 - Better control over object layout, allocations and GC
- Draws lessons from Multicore OCaml execution
 - Several award-winning papers at POPL, ICFP, OOPSLA
 - CI for the external universe – <https://oxcaml.check.ci.dev/>
- But different in other ways...
 - In production at Jane Street
 - Valuable user-feedback-oriented design



CS6868 Concurrent Programming

CS6868 Spring 2026

Schedule Assignments Resources GitHub



Photo © Madras Inherited

CS6868: Concurrent Programming

Course Overview

This course explores the fundamentals of concurrent and parallel programming with a focus on shared-memory multiprocessor systems. You'll learn to design and implement correct, efficient concurrent programs while understanding the theoretical foundations and practical challenges of concurrency and parallelism.

Key Topics:

- Principles of concurrent programming
- Mutual exclusion and synchronization
- Concurrent data structures
- Lock-free and wait-free algorithms
- Memory models and consistency
- Parallel programming patterns
- Effect handlers
- Nested parallelism, Asynchronous I/O
- Practical implementations using OCaml 5's multicore features
- Safe parallel programming with OxCaml

The course uses **OCaml 5** with native support for parallelism via [domains](#) and concurrency via [effect handlers](#), providing hands-on experience with modern concurrent programming techniques.

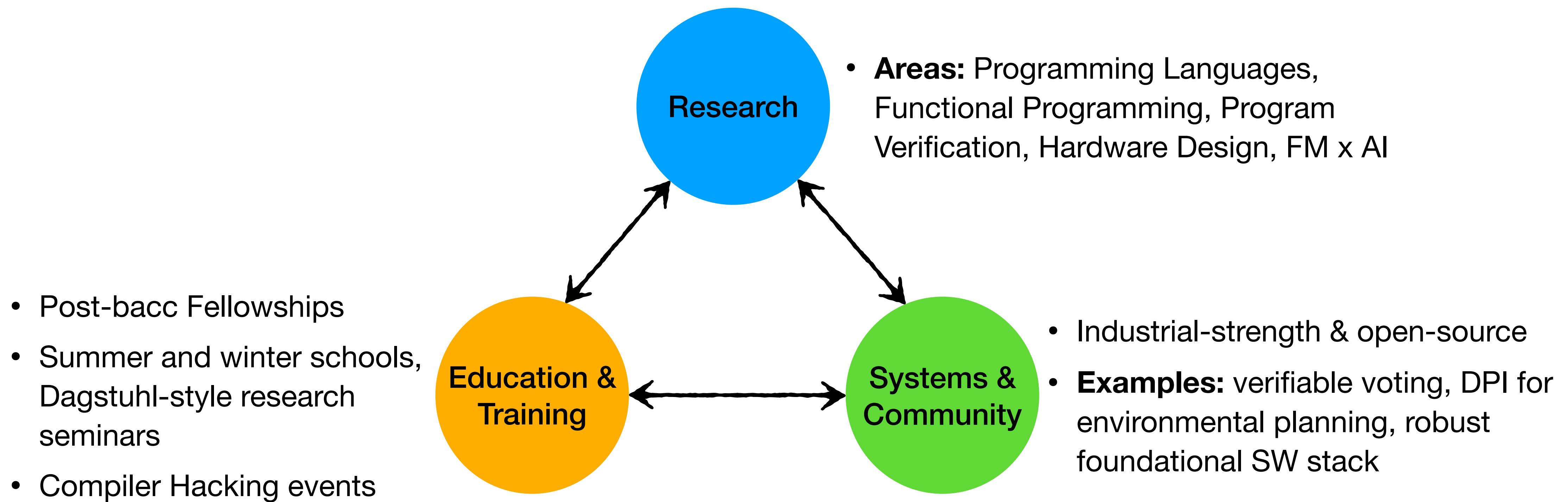
https://kcsrk.info/cs6868_s26/



*Build research and educational capacity for crafting **efficient, reliable**
and trustworthy software with mathematical guardrails.*

