PyCon US · 24

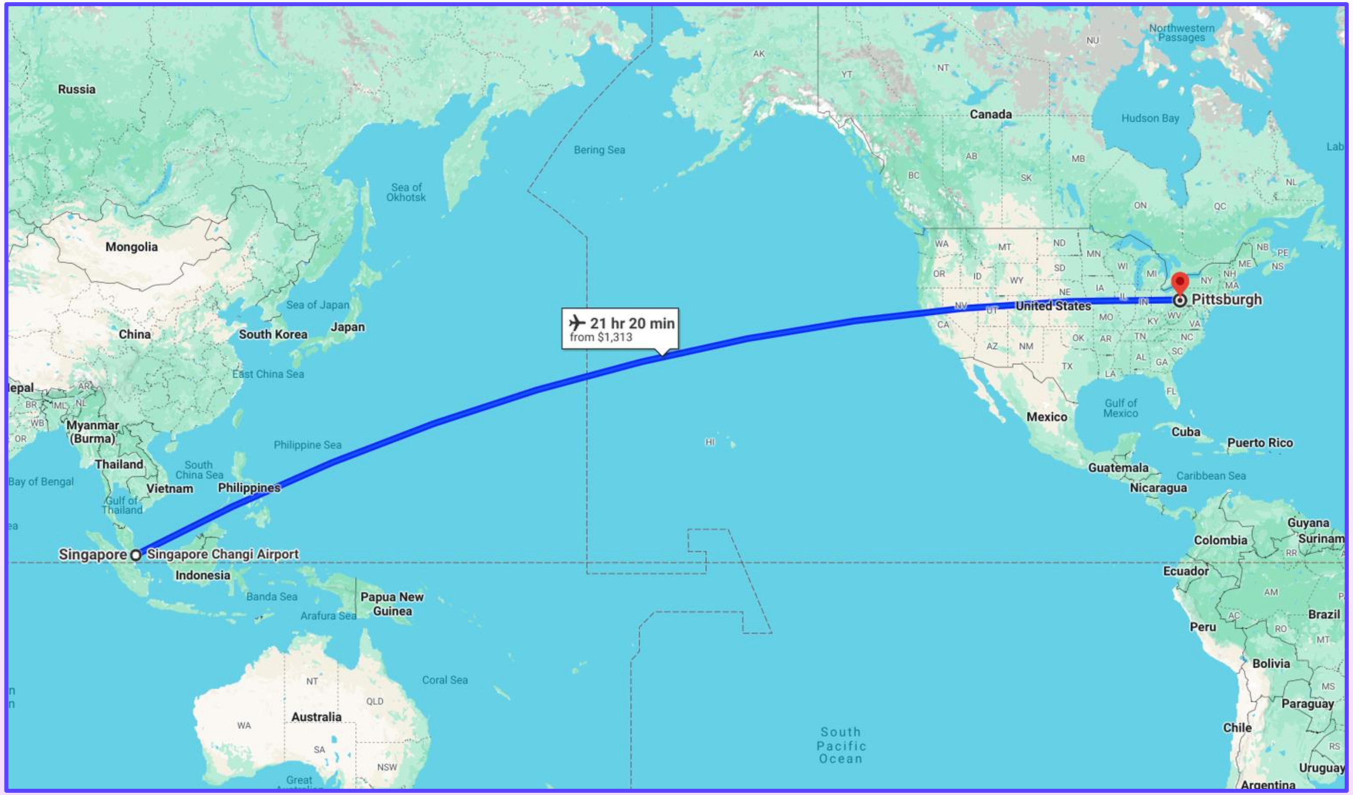
By the community, for the community

How two undergrads from the other side of the planet are speeding up your future code

Ken Jin, Jules Poon

Introduction

The Other Side of the Planet



Introductions

Ken Jin:

- Undergrad student
- CPython Core Developer since 2021
- (Part-time) SWE at Quansight Labs

Jules:

- Math undergrad
- Interested in commutative algebra and programming languages

Goal


Our Goal
CPython Bytecode
Project Traces
Uop Trace
Trace Optimisation
Optimised Trace
JIT Compiler
Machine Code

Timeline

* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Timeline

Ken Jin is an external collaborator to the Faster CPython Team

< 2022

Jan 2023

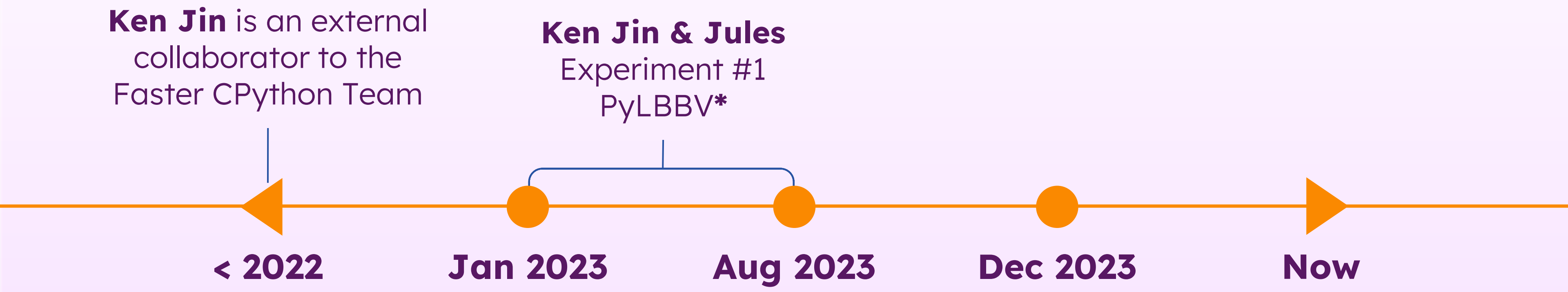
Aug 2023

Dec 2023

Now

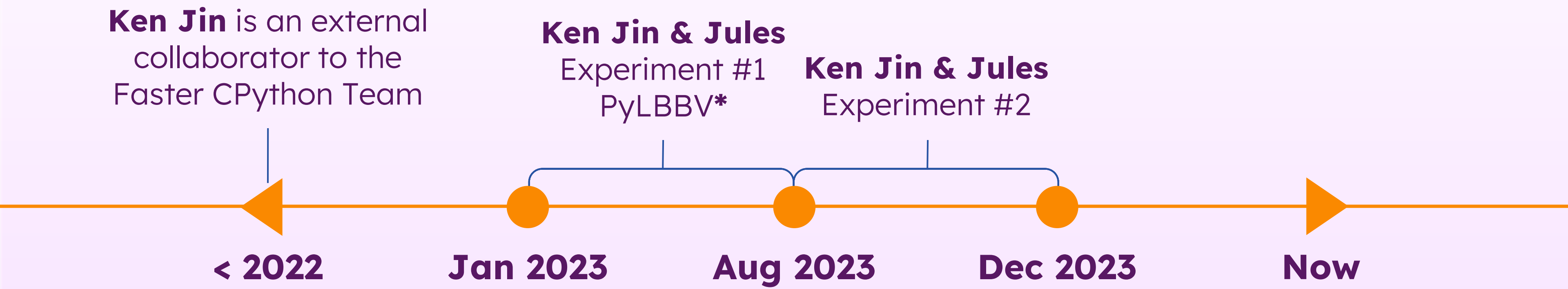
* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Timeline



