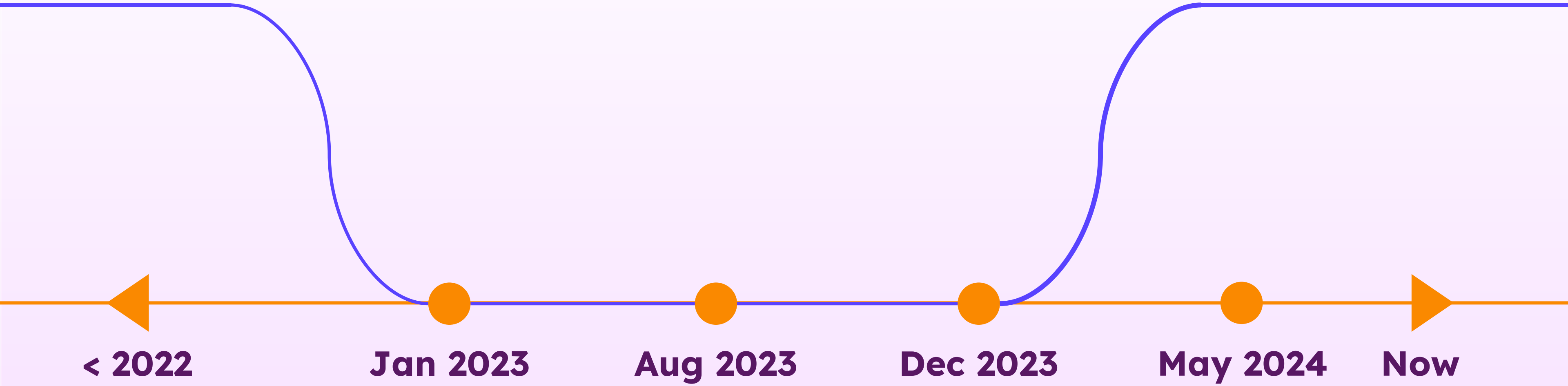
An overview of the optimisation pipeline in CPython 3.13 and onwards