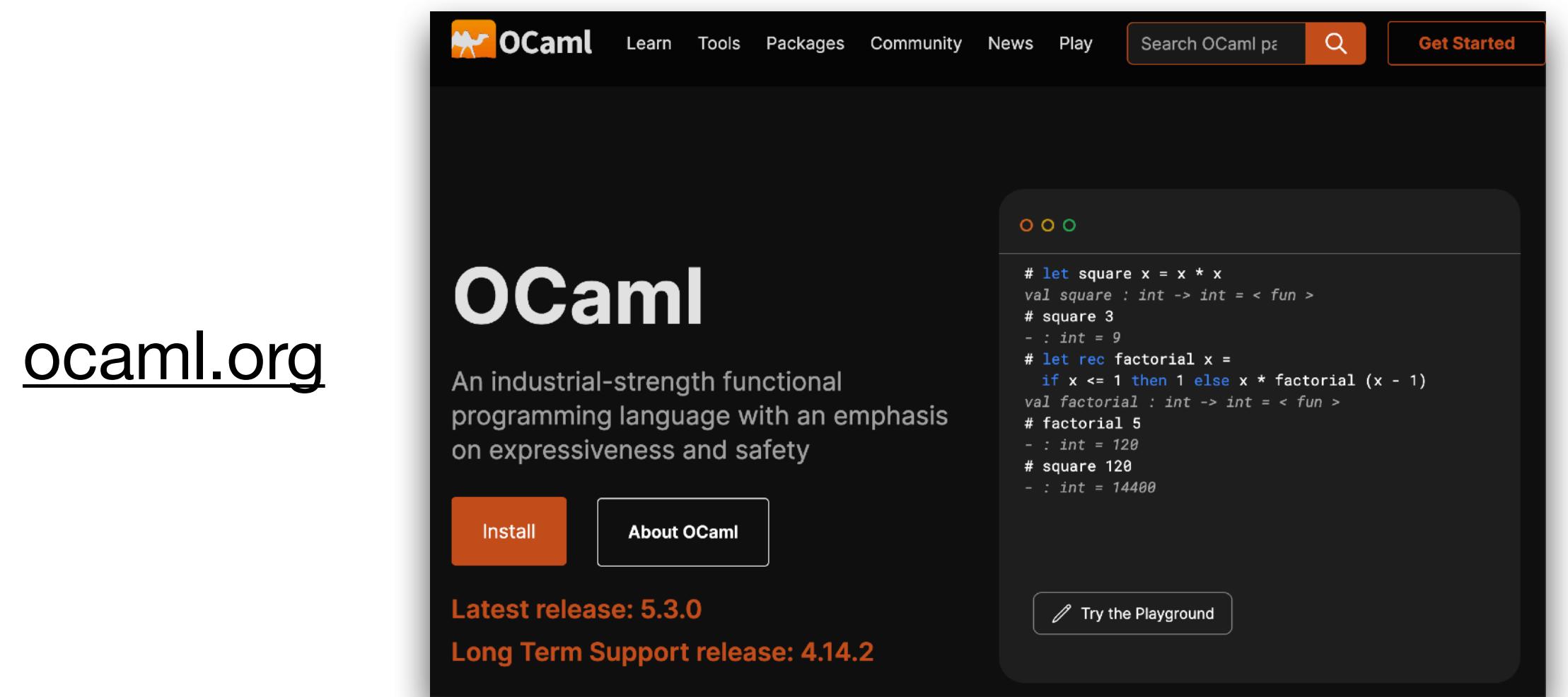


*Build research and educational capacity for crafting **efficient, reliable** and **trustworthy software with mathematical guardrails**.*