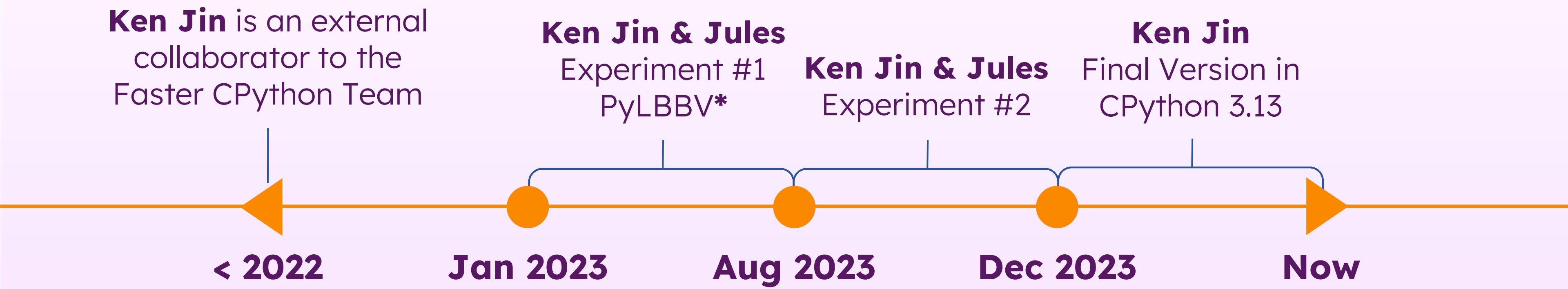
* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Timeline



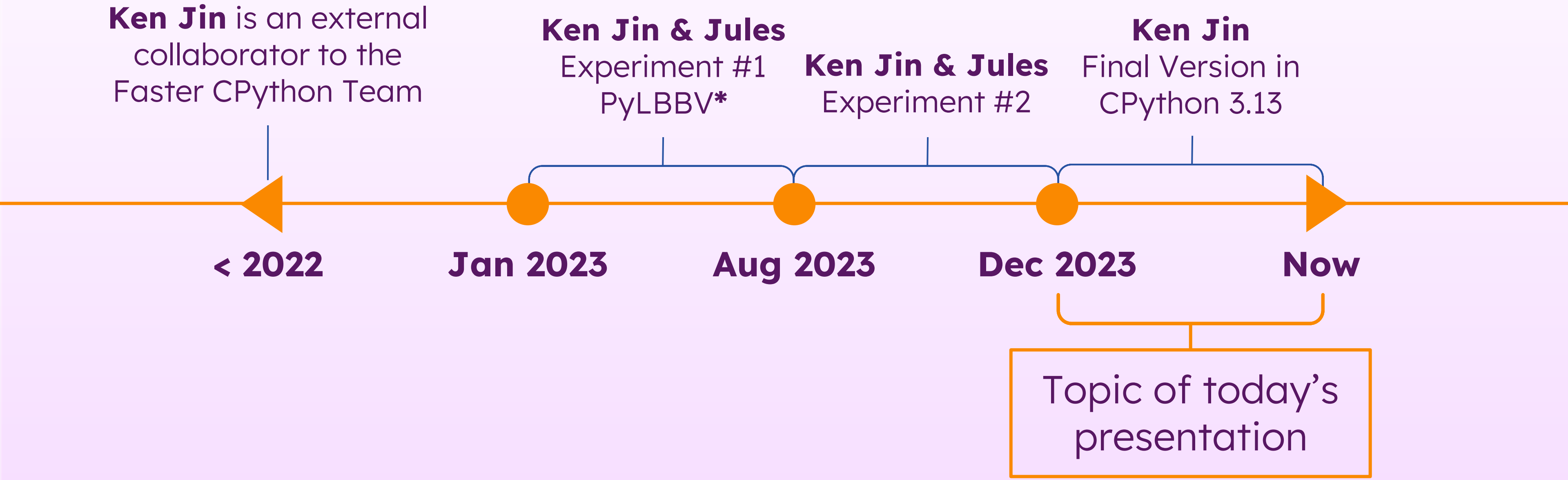
* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Timeline



* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Timeline



* CPython with lazy basic block versioning (Maxime Chevalier-Boisvert and Marc Feeley, 2014).

Background

CPython

Written in C

CPython

Reference implementation of Python

CPython: Bytecode Stack Machine

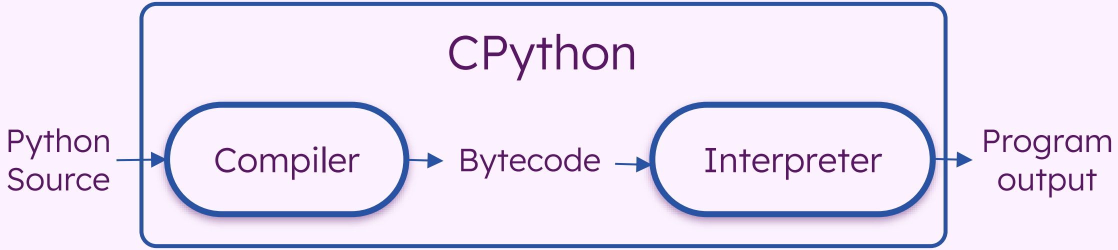
Bytecode Stack Machine

CPython: Bytecode Stack Machine

Bytecode Stack Machine

An instruction set for easy interpretation by the *interpreter*

- **Bytecode** is easier to interpret compared to **Python source**
- A *compiler* converts **Python source** to **Bytecode**



CPython: Bytecode Stack Machine

Bytecode Stack Machine

A linear data structure containing objects

- Last In, Last Out

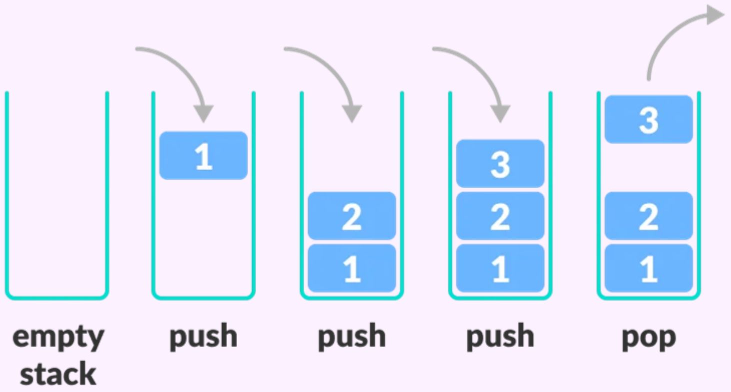


Image from: www.programiz.com/dsa/stack

CPython: Bytecode Stack Machine

Bytecode Stack Machine

CPython interpreter uses the **stack** to store its intermediate results.