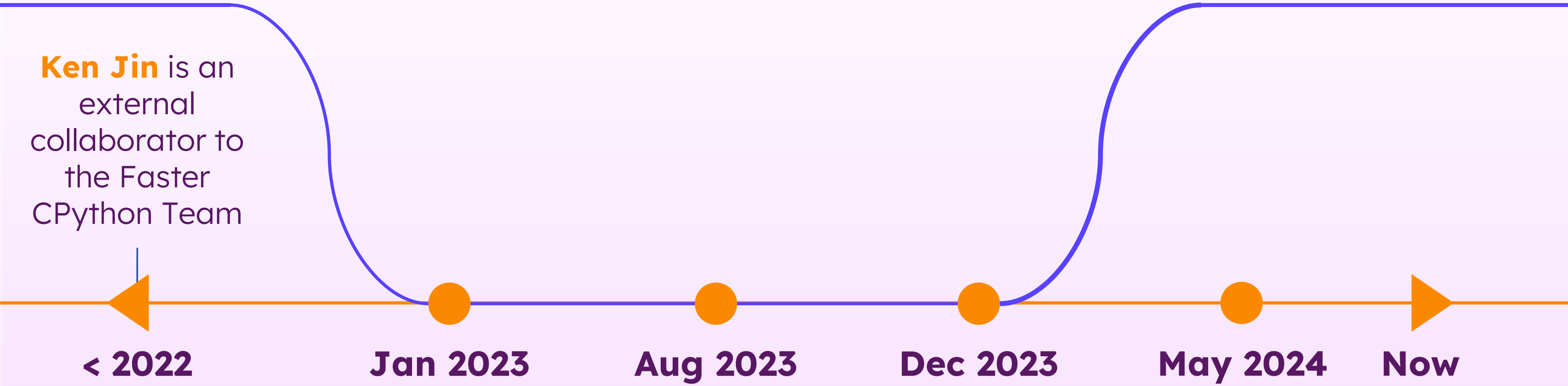
Ken Jin (not present), Jules Poon

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

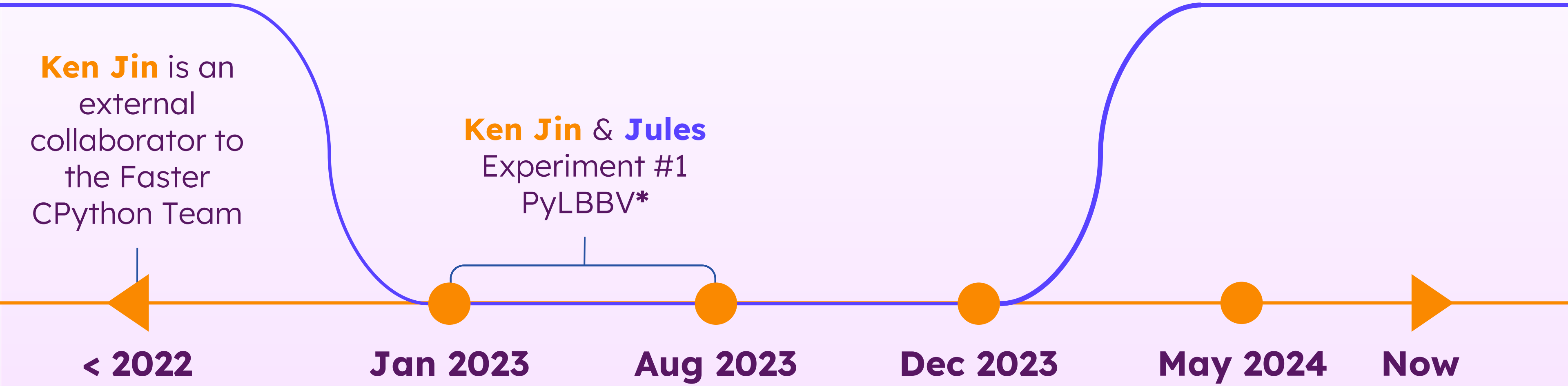
Dec 2023

May 2024

Now

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

Timeline

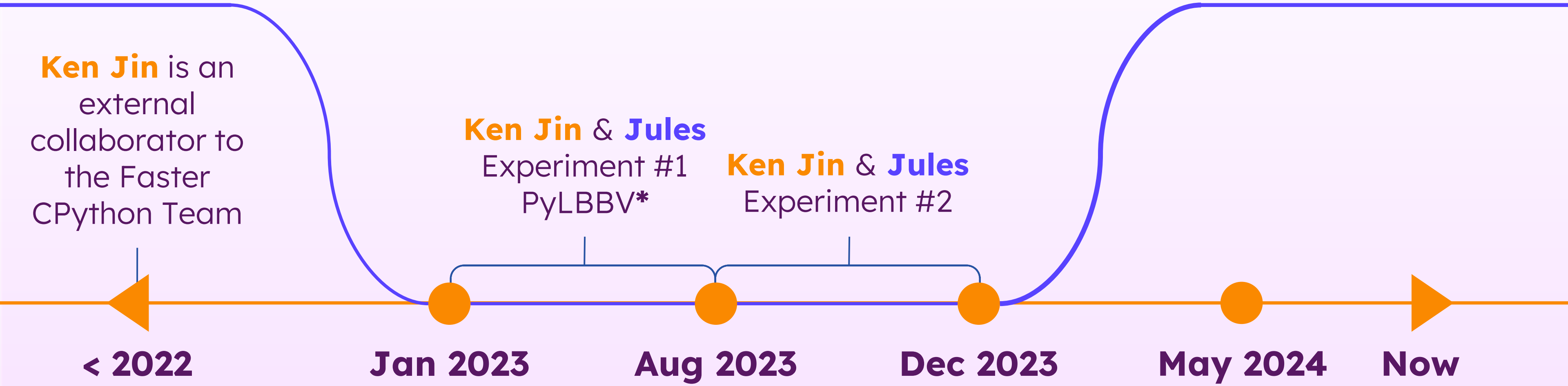


Ken Jin is an external collaborator to the Faster CPython Team

Ken Jin & **Jules**
Experiment #1
PyLBBV*

* 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

Ken Jin & **Jules**
Experiment #1
PyLBBV*

Ken Jin & **Jules**
Experiment #2

< 2022

Jan 2023

Aug 2023

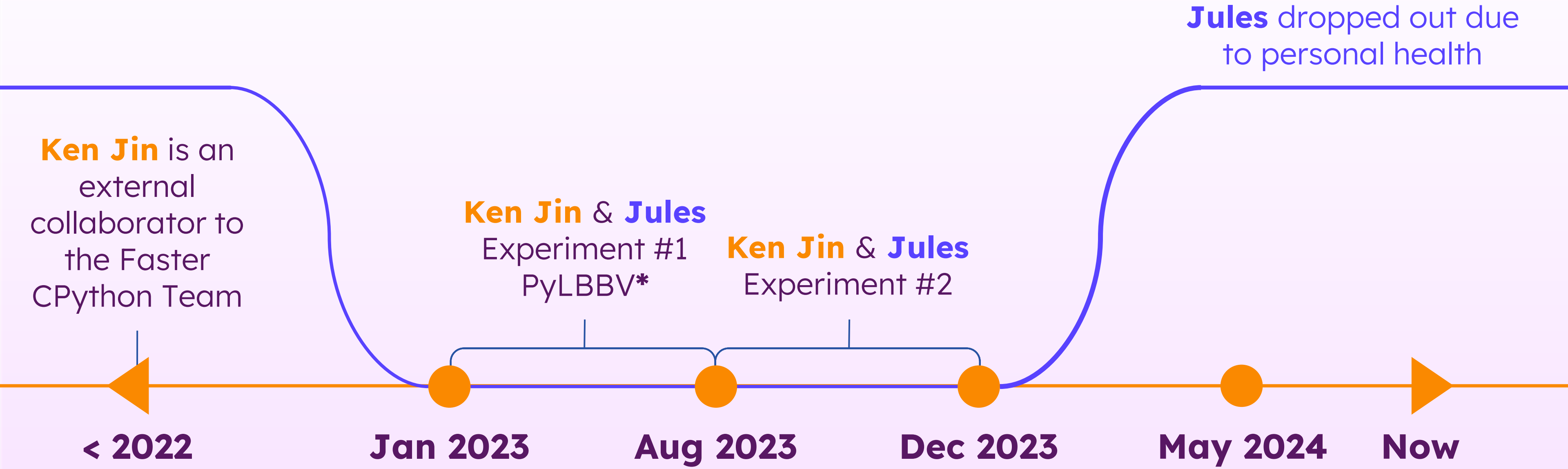
Dec 2023

May 2024

Now

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

Timeline



Jules dropped out due to personal health

Ken Jin is an external collaborator to the Faster CPython Team

Ken Jin & **Jules** Experiment #1 PyLBBV*

Ken Jin & **Jules** Experiment #2

< 2022

Jan 2023

Aug 2023

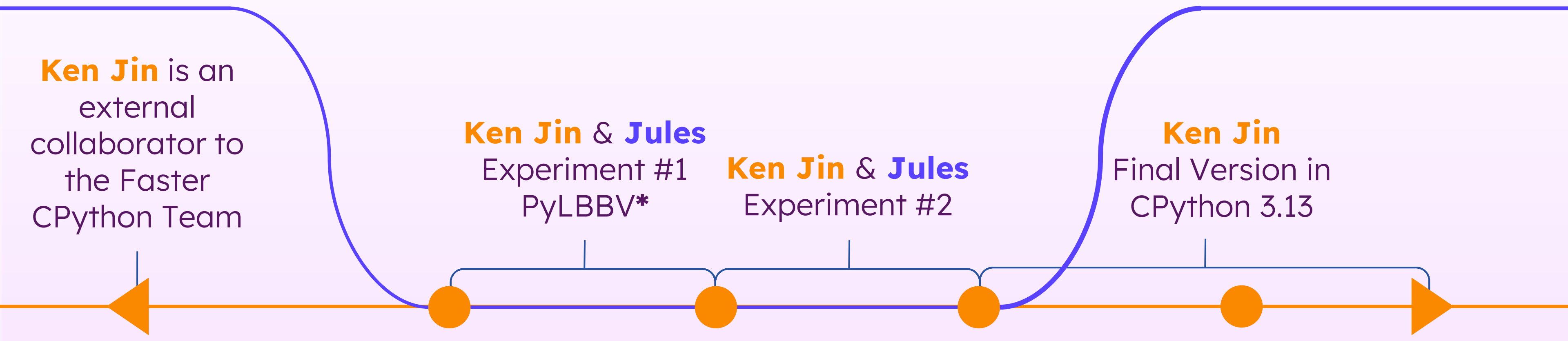
Dec 2023

May 2024

Now

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

Timeline



Jules dropped out due to personal health

Ken Jin is an external collaborator to the Faster CPython Team

Ken Jin & **Jules**
Experiment #1
PyLBBV*

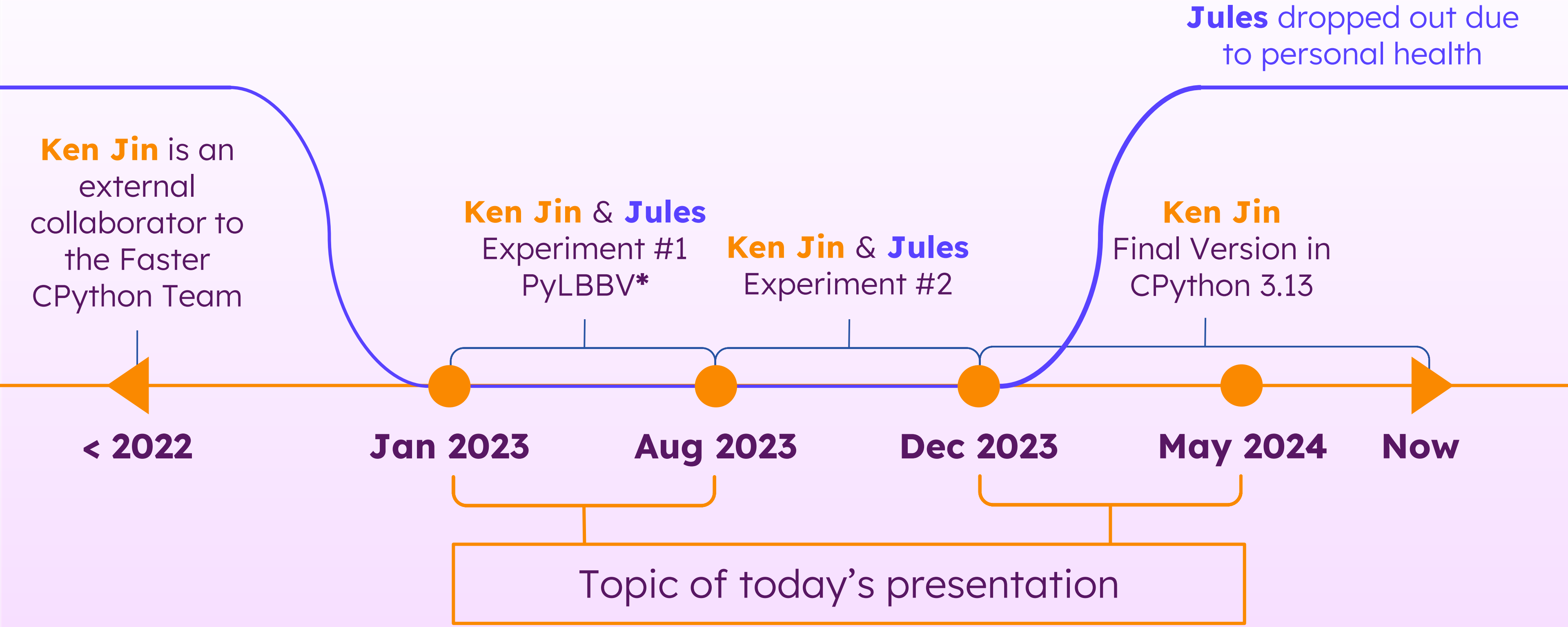
Ken Jin & **Jules**
Experiment #2

Ken Jin
Final Version in CPython 3.13

< 2022 **Jan 2023** **Aug 2023** **Dec 2023** **May 2024** **Now**

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

Timeline



Jules dropped out due to personal health