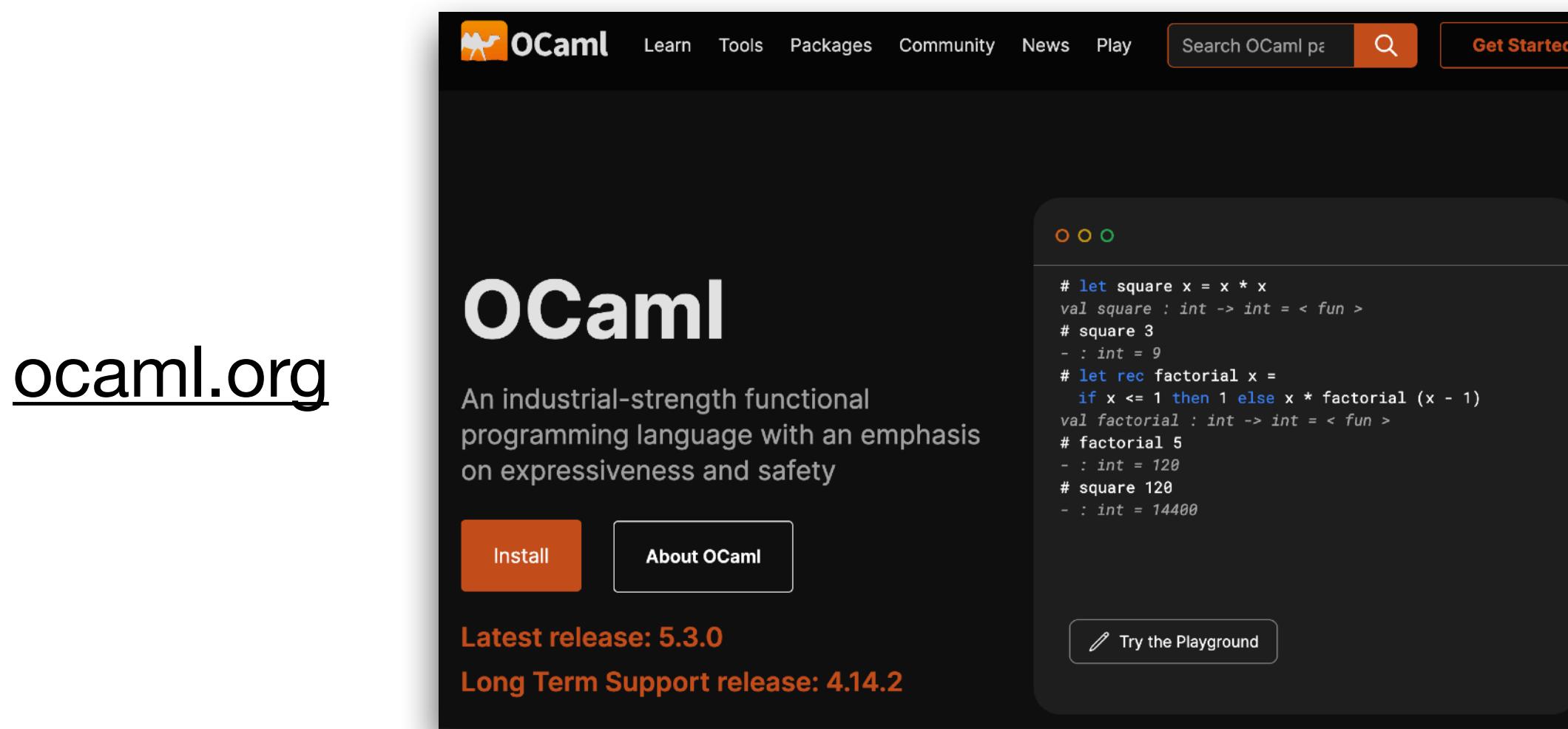


Get Involved!

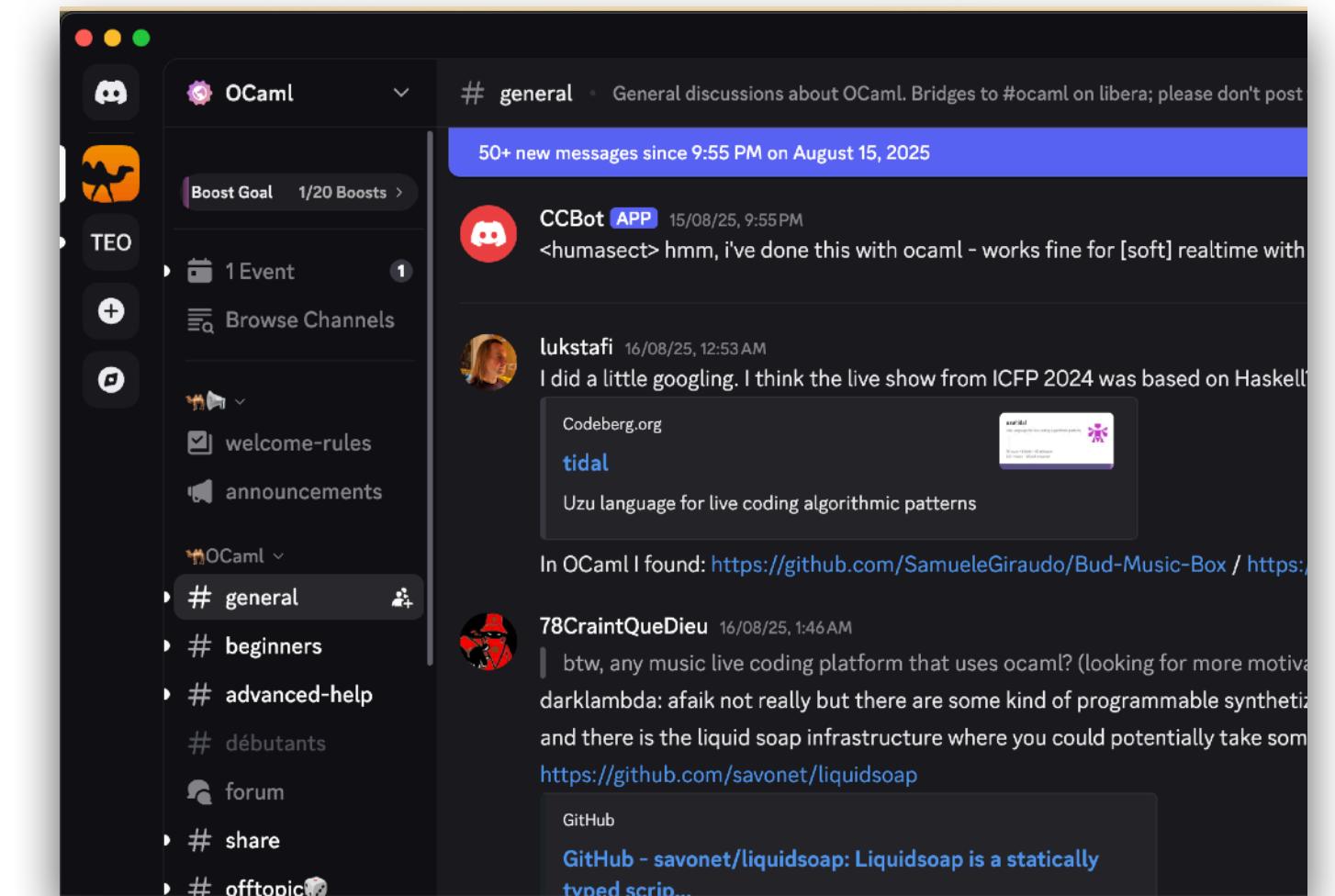
Get Involved!



Get Involved!