- CPython's Bytecode largely instructs how to manipulate data on the **stack**.

CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack

CPython: Bytecode Stack Machine

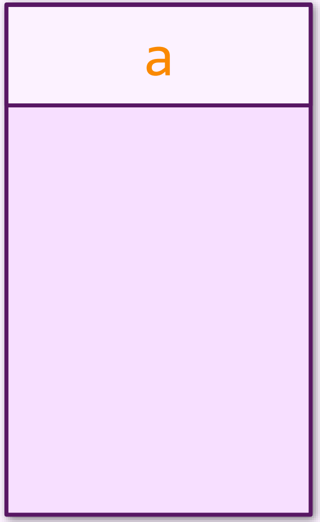
Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack



CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack

a
b

CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack

a+b

CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack

a+b

c

CPython: Bytecode Stack Machine

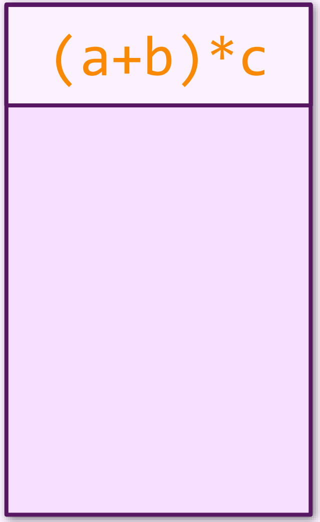
Bytecode Stack Machine

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Stack



CPython: 3.11 Specialising Interpreter

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

Generic:

- `a`, `b`, `c` can be `str`, `int`, `float` or even objects!
- `BINARY_OP` has to perform dynamic type dispatch→ **Slow!**

CPython: 3.11 Specialising Interpreter (Tier 1)

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

CPython: 3.11 Specialising Interpreter (Tier 1)

`a: int, b: int, c: int`

Expression: `(a + b) * c`

Compiled:

```
 2 LOAD_GLOBAL    0 (a)
14 LOAD_GLOBAL    2 (b)
26 BINARY_OP      0 (+)
30 LOAD_GLOBAL    4 (c)
42 BINARY_OP      5 (*)
```

CPython: 3.11 Specialising Interpreter (Tier 1)

a: int, b: int, c: int

Expression: (a + b) * c

Compiled:

2 LOAD_GLOBAL 0 (a)

14 LOAD_GLOBAL 2 (b)

~~26 BINARY_OP 0 (+)~~ BINARY_OP_ADD_INT

30 LOAD_GLOBAL 4 (c)

~~42 BINARY_OP 5 (*)~~ BINARY_OP_MULTIPLY_INT

CPython: 3.11 Specialising Interpreter (Tier 1)

`a: int, b: int, c: int`

Expression: `(a + b) * c`

Compiled:

~~2 LOAD_GLOBAL 0 (a)~~	LOAD_GLOBAL_MODULE
~~14 LOAD_GLOBAL 2 (b)~~	LOAD_GLOBAL_MODULE
~~26 BINARY_OP 0 (+)~~	BINARY_OP_ADD_INT
~~30 LOAD_GLOBAL 4 (c)~~	LOAD_GLOBAL_MODULE
~~42 BINARY_OP 5 (*)~~	BINARY_OP_MULTIPLY_INT

CPython: 3.11 Specialising Interpreter (Tier 1)

https://youtu.be/shQtrn1v7sQ?si=2BT_V5JiOzwL1wzg

CPython 3.11 → 3.13 and onwards

CPython 3.11:

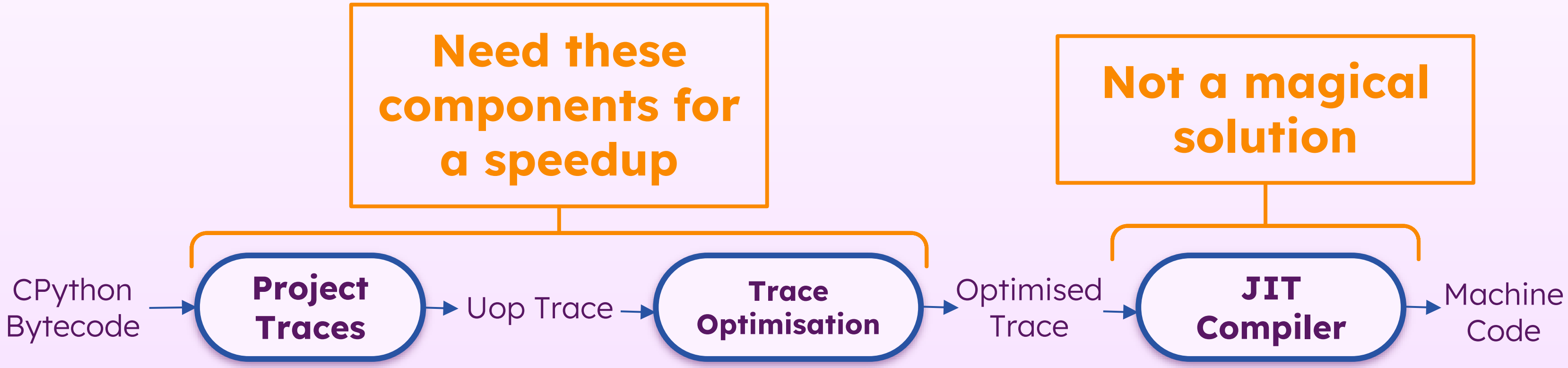
- Specialising Interpreter optimizes across one to two bytecode

CPython 3.13 and onwards:

- Learn commonly encountered types at runtime to optimize across larger regions
- Not a new idea, difficulty is implementing correctly and safely and in a maintainable way