Ken Jin is an external collaborator to the Faster CPython Team

Ken Jin & **Jules**
Experiment #1
PyLBBV*

Ken Jin & **Jules**
Experiment #2

Ken Jin
Final Version in CPython 3.13

< 2022

Jan 2023

Aug 2023

Dec 2023

May 2024

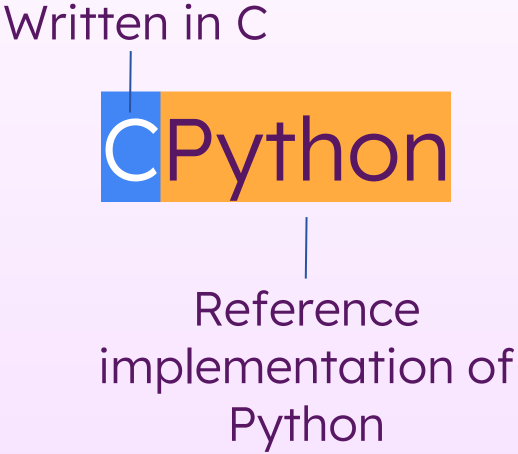
Now

Topic of today’s presentation

* 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

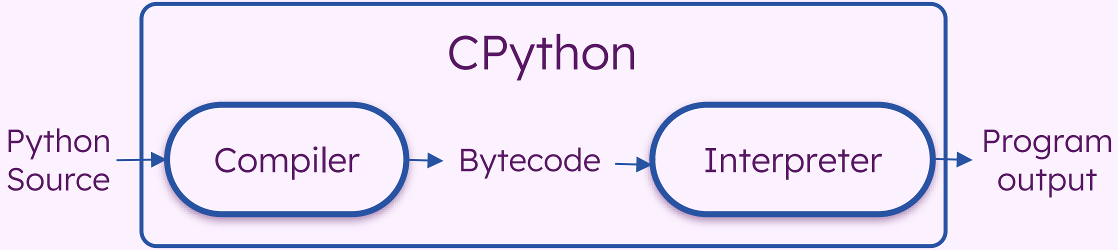
Bytecode Stack Machine

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

Python Source

Compiler

Bytecode

Interpreter

Program output

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

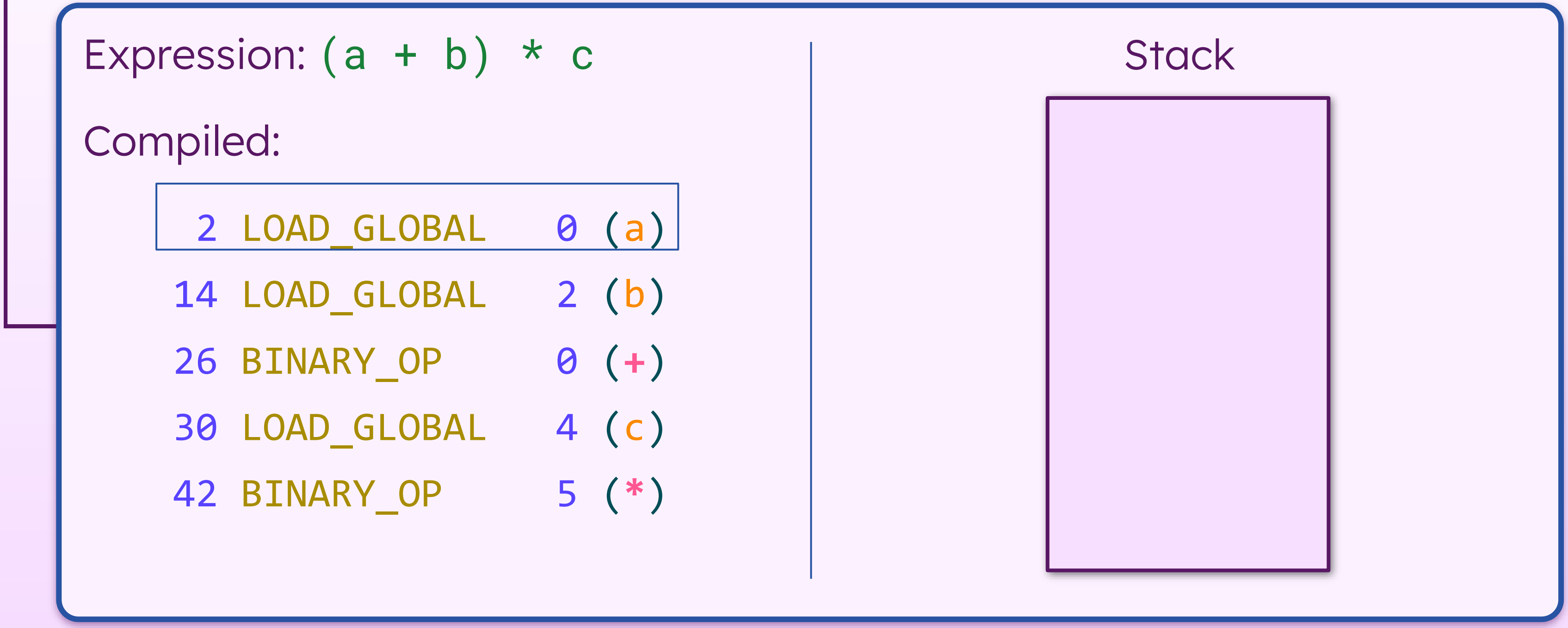
Expression: `(a + b) * c`

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 (*)
```

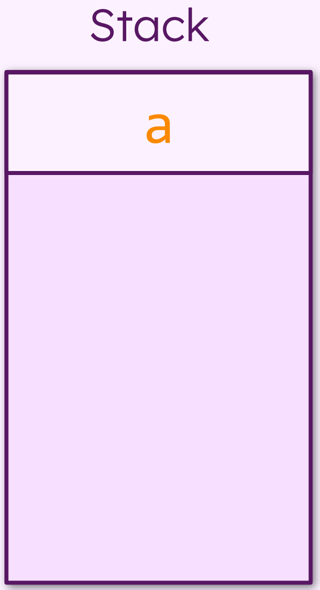
CPython: Bytecode Stack Machine

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

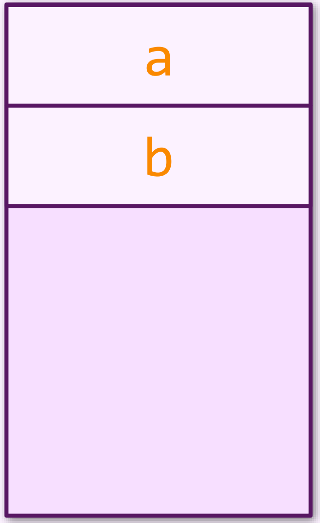
Bytecode Stack Machine

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



a

b

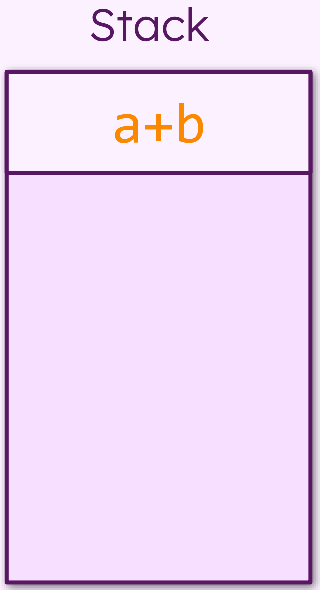
CPython: Bytecode Stack Machine

Bytecode Stack Machine

Expression: `(a + b) * c`

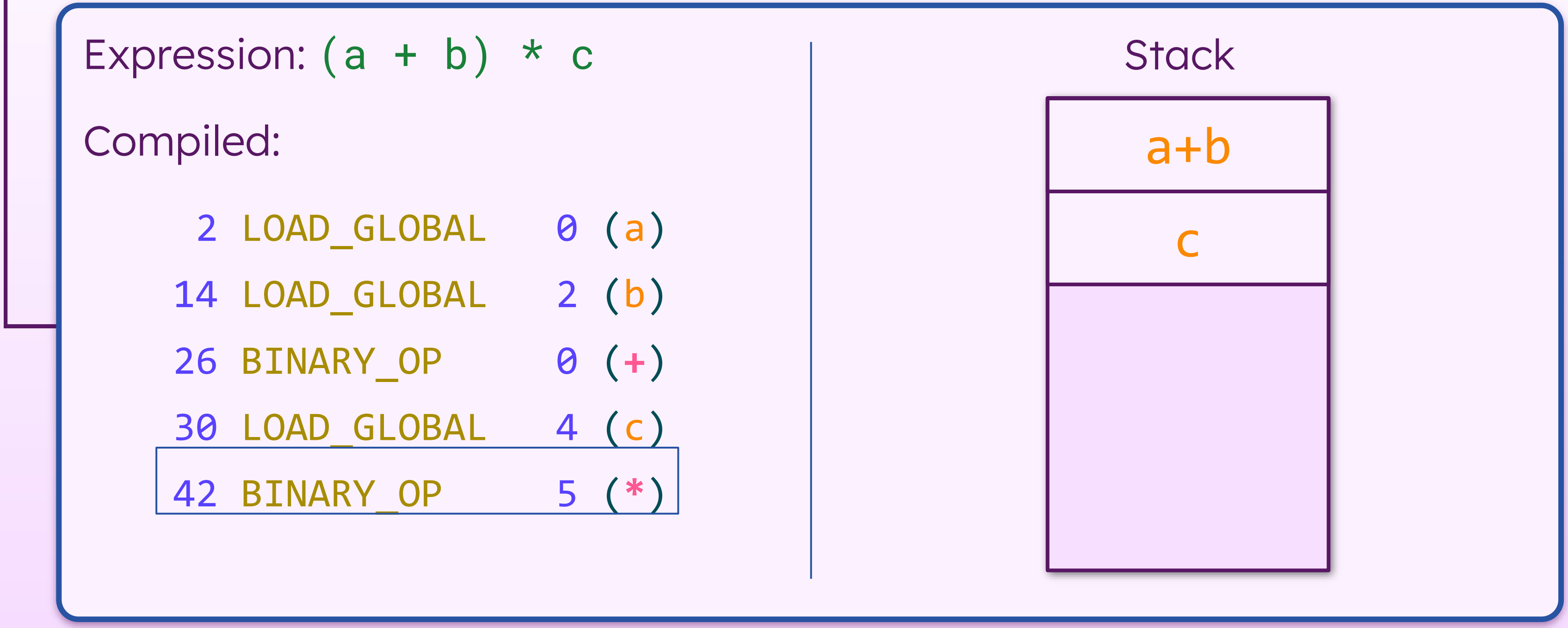
Compiled:

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



CPython: Bytecode Stack Machine

Bytecode Stack Machine



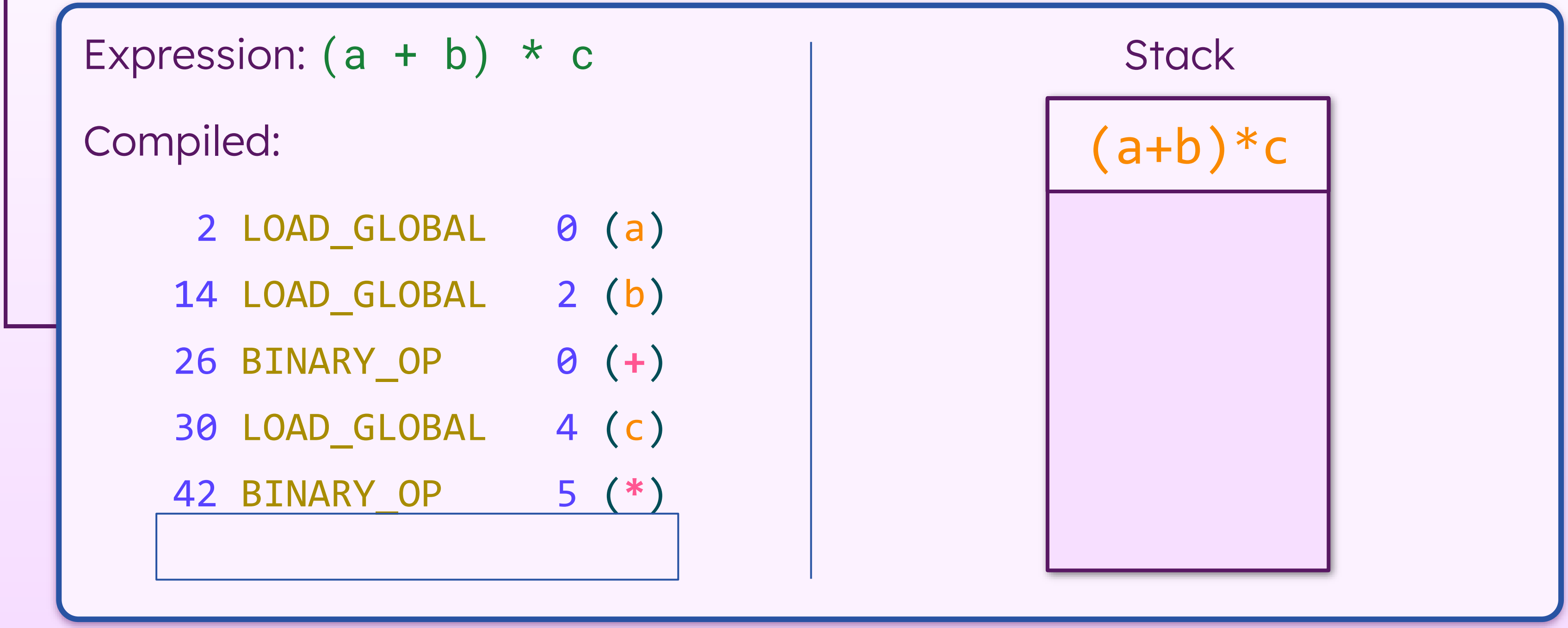
Expression: `(a + b) * c`

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 (*)
```

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)