The screenshot shows the OCaml website at ocaml.org. The header includes the OCaml logo, navigation links (Learn, Tools, Packages, Community, News, Play), a search bar, and a "Get Started" button. The main content features the OCaml logo, a tagline "An industrial-strength functional programming language with an emphasis on expressiveness and safety", and two buttons: "Install" and "About OCaml". Below this, it says "Latest release: 5.3.0" and "Long Term Support release: 4.14.2". A code editor window displays OCaml code for calculating the square and factorial of a number. A "Try the Playground" button is at the bottom.

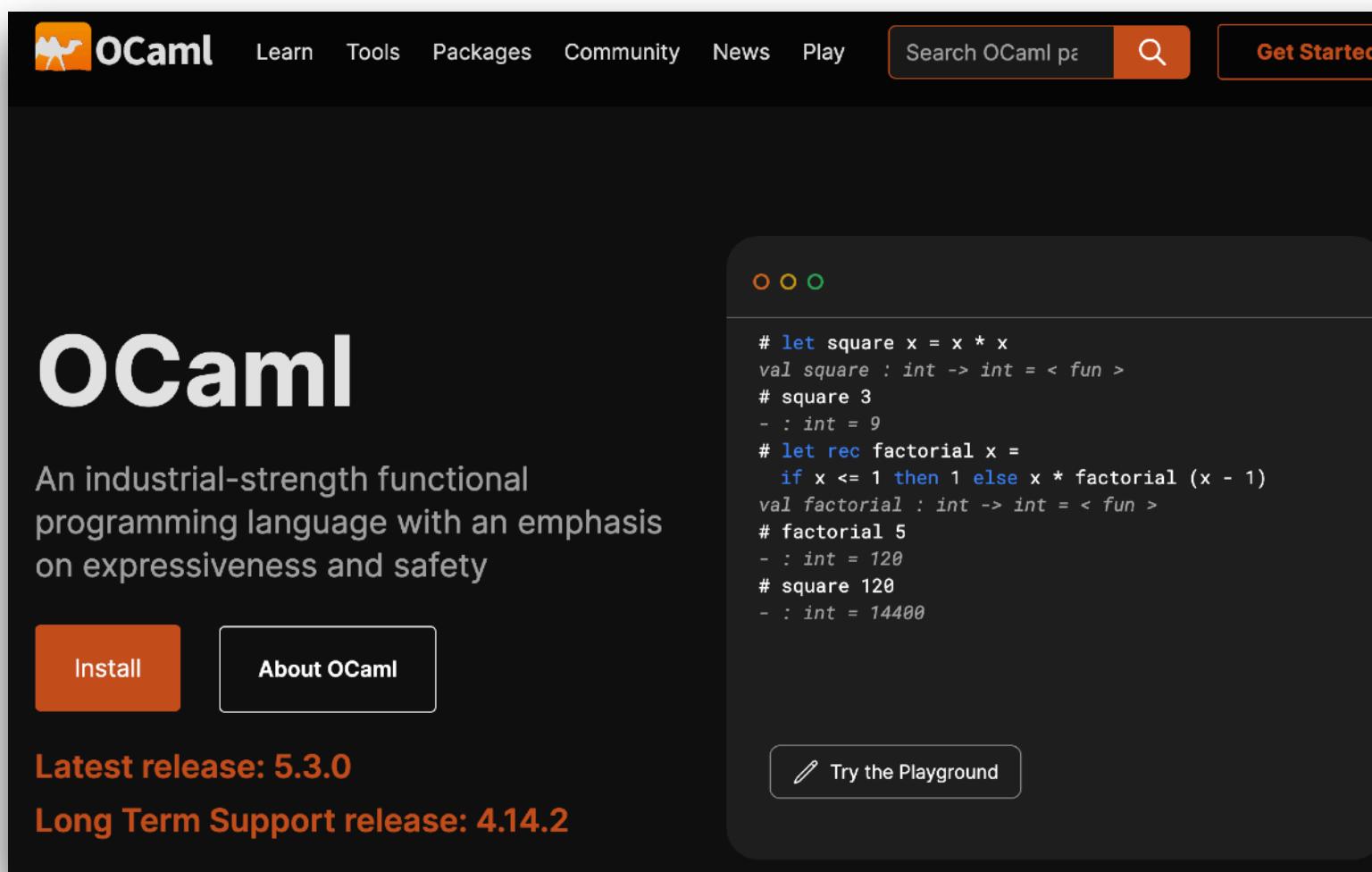


The screenshot shows a screenshot of the OCaml Discord server. The left sidebar lists channels: Boost Goal, TEO, +, OCaml, # general (selected), # beginners, # advanced-help, # débutants, # forum, # share, and # offtopic. The main # general channel has a blue header bar with "50+ new messages since 9:55 PM on August 15, 2025". It shows messages from CCBot, lukstafi, and 78CraintQueDieu. The messages discuss OCaml, Haskell, and Liquidsoap.