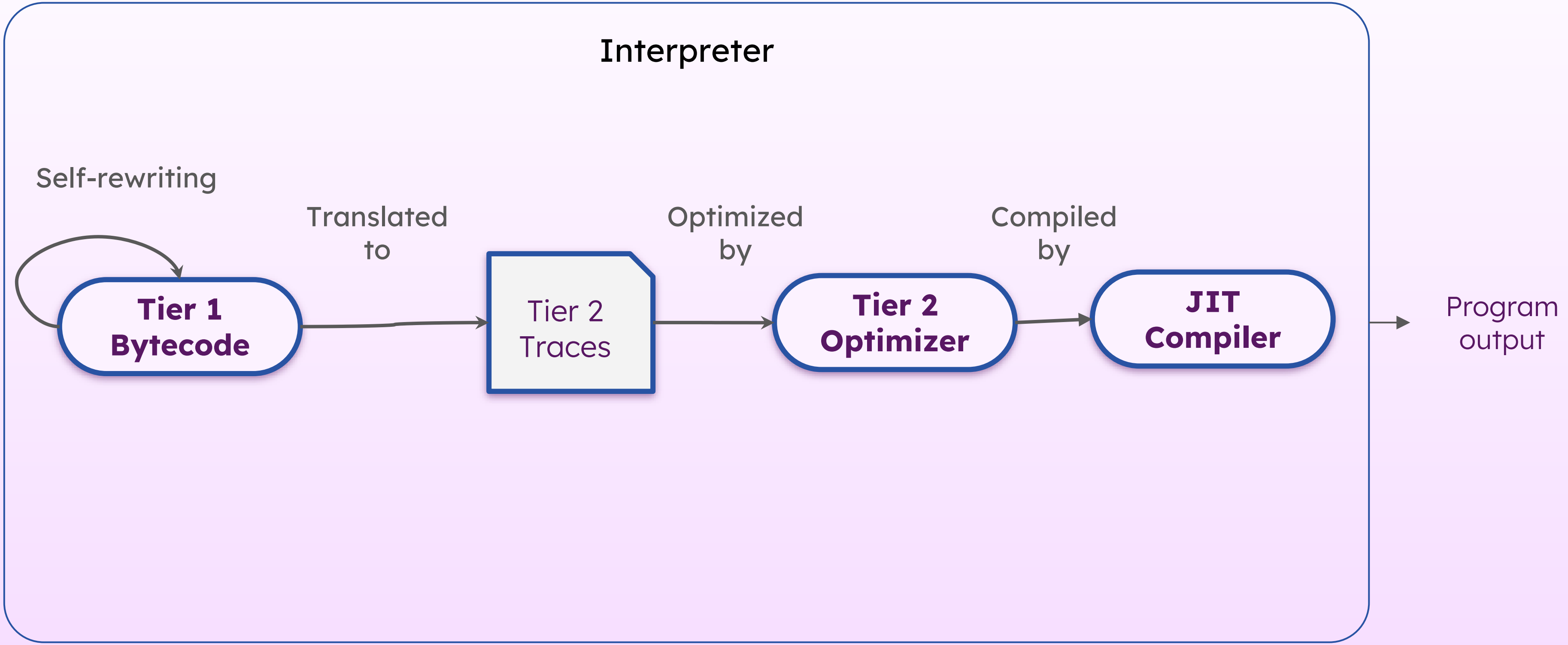
CPython 3.13 and onwards

CPython 3.13



Need these components for a speedup
Not a magical solution
CPython Bytecode
Project Traces
Uop Trace
Trace Optimisation
Optimised Trace
JIT Compiler
Machine Code

CPython 3.13 Interpretation Pipeline




CPython Bytecode
Project Traces
Uop Trace
Trace Optimisation
Optimised Trace
JIT Compiler
Machine Code

CPython 3.13

Project Traces

Python Source

```python
for i in
range(128):
    a += b
```

Specialised Bytecode (Tier 1)

```
GET_ITER
FOR_ITER_RANGE 7
STORE_FAST           2
LOAD_FAST_LOAD_FAST  1
BINARY_OP_ADD_INT   13
STORE_FAST           0
JUMP_BACKWARD 9
END_FOR
POP_TOP
RETURN_CONST 0
```

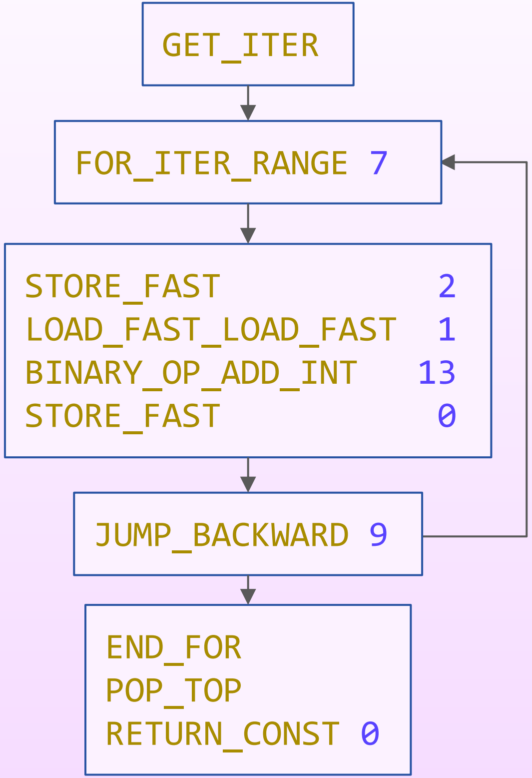
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted

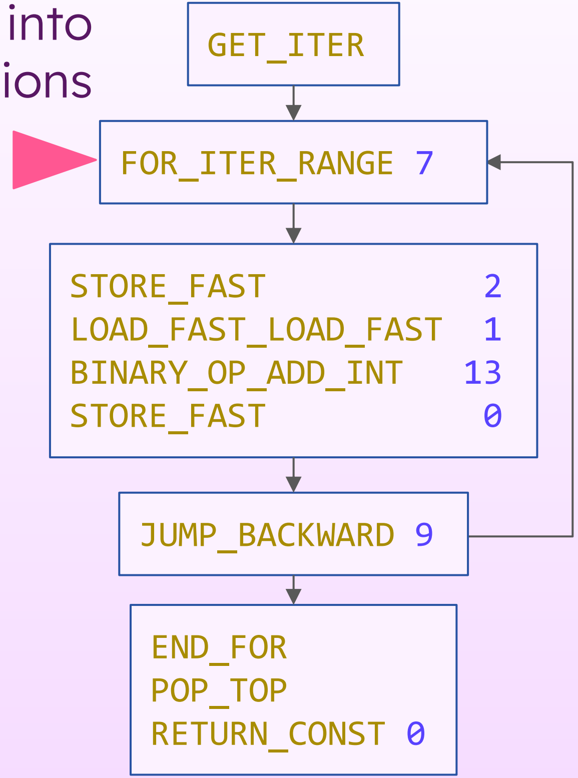
```
GET_ITER

FOR_ITER_RANGE 7

STORE_FAST            2
LOAD_FAST_LOAD_FAST   1
BINARY_OP_ADD_INT    13
STORE_FAST            0

JUMP_BACKWARD 9

END_FOR
POP_TOP
RETURN_CONST 0
```