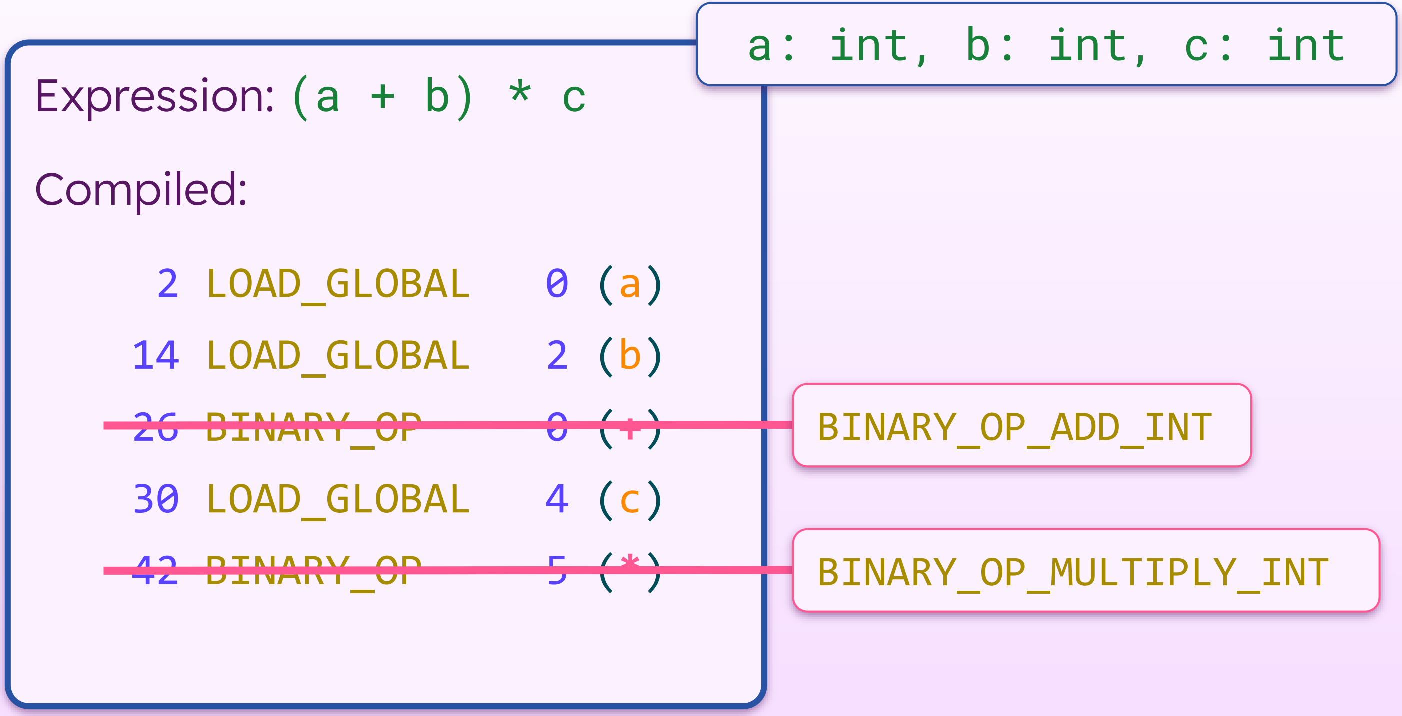
```
a: int, b: int, c: int
```

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
```

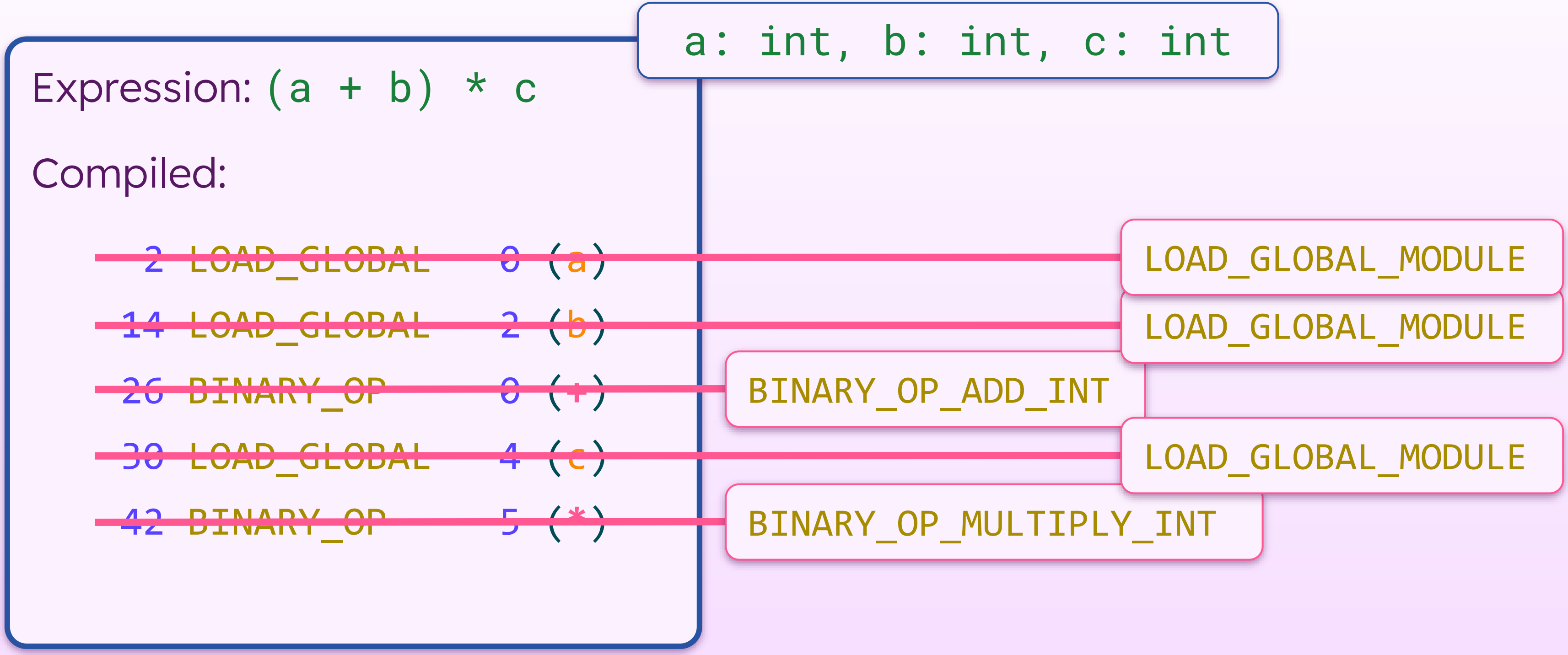
Expression: `(a + b) * c`

Compiled:

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

```
BINARY_OP_ADD_INT
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)~~

~~14 LOAD_GLOBAL 2 (b)~~

~~26 BINARY_OP 0 (+)~~

~~30 LOAD_GLOBAL 4 (c)~~

~~42 BINARY_OP 5 (*)~~

LOAD_GLOBAL_MODULE

LOAD_GLOBAL_MODULE

BINARY_OP_ADD_INT

LOAD_GLOBAL_MODULE

BINARY_OP_MULTIPLY_INT

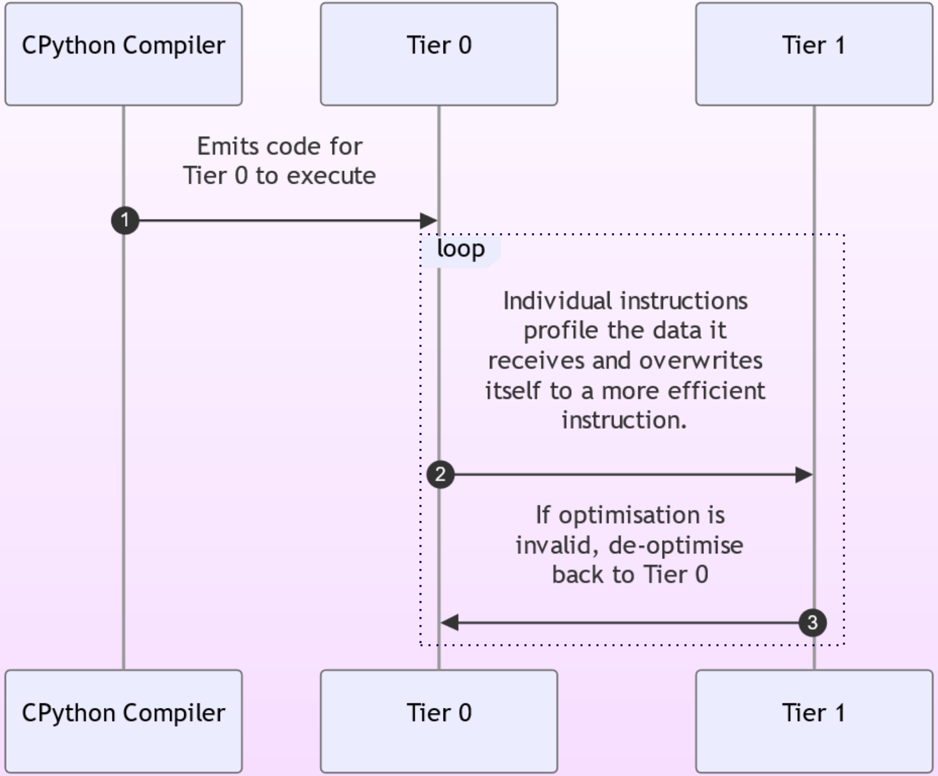
CPython: 3.11 Specialising Interpreter (Tier 1)

INSIDE CPYTHON 3.11'S NEW SPECIALIZING, ADAPTIVE INTERPRETER.

Brandt Bucher

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

CPython: 3.11 Overview



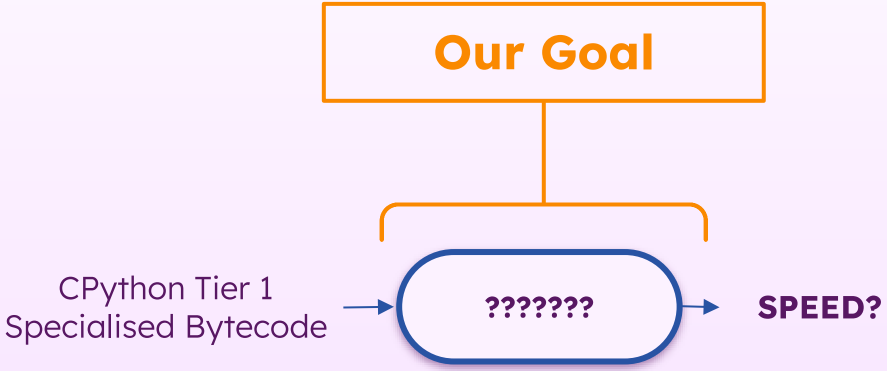
Emits code for Tier 0 to execute

Individual instructions profile the data it receives and overwrites itself to a more efficient instruction.

If optimisation is invalid, de-optimise back to Tier 0

Experiment #1: PyLBBV

Goal: Experiment #1



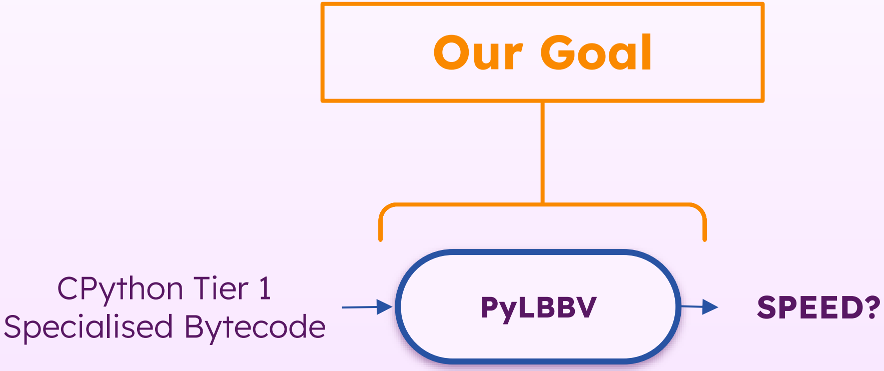
Our Goal

CPython Tier 1
Specialised Bytecode

???????

SPEED?

Goal: Experiment #1



Lazy Basic Block Versioning (LBBV)

Maxime Chevalier-Boisvert and Marc Feeley: doi: 10.48550/arXiv.1411.0352

Novel idea to remove overhead of dynamic typing for JIT compilers:

- Insert explicit type checks into code
- Lazily generate optimised versions of a basic block (**BB**) according to the runtime types encountered at the type checks

PyLBBV: LBBV in Python

Introduce new class of instructions: **Tier 2 Instructions**

{ Later, when discussing CPython 3.13 I'll refer to a similar construct as **Uops** }

- Split Tier 1 instructions up into smaller instructions
  - Allow identification of repeated work. E.g., separating type checks from the execution of the instruction.
- Type guards added
- Special branching instructions added to support lazy BB generation
- { Extra } Float unboxing instructions added

After code gets **hot** (executed 63 times), **Tier 2** execution begins.