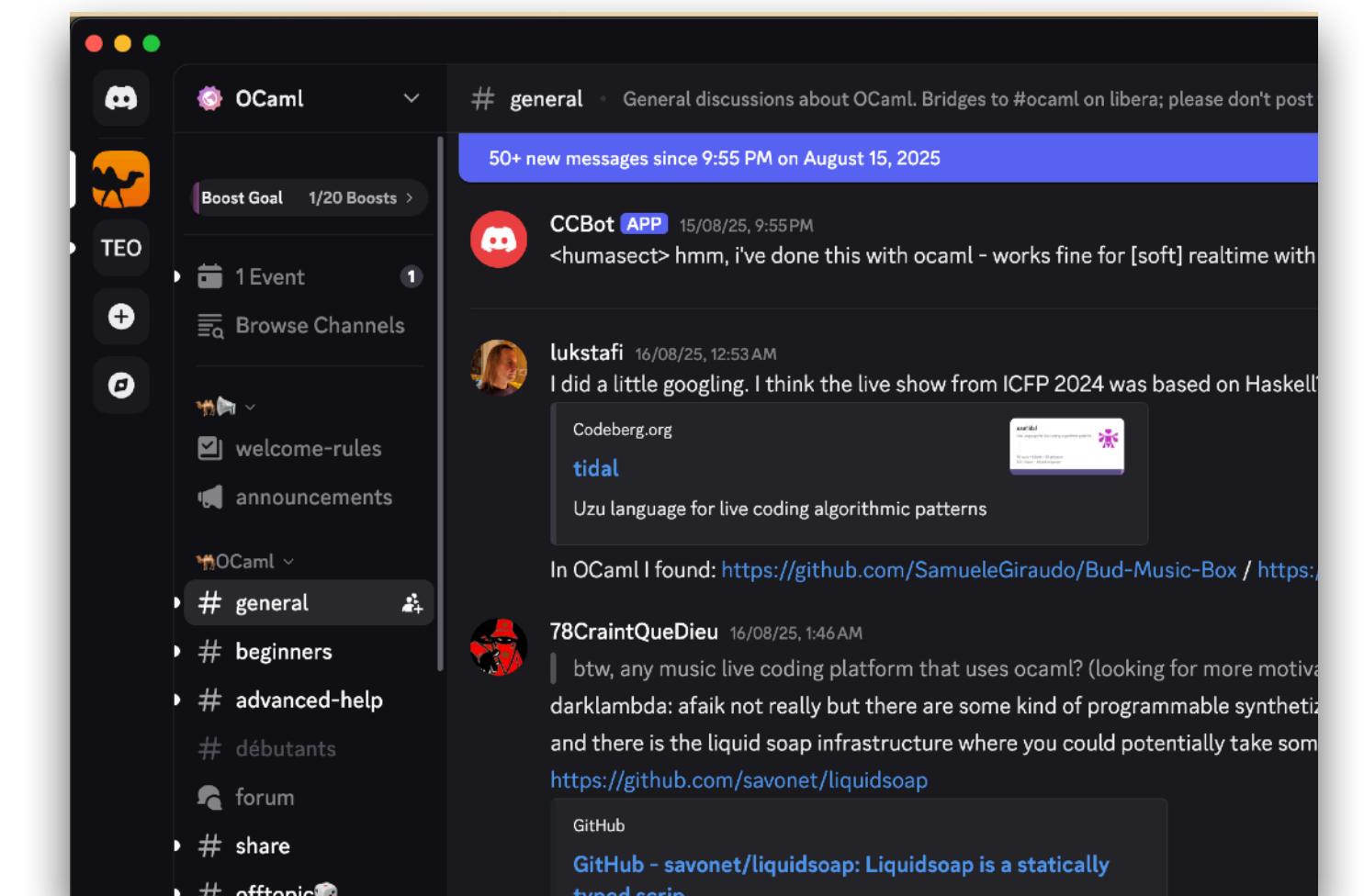
OCaml
Discord

Get Involved!