Trace starts at backwards edge

CPython 3.13

Project Traces

Bytecode gets broken down into smaller operations



Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
```

Note:
_CHECK_VALIDITY_AND_SET_IP omitted

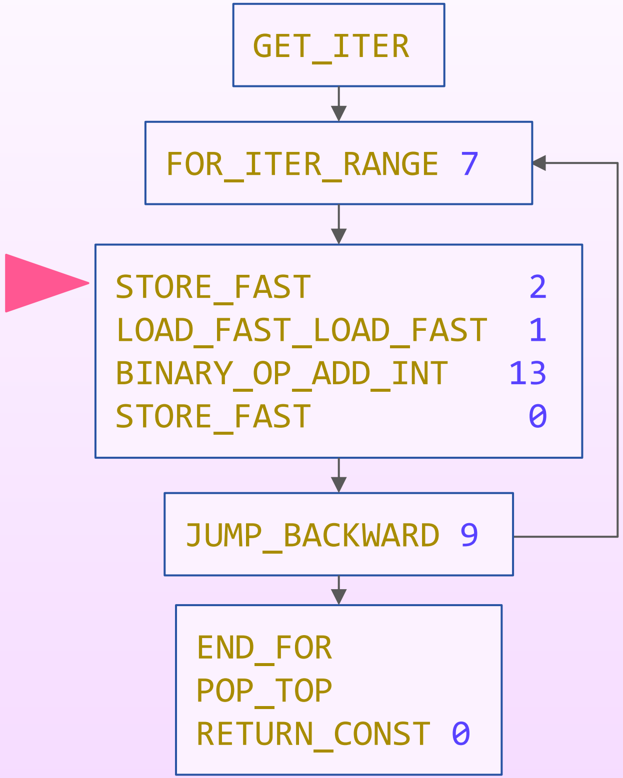
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE          7
_GUARD_NOT_EXHAUSTED_RANGE 7
_ITER_NEXT_RANGE           7
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



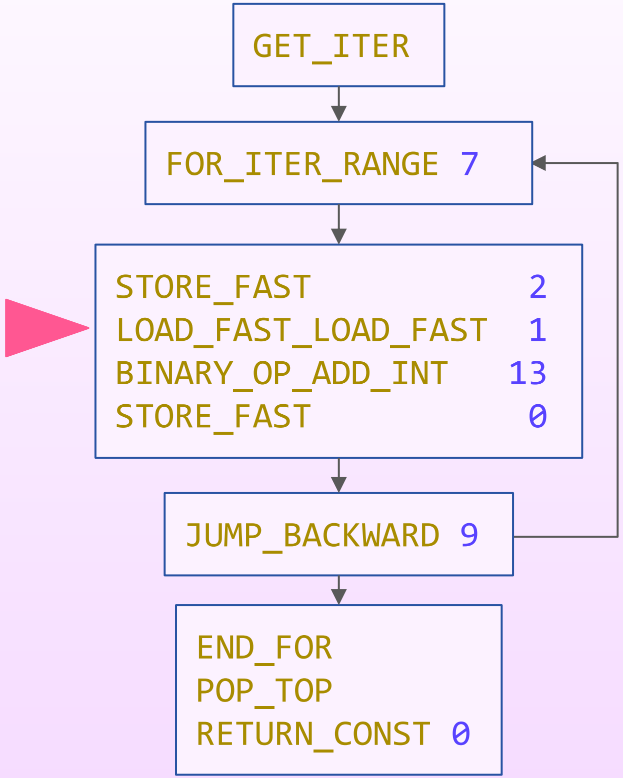
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE           7
_GUARD_NOT_EXHAUSTED_RANGE  7
_ITER_NEXT_RANGE            7
_STORE_FAST                 2
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



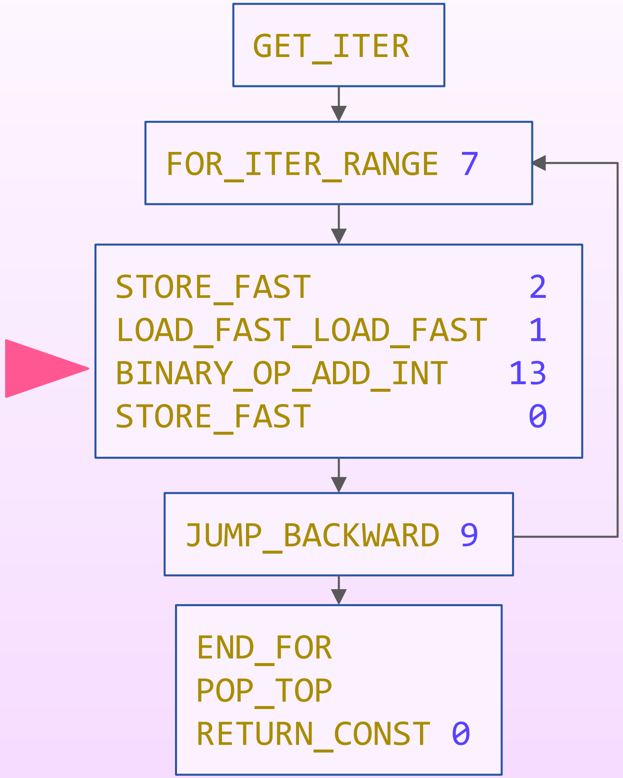
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE              7
_GUARD_NOT_EXHAUSTED_RANGE     7
_ITER_NEXT_RANGE               7
_STORE_FAST                    2
_LOAD_FAST                     0
_LOAD_FAST                     1
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



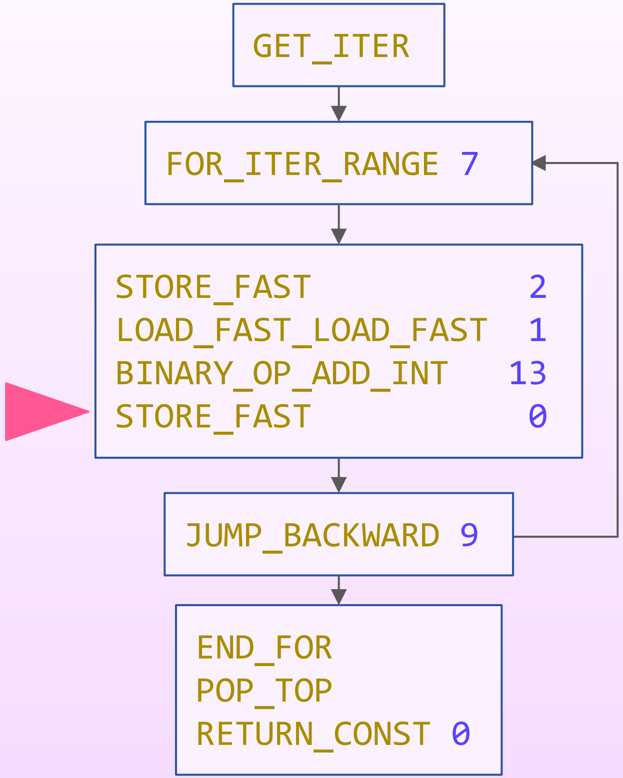
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE            7
_GUARD_NOT_EXHAUSTED_RANGE   7
_ITER_NEXT_RANGE             7
_STORE_FAST                  2
_LOAD_FAST                   0
_LOAD_FAST                   1
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



CPython 3.13

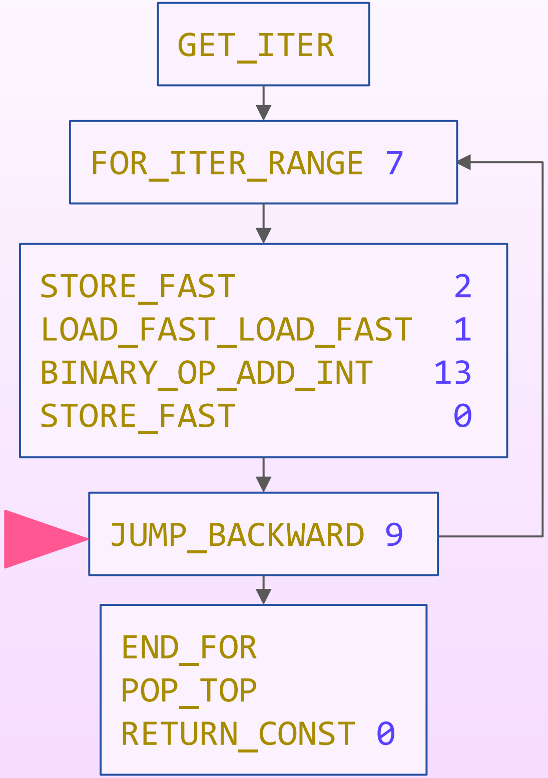
Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE           7
_GUARD_NOT_EXHAUSTED_RANGE  7
_ITER_NEXT_RANGE            7
_STORE_FAST                 2
_LOAD_FAST                  0
_LOAD_FAST                  1
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_STORE_FAST                 0
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted

Trace ends when loop closes



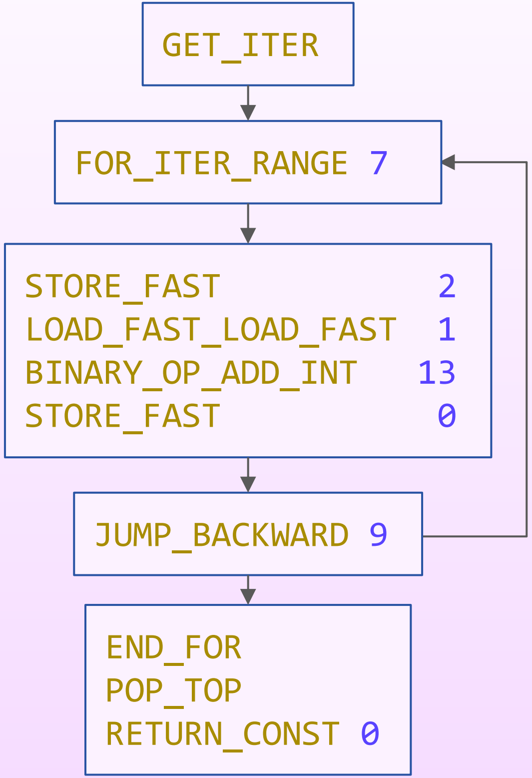