PyLBBV: Example

Python Source

```
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

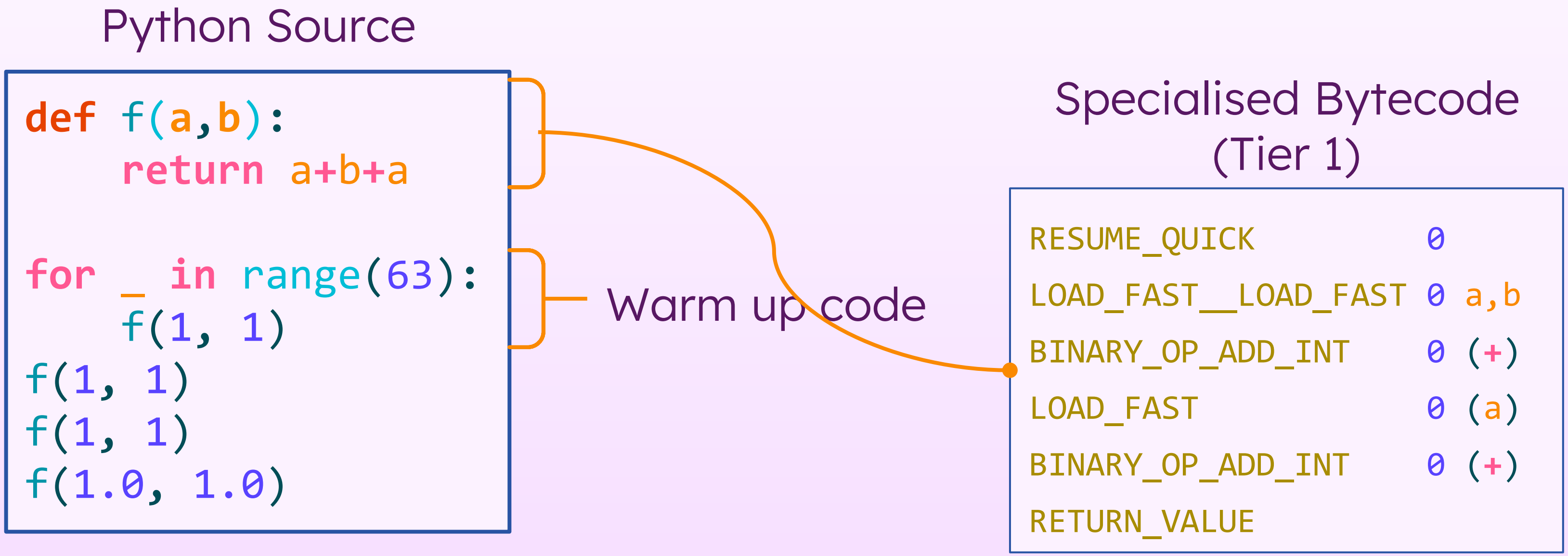
PyLBBV: Example

Python Source

```python
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

Warm up code

PyLBBV: Example



Python Source

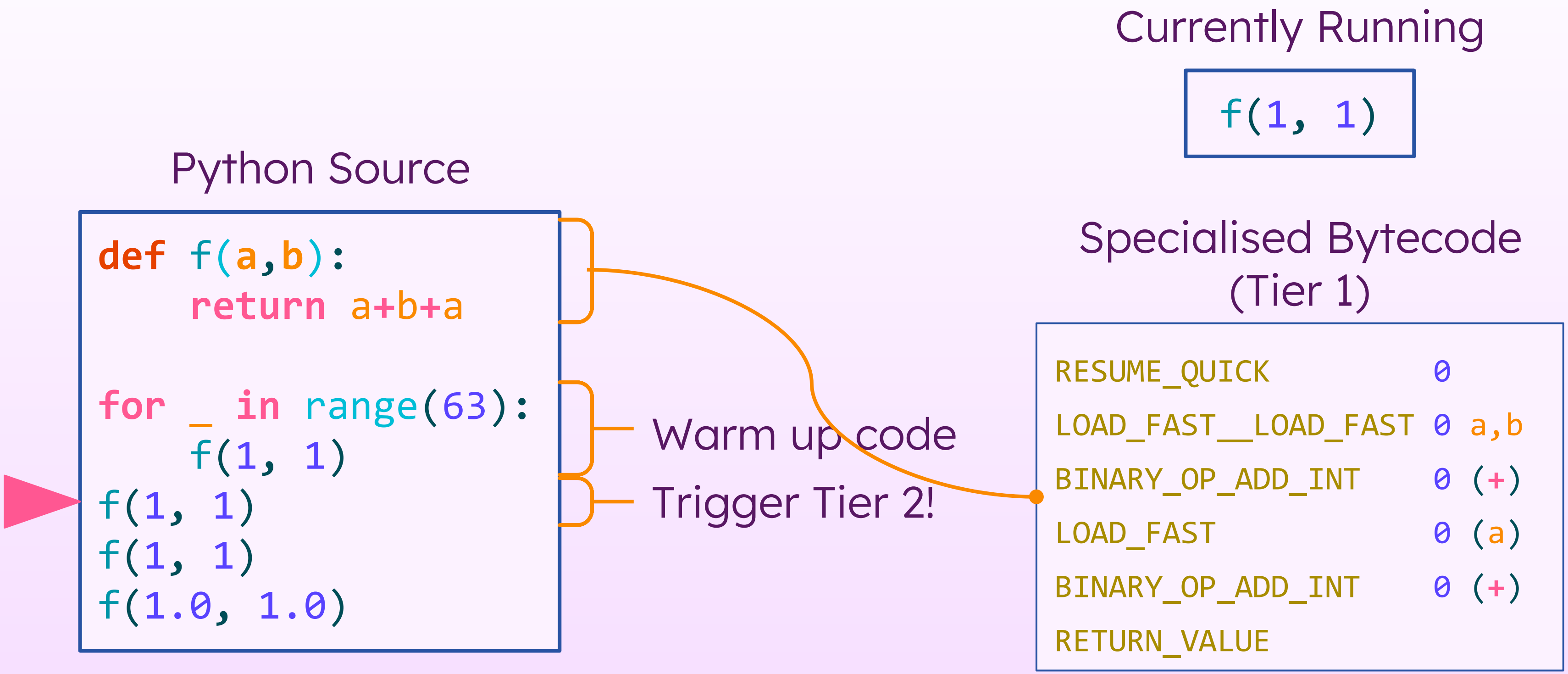
```
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

Warm up code

Specialised Bytecode
(Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

PyLBBV: Example



Currently Running

```
f(1, 1)
```

Python Source

```
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

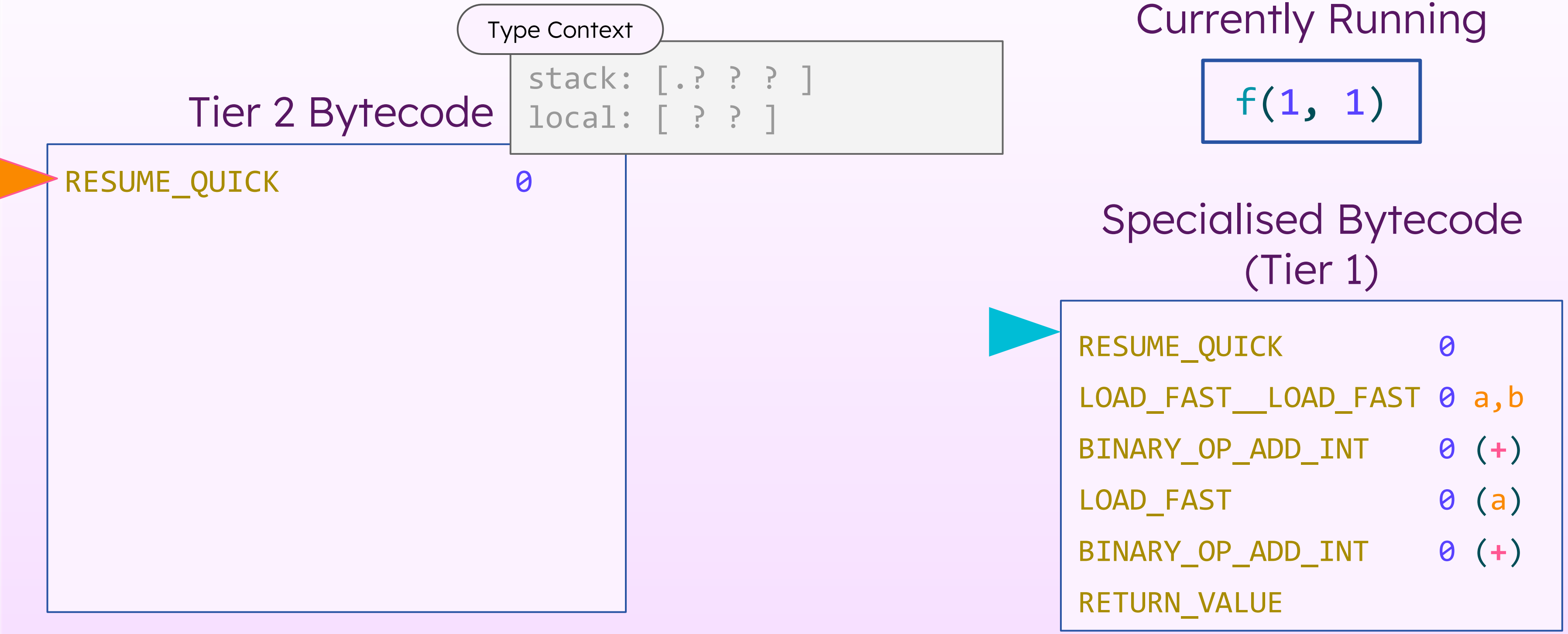
Warm up code

Trigger Tier 2!