ocaml.org



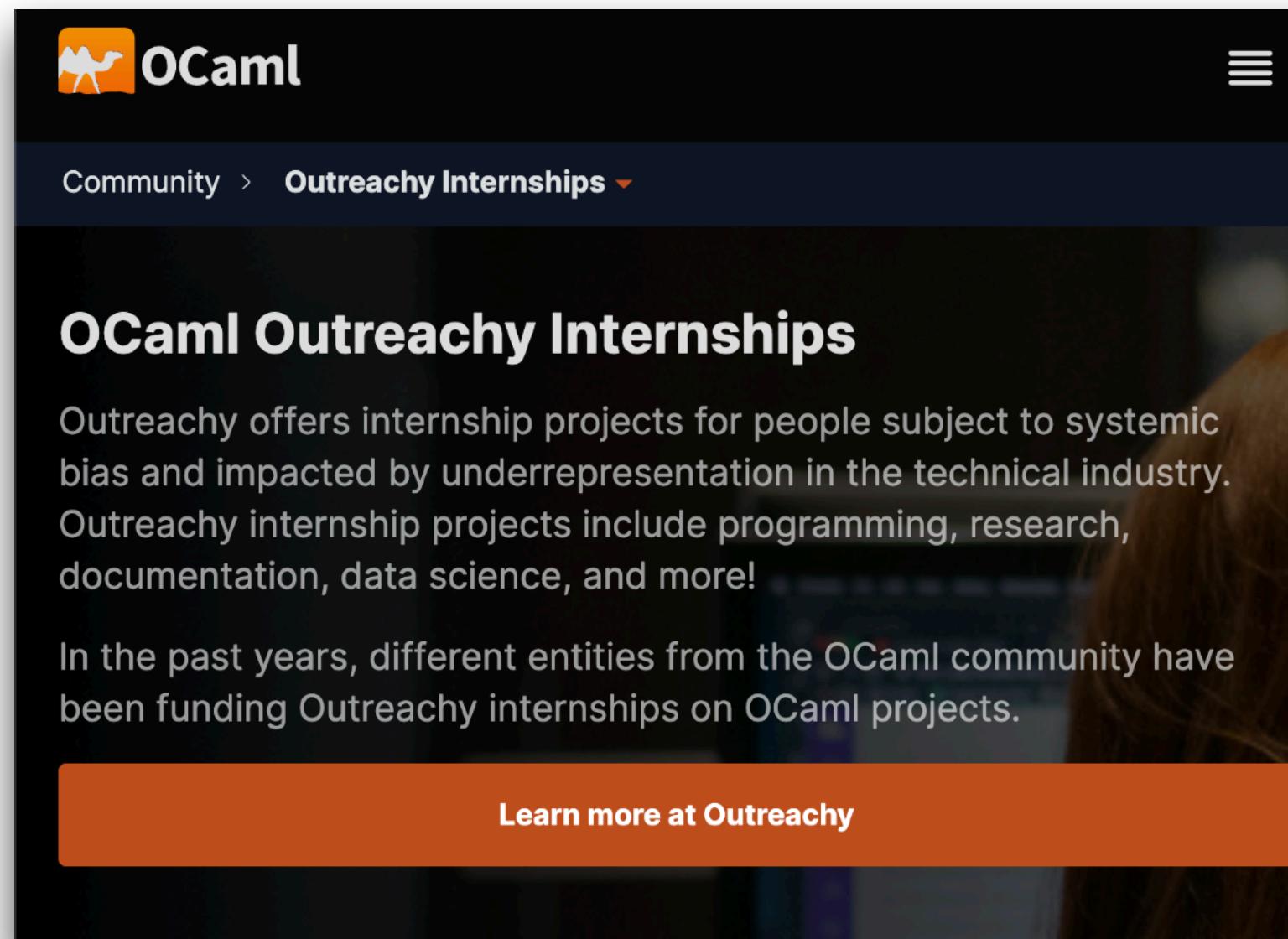
The screenshot shows the OCaml website homepage. The header includes the OCaml logo, navigation links for Learn, Tools, Packages, Community, News, Play, a search bar, and a "Get Started" button. The main content features the OCaml logo, a brief description of the language, and two buttons: "Install" and "About OCaml". Below this, it shows the "Latest release: 5.3.0" and "Long Term Support release: 4.14.2". A code editor window displays a snippet of OCaml code for calculating the factorial of a number. A "Try the Playground" button is at the bottom of the code editor.



The screenshot shows the OCaml Discord server interface. The left sidebar lists channels: Boost Goal, TEO, +, # general (selected), # beginners, # advanced-help, # débutants, # forum, # share, and # offtopic. The main # general channel has a message from CCBot (@CCBot) at 15/08/25, 9:55 PM. Other messages from users like lukstafi and 78CrainQueDieu are visible, along with links to GitHub repositories like Codeberg.org and tidal.