CPython 3.13

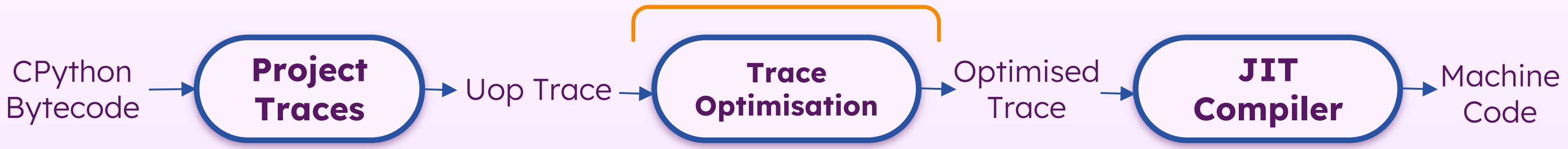
Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_ITER_CHECK_RANGE            7
_GUARD_NOT_EXHAUSTED_RANGE   7
_ITER_NEXT_RANGE             7
_STORE_FAST                  2
_LOAD_FAST                   0
_LOAD_FAST                   1
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_STORE_FAST                  0
_JUMP_TO_TOP
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted




CPython Bytecode
Project Traces
Uop Trace
Trace Optimisation
Optimised Trace
JIT Compiler
Machine Code

CPython 3.13 and Beyond

Trace Optimisation

First pass: By Mark Shannon

- Promoting **globals** to **constants** (3.13)

Second pass: By Ken Jin, with contributions from Mark Shannon, Guido van Rossum, Peter Lazorchak

- Guard Elimination (3.13 partially implemented)
- True Function Inlining (WIP)
- Deferred Object Creation (WIP)
- Register allocation/TOS caching (WIP)

Analysis via **Abstract Interpretation** (3.13)

CPython 3.13 and Beyond

Trace Optimisation

Second pass: Abstract Interpretation (3.13)

Normal interpretation: Operate on values

Abstract interpretation: Operate on abstractions of values

- In CPython 3.13, the abstraction is (mostly) the type of the value/object

Problem:

Need to maintain two largely disconnected interpretation specifications

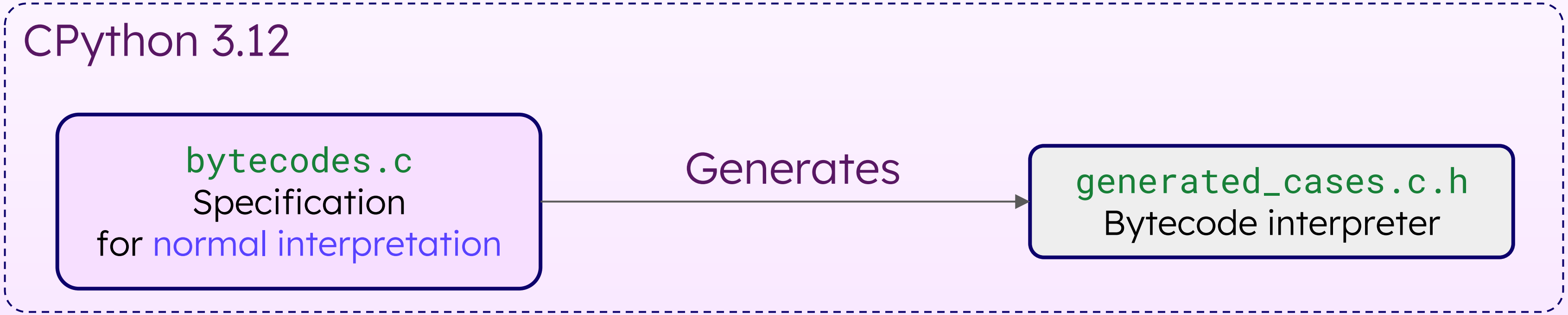
Solution:

CPython 3.12 introduced a DSL to specify the operations of bytecode

CPython 3.13 and Beyond

Trace Optimisation

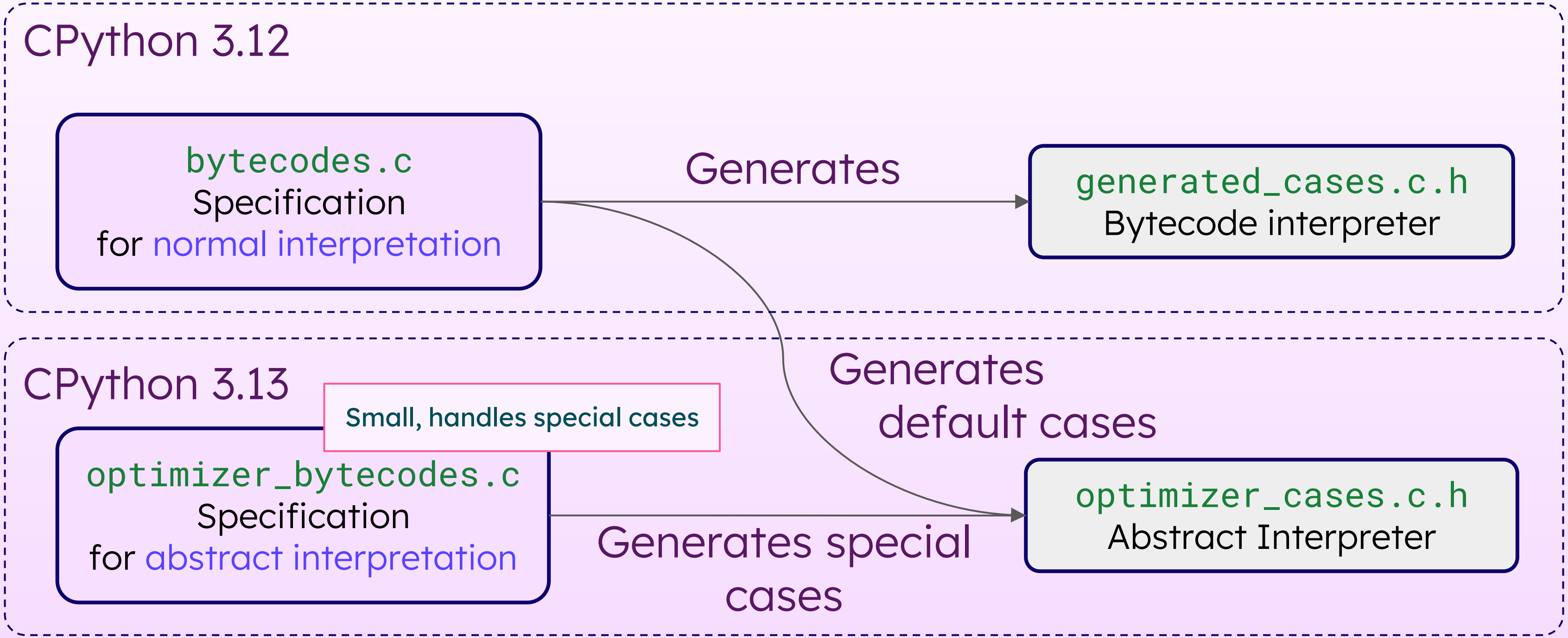
Second pass: Abstract Interpretation



CPython 3.13 and Beyond

Trace Optimisation

Second pass: Abstract Interpretation



CPython 3.13 and Beyond

Trace Optimisation

Second pass: Guard Elimination (3.13 partially implemented)

Abstract Interpretation learns the types of each variable

Python Source