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

PyLBBV: Example



Type Context

```
stack: [.? ? ? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK            0
```

Currently Running

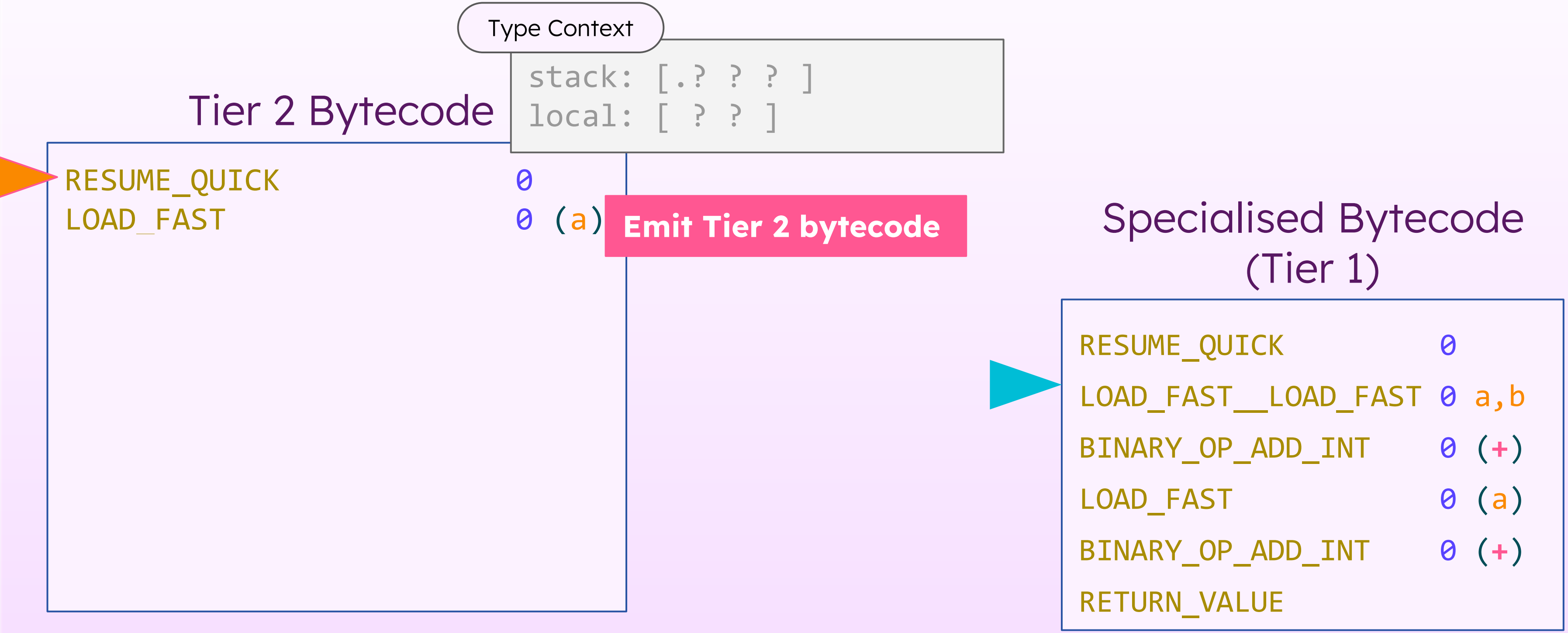
```
f(1, 1)
```

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [.? ? ? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK           0
LOAD_FAST              0 (a)
```

Emit Tier 2 bytecode

Specialised Bytecode
(Tier 1)

```
RESUME_QUICK           0
LOAD_FAST__LOAD_FAST   0 a,b
BINARY_OP_ADD_INT      0 (+)
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT      0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ ?.? ? ]
local: [ ? ? ]
```

Type Context type propagates across emitted instruction

Tier 2 Bytecode

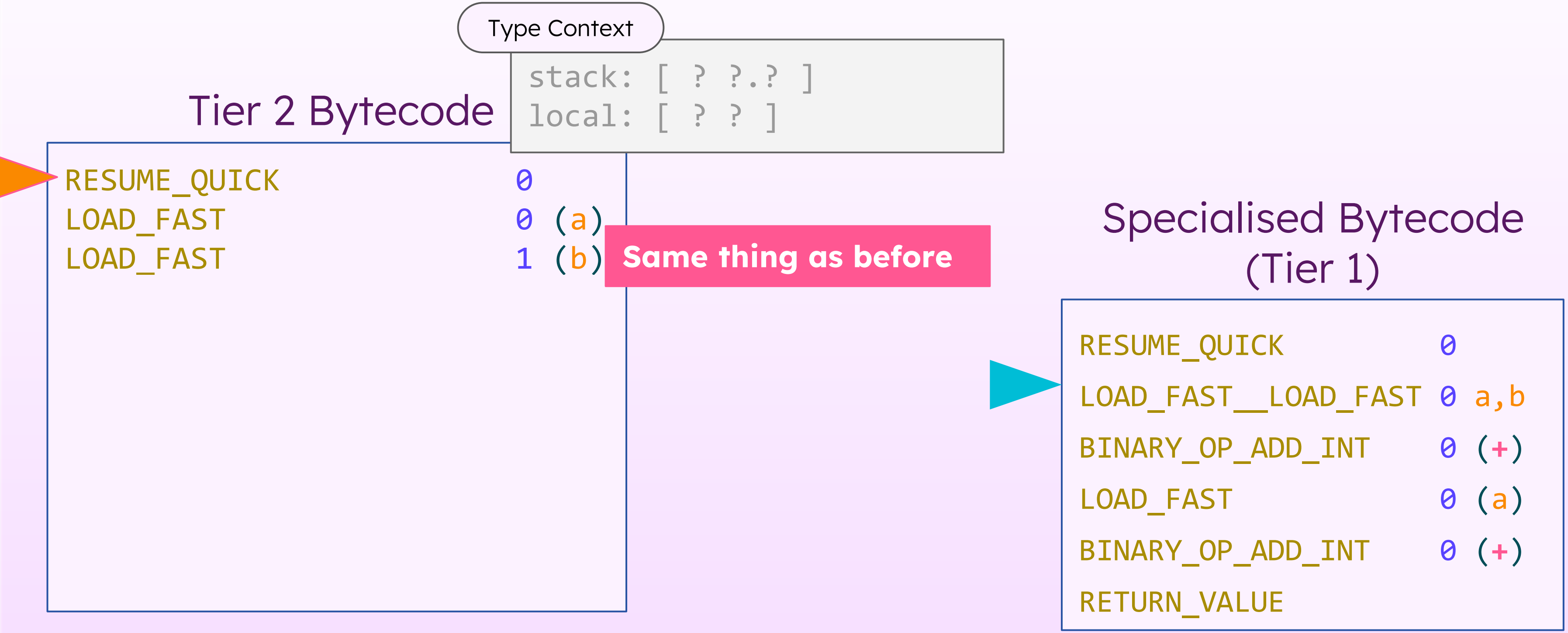
```
RESUME_QUICK                0
LOAD_FAST                   0 (a)
```

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK               0
LOAD_FAST                  0 (a)
LOAD_FAST                  1 (b)
```

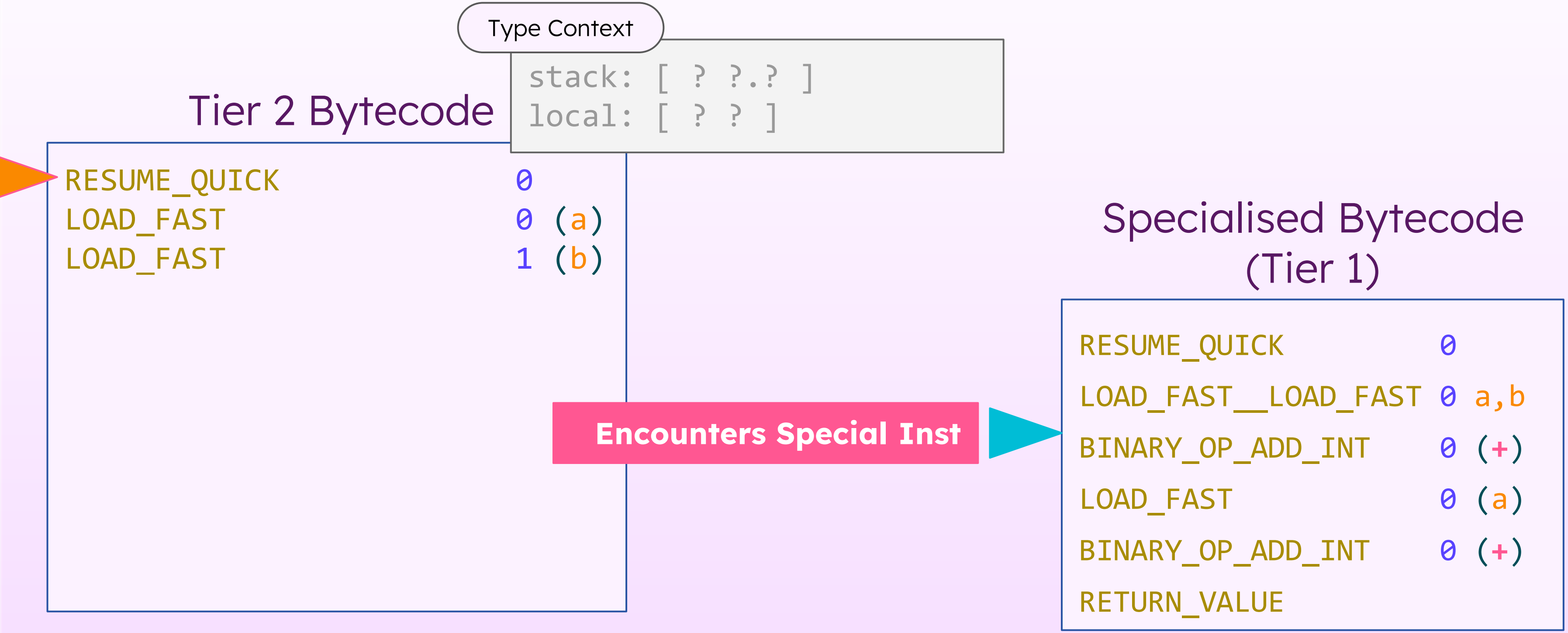
Same thing as before

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
```

Encounters Special Inst

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK        0
LOAD_FAST           0 (a)
LOAD_FAST           1 (b)
CHECK_INT
```

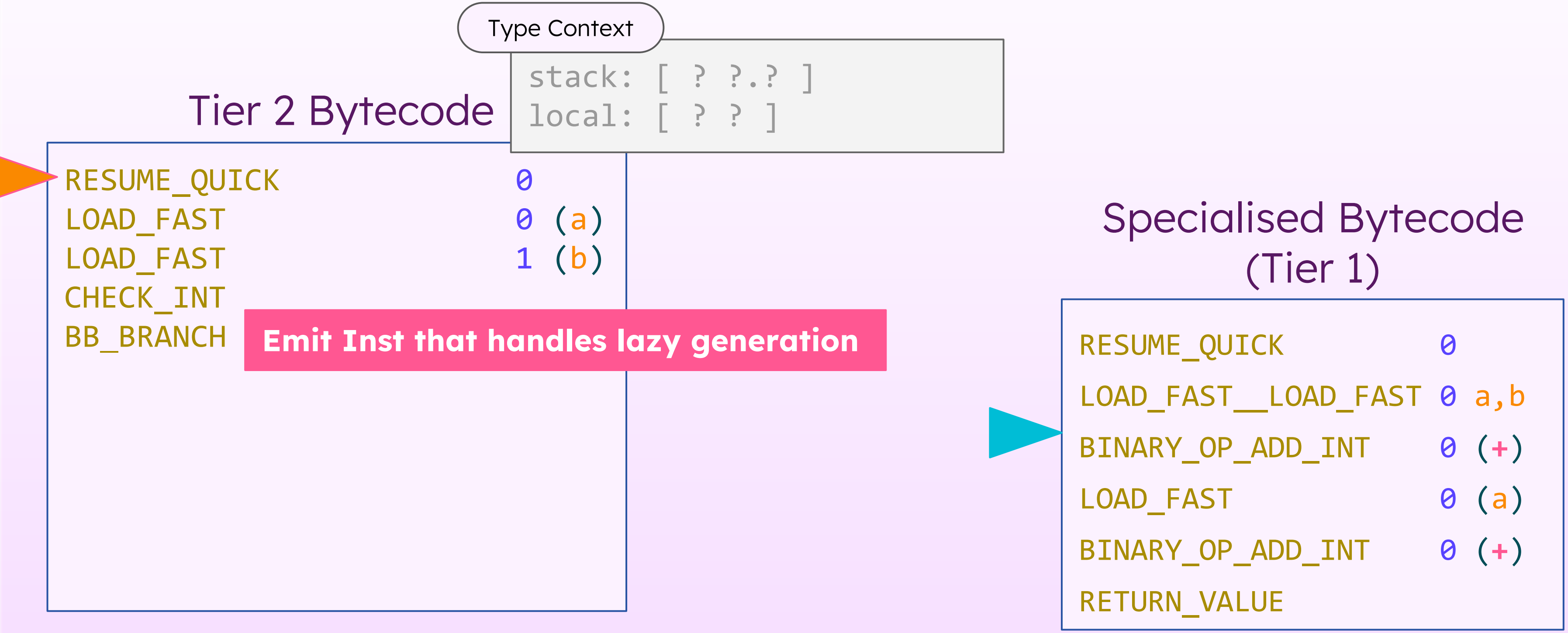
Emit Guard for a **since Type Context has no idea what type it is**