OCaml
Discord

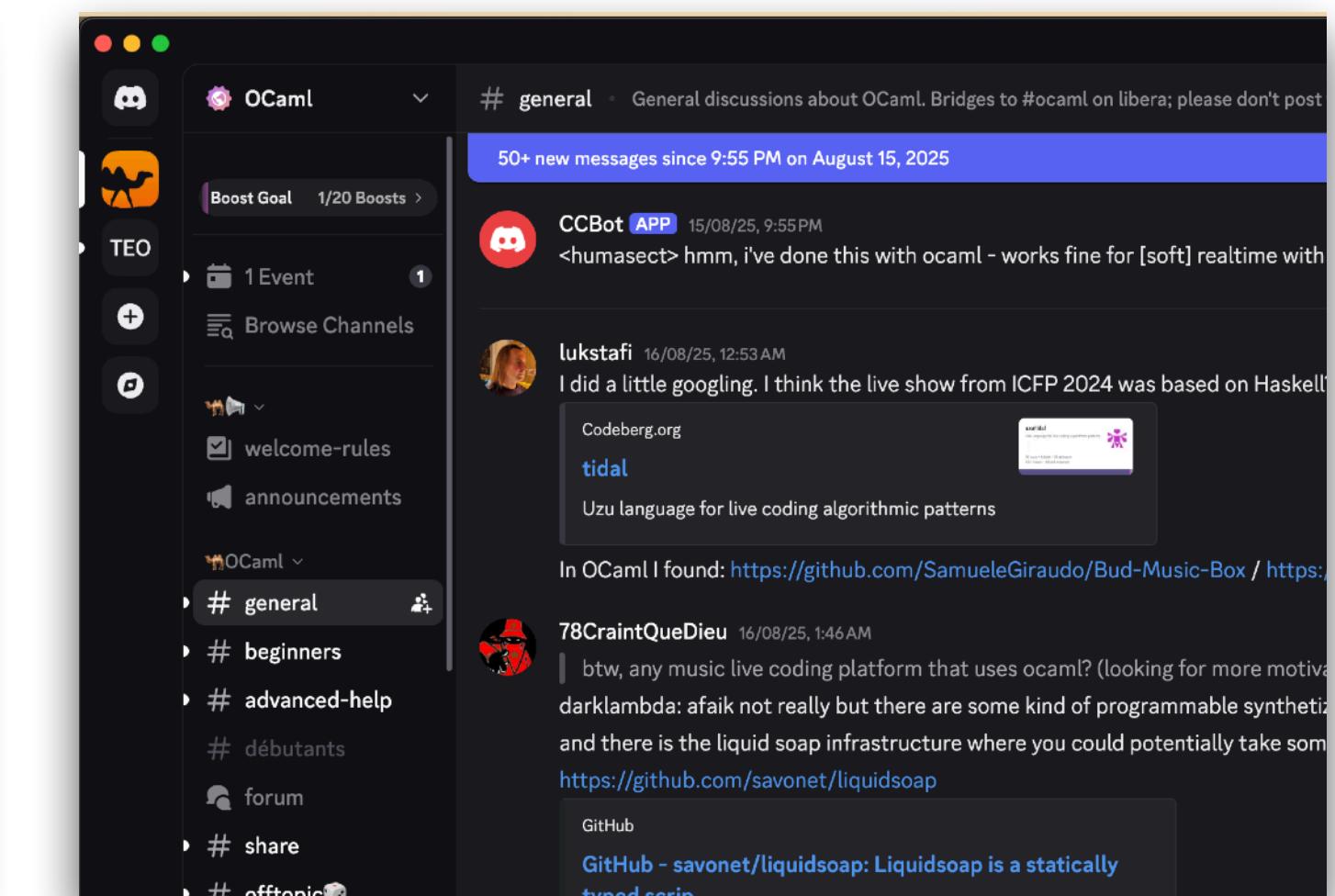
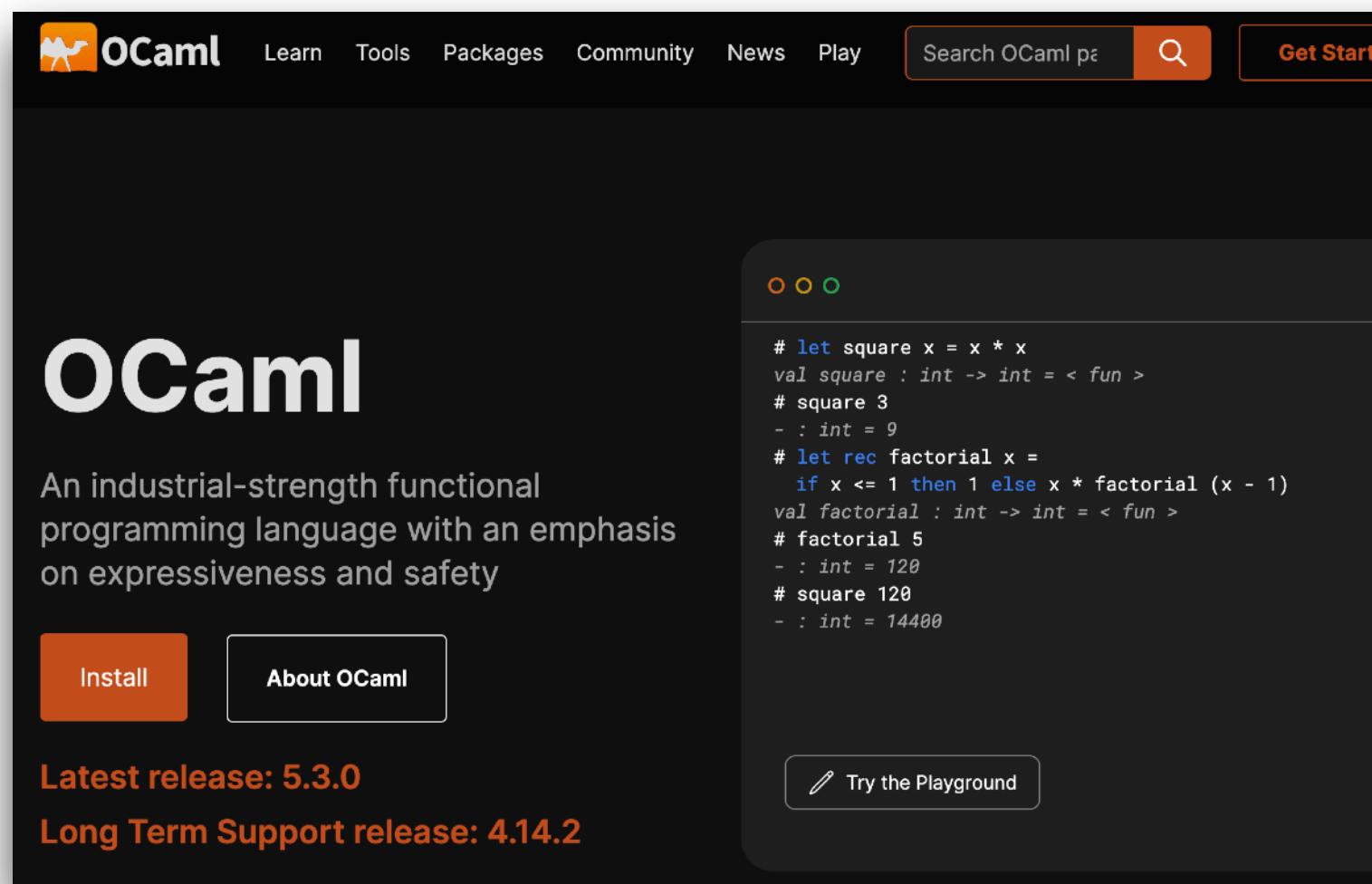
ocaml.org/outreachy