```
a += b + b + b
```

Trace

```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT    # Compute `(b+b)+b`
_GUARD_BOTH_INT       # Check `a` and `(b+b(+b` are int
_BINARY_OP_ADD_INT    # Compute `a+((b+b)+b)`
_STORE_FAST         1 # Store result in `a`
```

CPython 3.13 and Beyond

Trace Optimisation

Second pass: Guard Elimination (3.13 partially implemented)

Abstract Interpretation learns the types of each variable

Python

```
a += b + b + b
```

We know b is int

Trace

```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT    # Compute `(b+b)+b`
_GUARD_BOTH_INT       # Check `a` and `(b+b(+b` are int
_BINARY_OP_ADD_INT    # Compute `a+((b+b)+b)`
_STORE_FAST         1 # Store result in `a`
```

CPython 3.13 and Beyond

Trace Optimisation

Second pass: Guard Elimination (3.13 partially implemented)

Abstract Interpretation learns the types of each variable

Python

```
a += b + b + b
```

We know b is int

Not needed

Trace

```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT    # Compute `(b+b)+b`
_GUARD_BOTH_INT       # Check `a` and `(b+b(+b` are int
_BINARY_OP_ADD_INT    # Compute `a+((b+b)+b)`
_STORE_FAST         1 # Store result in `a`
```

CPython 3.13 and Beyond

Trace Optimisation

Second pass: Guard Elimination (3.13 partially implemented)

Abstract Interpretation learns the types of each variable

Python

```python
a += b + b + b
```

We know b is int

Not needed

Need to check a only

Trace

```
_LOAD_FAST           1 # Load `a`
_LOAD_FAST           0 # Load `b`
_LOAD_FAST           0 # Load `b`
_GUARD_BOTH_INT        # Check `b` and `b` are int
_BINARY_OP_ADD_INT     # Compute `b+b`
_LOAD_FAST           0 # Load `b`
_GUARD_BOTH_INT        # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT     # Compute `(b+b)+b`
_GUARD_NOS_INT         # Check `a` and `(b+b)+b` are int
_BINARY_OP_ADD_INT     # Compute `a+((b+b)+b)`
_STORE_FAST          1 # Store result in `a`
```

CPython 3.14 and Beyond

Trace Optimisation

Second pass: True Function Inlining (WIP)

Currently worked on by Ken Jin.

Problem:

Function calls have some overhead (E.g., Creating a new frame)

Idea:

Inline the function as if it is one big function

CPython 3.14 and Beyond

Trace Optimisation

Second pass: True Function Inlining (WIP)

Python Source

```python
def g(b):
    return b+b
def f(b):
    for _ in range(128):
        g(b)
```

Trace

```
_CHECK_FUNCTION_EXACT_ARGS 1
_CHECK_STACK_SPACE_OPERAND
_INIT_CALL_PY_EXACT_ARGS   1 # Create frame
_SAVE_RETURN_OFFSET        4
_PUSH_FRAME                  # “Activate” frame
_RESUME_CHECK
_LOAD_FAST                 0
_LOAD_FAST                 0
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_POP_FRAME                   # “Deactivate” frame
```

CPython 3.14 and Beyond

Trace Optimisation

Second pass: True Function Inlining (WIP)

Python Source

```python
def g(b):
    return b+b
def f(b):
    for _ in range(128):
        g(b)
```

Trace