Specialised Bytecode (Tier 1)

```
RESUME_QUICK           0
LOAD_FAST__LOAD_FAST   0 a,b
BINARY_OP_ADD_INT      0 (+)
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT      0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK          0
LOAD_FAST             0 (a)
LOAD_FAST             1 (b)
CHECK_INT
BB_BRANCH
```

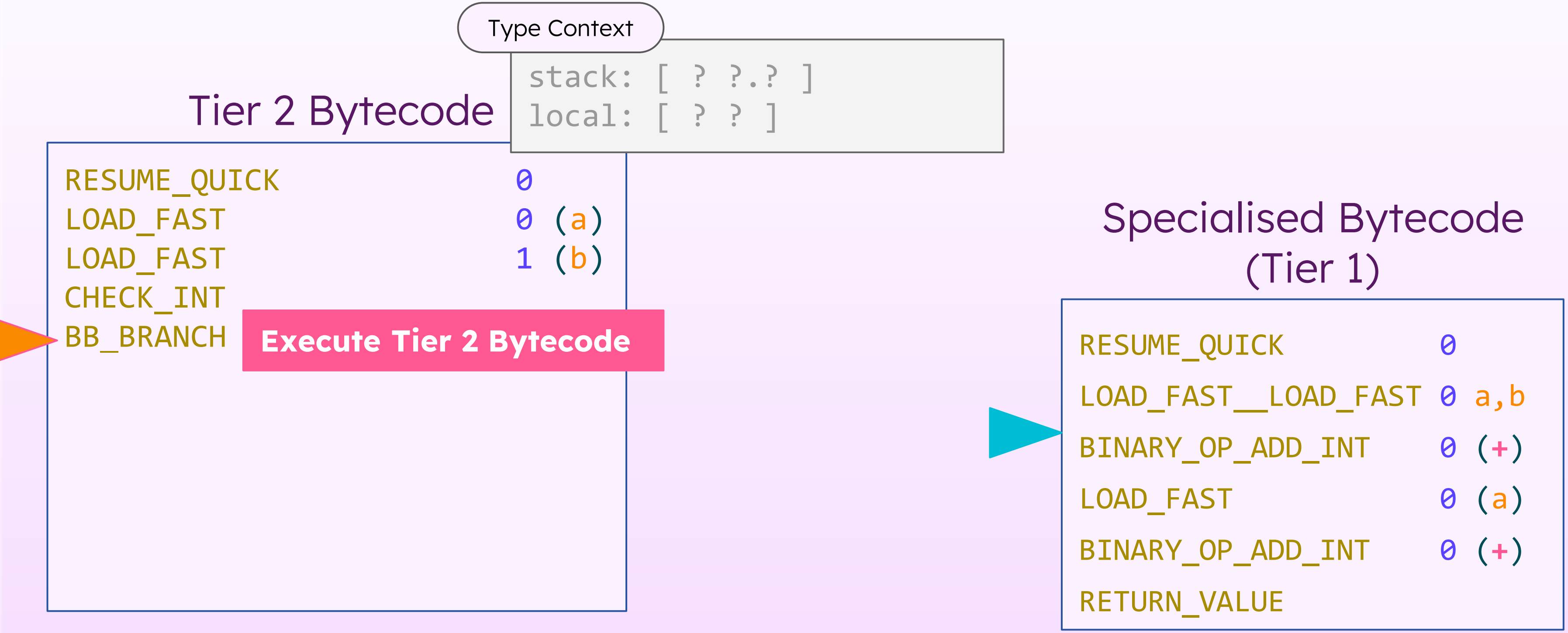
Emit Inst that handles lazy generation

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK         0
LOAD_FAST            0 (a)
LOAD_FAST            1 (b)
CHECK_INT
BB_BRANCH
```

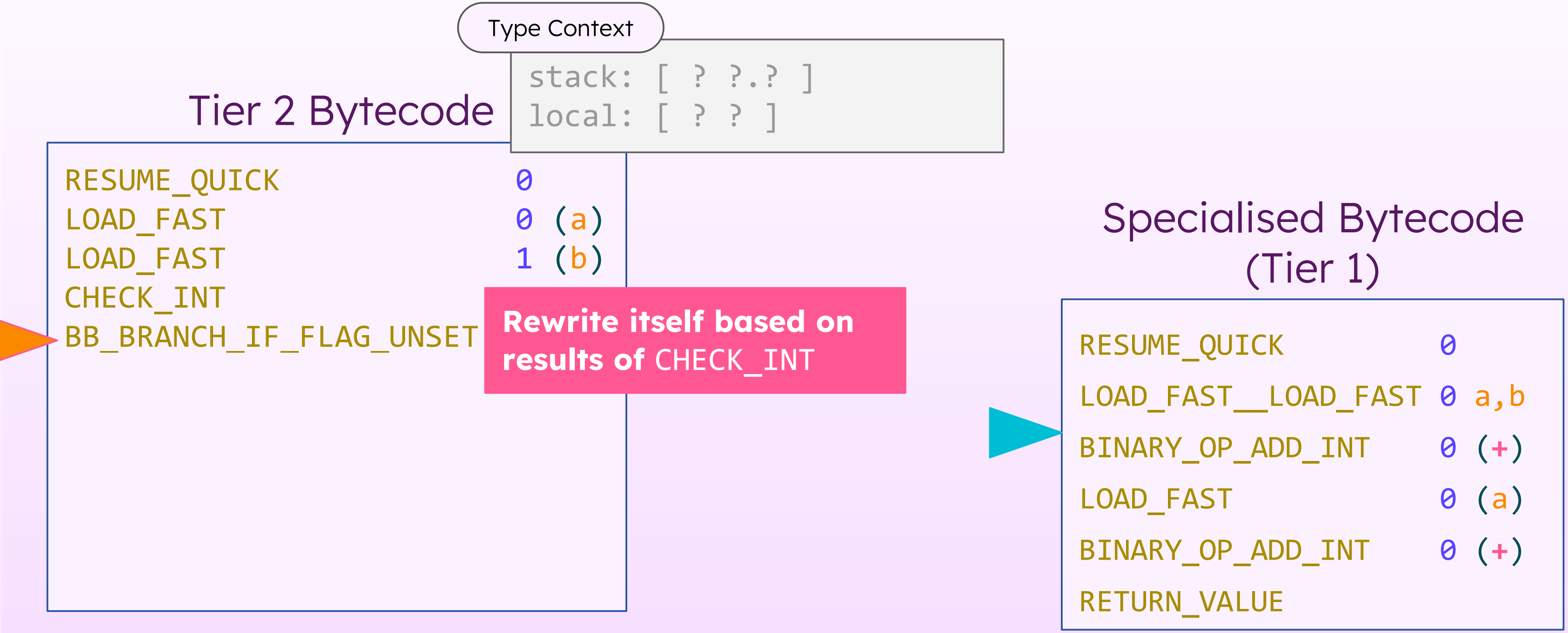
Execute Tier 2 Bytecode

Specialised Bytecode
(Tier 1)

```
RESUME_QUICK         0
LOAD_FAST__LOAD_FAST 0 a,b
BINARY_OP_ADD_INT    0 (+)
LOAD_FAST            0 (a)
BINARY_OP_ADD_INT    0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET
```

Rewrite itself based on results of CHECK_INT

Specialised Bytecode (Tier 1)

```
RESUME_QUICK           0
LOAD_FAST__LOAD_FAST   0 a,b
BINARY_OP_ADD_INT      0 (+)
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT      0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ int ?.? ]
local: [ int ? ]
```

Update Type Context based on BB we intend to generate

Tier 2 Bytecode

```
RESUME_QUICK            0
LOAD_FAST               0 (a)
LOAD_FAST               1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET 0
```

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ int ?.? ]
local: [ int ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF
CHECK_INT
```

Emit Guard for b since Type Context has no idea what type it is

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ int int.? ]
local: [ int int ]
```

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
```