The screenshot shows the OCaml Outreachy Internships page. The header includes the OCaml logo and a navigation bar with "Community" and "Outreachy Internships". The main content features the title "OCaml Outreachy Internships" and a paragraph about Outreachy's mission to offer internships to underrepresented groups. It also mentions that Outreachy internships are funded by the OCaml community. A "Learn more at Outreachy" button is at the bottom.

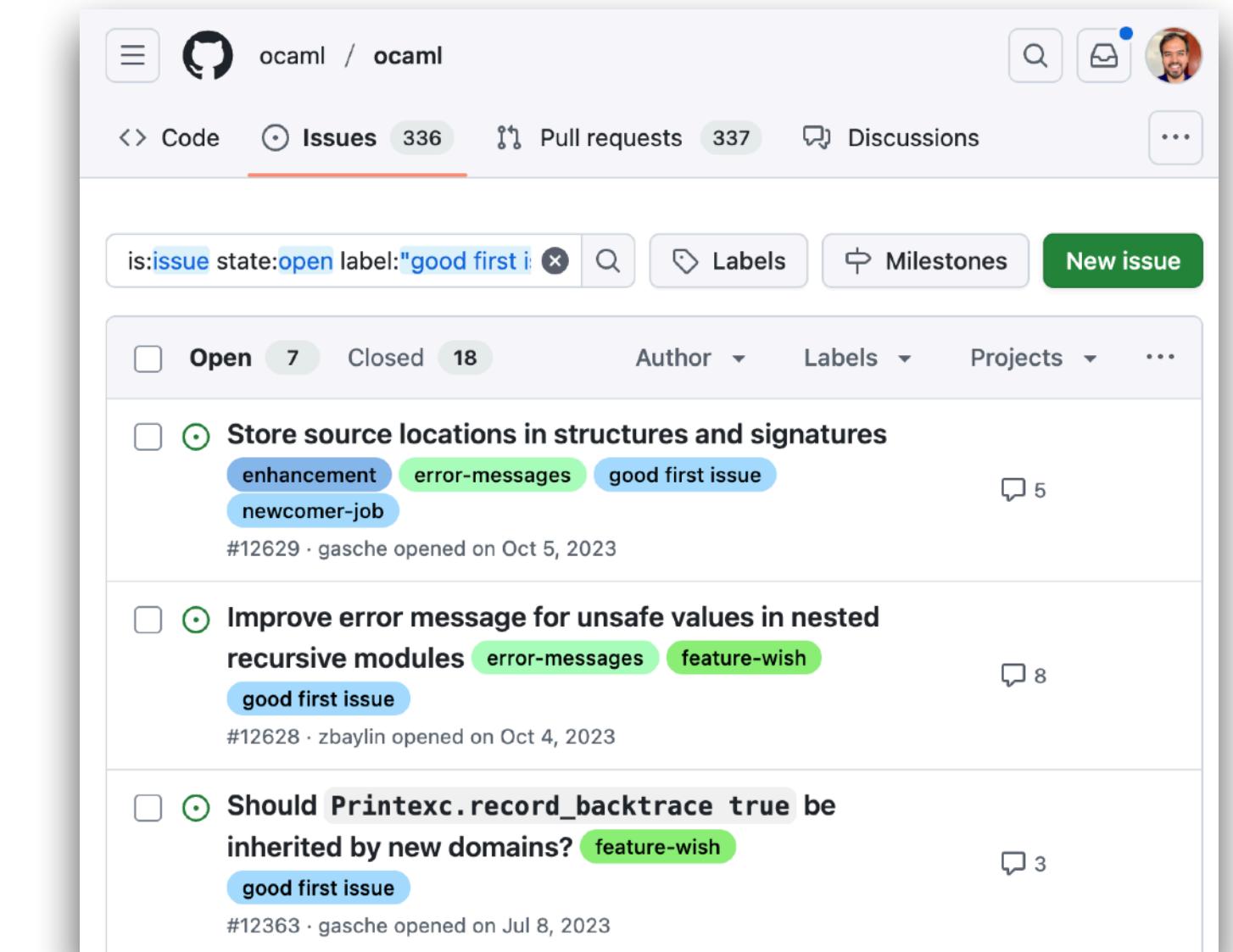
Get Involved!

ocaml.org



OCaml
Discord

ocaml.org/outreachy



github.com/ocaml