```
_CHECK_FUNCTION_EXACT_ARGS 1
_CHECK_STACK_SPACE_OPERAND
_INIT_CALL_PY_EXACT_ARGS    1 # Create frame
_SAVE_RETURN_OFFSET         4
_PUSH_FRAME                   # "Activate" frame
_RESUME_CHECK
_LOAD_FAST                  0
_LOAD_FAST                  0
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_POP_FRAME                    # "Deactivate" frame
```

Can these be optimized away?

CPython 3.14 and Beyond

Trace Optimisation

Second pass: Deferred Object Creation/Scalar Replacement
(WIP)

Idea from Mark Shannon: Defer or avoid entirely object creation if possible

E.g.,

- `[0,1,2,3][3]` returns `3` without creating a `list`
- `filter(lambda x: x%2, [1,2,3,4,5])` returns `filter` without creating an intermediate `list` literal

CPython 3.14 and Beyond

Trace Optimisation

Second pass: Register allocator/top of stack caching (WIP)

Currently worked on by Brandt Bucher.

Based on:

Ertl, M. A. (1995b). Stack caching for interpreters. SIGPLAN Not., 30(6), 315–327. doi:10.1145/223428.207165

Idea: Cache the top few items on the stack in registers

- Memory access is slow, register access is fast


CPython Bytecode
Project Traces
Uop Trace
Trace Optimisation
Optimised Trace
JIT Compiler
Machine Code

CPython 3.13 and Beyond

JIT Compiler

Talks: Building a JIT compiler for CPython

Sunday - May 19th, 2024 1 p.m.-1:30 p.m. in Ballroom A

Presented by:

- Brandt Bucher

Description

CPython is a programming language implementation that is mostly maintained by volunteers, but has a huge, diverse user base spread across a wide variety of platforms. These factors present a difficult set of challenges and tradeoffs when making design decisions, especially those related to just-in-time machine code generation.

As one of the engineers working on Microsoft's ambitious "Faster CPython" project, I'll introduce our prototype of "copy-and-patch", an interesting technique for generating high-quality template JIT compilers. Along the way, I'll also cover some of the important work in recent CPython releases that this approach builds upon, and how copy-and-patch promises to be an incredibly attractive tool for pushing Python's performance forward in a scalable, maintainable way.

CPython 3.13

Side Exits + De-Opts:

When

- The assumptions a trace made is invalid, or
  - (e.g., invalid cache/different runtime type encountered)
- Control-flow exits the trace,

CPython performs one of two exits:

1. Side Exits
2. De-Opts (De-Optimisation)

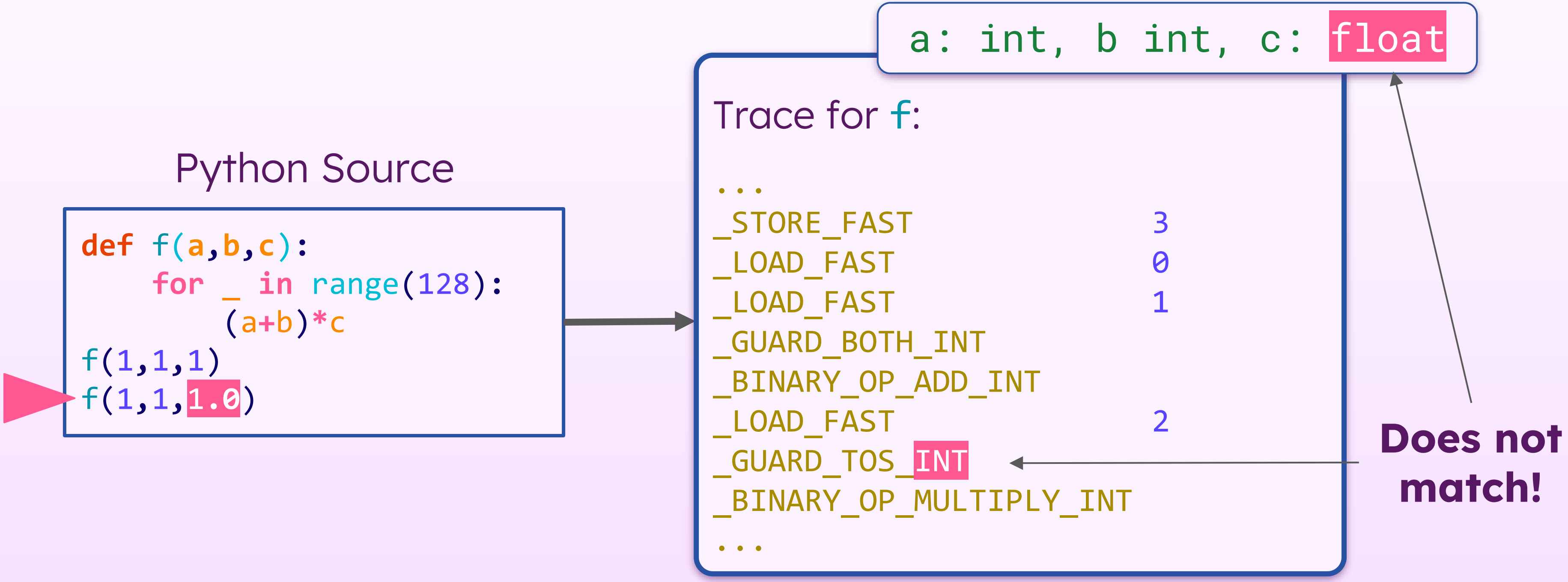
CPython 3.13

Side Exits:

If current progress of execution of the trace is still valid, a side exit either:

1. Jumps back to Tier 1
2. Creates a new trace corresponding to the side exit (if the side exit is taken enough times)
3. Jumps to an existing trace (if **2** has already happened)

CPython 3.13



Python Source
def f(a,b,c):
    for _ in range(128):
        (a+b)*c
f(1,1,1)
f(1,1,1.0)
a: int, b int, c: float
Trace for f:
...
_STORE_FAST 3
_LOAD_FAST 0
_LOAD_FAST 1
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_LOAD_FAST 2
_GUARD_TOS_INT
_BINARY_OP_MULTIPLY_INT
...
Does not match!

CPython 3.13 and Beyond

Side Exit!

```
_START_EXECUTOR
_SET_IP
_BINARY_OP      5 # Generic mult
_CHECK_VALIDITY
_POP_TOP
_EXIT_TRACE
```

`a: int, b int, c: float`

Trace for f:

```
...
_STORE_FAST                   3
_LOAD_FAST                    0
_LOAD_FAST                    1
_GUARD_BOTH_INT
_BINARY_OP_ADD_INT
_LOAD_FAST                    2
_GUARD_TOS_INT
_BINARY_OP_MULTIPLY_INT
...
```

Does not match!

CPython 3.13

De-Opts:

If continued execution of the trace is no longer valid (rare)

- Drop back to Tier 1

Metadata:

Open Source as a University Student

How did we come to all this?

Before University:

- Ken Jin started on CPython before university.

University Sem 1:

- We took Programming Language Implementation under Martin Henz.
- Martin Henz supervised our first Experiment #1

Year 1 Summer:

- Continued Experiment #1 as a credit bearing software engineering project

Subsequent Work:

- Manuel Rigger became our advisor for the next year

Lesson 1: Make full use of your university's mentorship as a student.

How did we come to all this?

Classes we took:

- Programming Language Implementation
- Compiler Design
- Principles of Program Analysis (Ken Jin)

Lesson 2: Take classes that help your journey

How did we come to all this?

- Jules and Ken Jin wrote 2 versions of the optimizer, Ken Jin then rewrote that 2 more times before it went into CPython.

Lesson 3: Be patient with yourself

That’s a wrap!

- Ken Jin will try to graduate.
- Jules has no plans.

Thank You!

@Fidget-Spinner | kenjin@python.org
@JuliaPoo | juliapoopoopoo@gmail.com