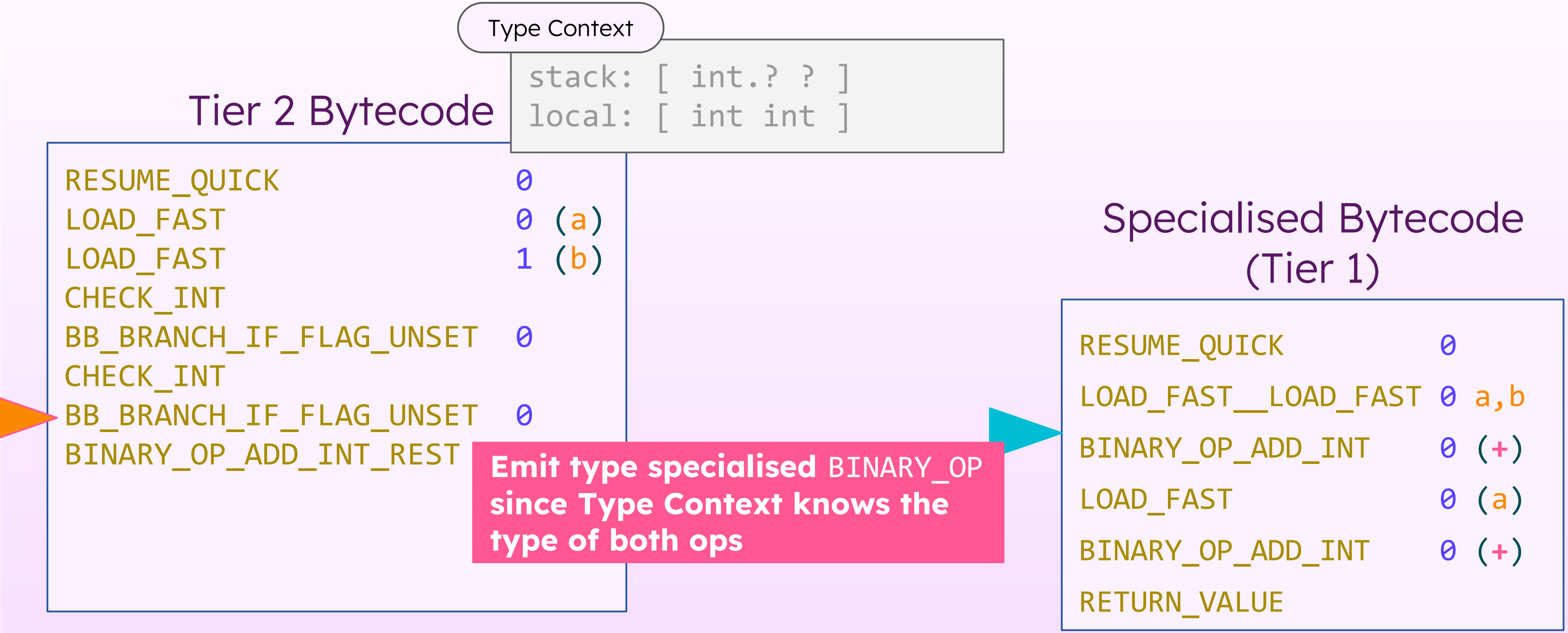
Same thing happens as before

Specialised Bytecode (Tier 1)

```
RESUME_QUICK           0
LOAD_FAST__LOAD_FAST 0 a,b
BINARY_OP_ADD_INT      0 (+)
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT      0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ int.? ? ]
local: [ int int ]
```

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
BINARY_OP_ADD_INT_REST
```

Emit type specialised BINARY_OP **since Type Context knows the type of both ops**

Specialised Bytecode (Tier 1)

```
RESUME_QUICK           0
LOAD_FAST__LOAD_FAST   0 a,b
BINARY_OP_ADD_INT      0 (+)
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT      0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ int int.? ]
local: [ int int ]
```

Tier 2 Bytecode

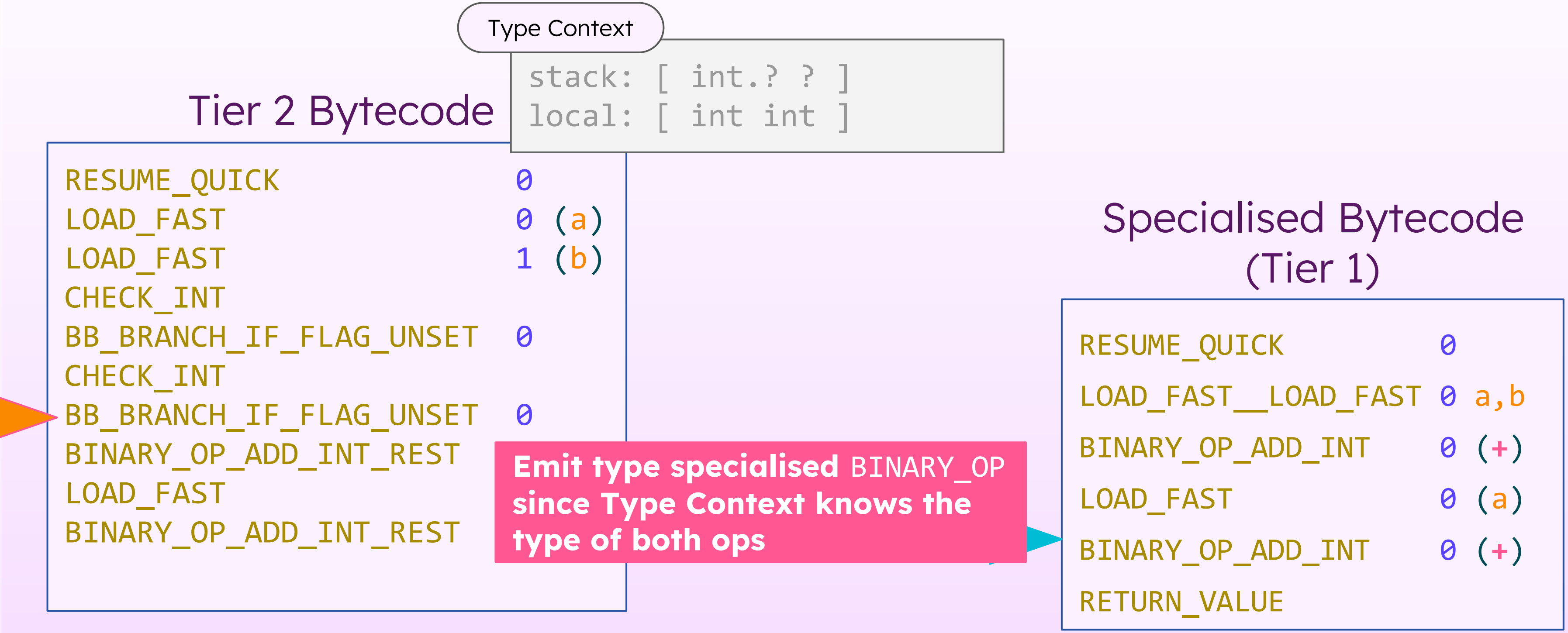
```
RESUME_QUICK            0
LOAD_FAST               0 (a)
LOAD_FAST               1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET 0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET 0
BINARY_OP_ADD_INT_REST
LOAD_FAST               0 (a)
```

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ int.? ? ]
local: [ int int ]
```

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
BINARY_OP_ADD_INT_REST
LOAD_FAST
BINARY_OP_ADD_INT_REST
```

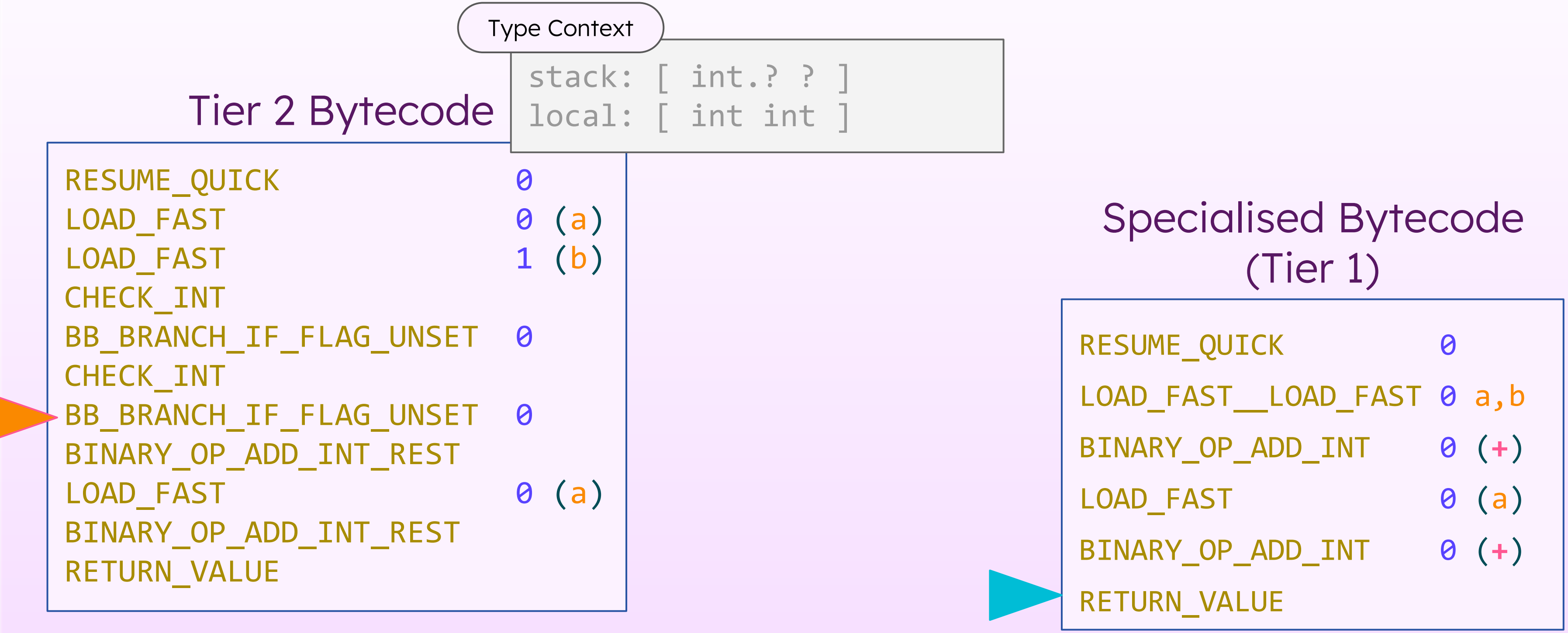
Emit type specialised BINARY_OP **since Type Context knows the type of both ops**

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ int.? ? ]
local: [ int int ]
```

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST                0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Specialised Bytecode (Tier 1)

```
RESUME_QUICK         0
LOAD_FAST__LOAD_FAST 0 a,b
BINARY_OP_ADD_INT    0 (+)
LOAD_FAST            0 (a)
BINARY_OP_ADD_INT    0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [.? ? ? ]
local: [ int int ]
```

Tier 2 Bytecode

```
RESUME_QUICK               0
LOAD_FAST                  0 (a)
LOAD_FAST                  1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET    0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET    0
BINARY_OP_ADD_INT_REST
LOAD_FAST                  0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Execute Tier 2 Bytecode

Specialised Bytecode (Tier 1)

```
RESUME_QUICK          0
LOAD_FAST__LOAD_FAST  0 a,b
BINARY_OP_ADD_INT     0 (+)
LOAD_FAST             0 (a)
BINARY_OP_ADD_INT     0 (+)
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Currently Running

```
f(1, 1)
```

Python Source

```
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

Running Tier 2 again does not trigger anymore codegen

PyLBBV: Example

Currently Running

```
f(1.0, 1.0)
```

Python Source

```
def f(a,b):
    return a+b+a

for _ in range(63):
    f(1, 1)
f(1, 1)
f(1, 1)
f(1.0, 1.0)
```

Run Tier 2 bytecode with different types

PyLBBV: Example

Currently Running

```
f(1.0, 1.0)
```

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST                0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: Example

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
BINARY_OP_ADD_INT_REST
LOAD_FAST                 0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Guard fails!

* Small modifications made for explainability

PyLBBV: Example

Tier 2 Bytecode

```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST                0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Prompts to generate BB_BRANCH_IF_FLAG_UNSET **new BB**

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK           0
LOAD_FAST              0 (a)
LOAD_FAST              1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST              0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Recover Type Context for the BB generation

* Small modifications made for explainability

PyLBBV: Example

Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

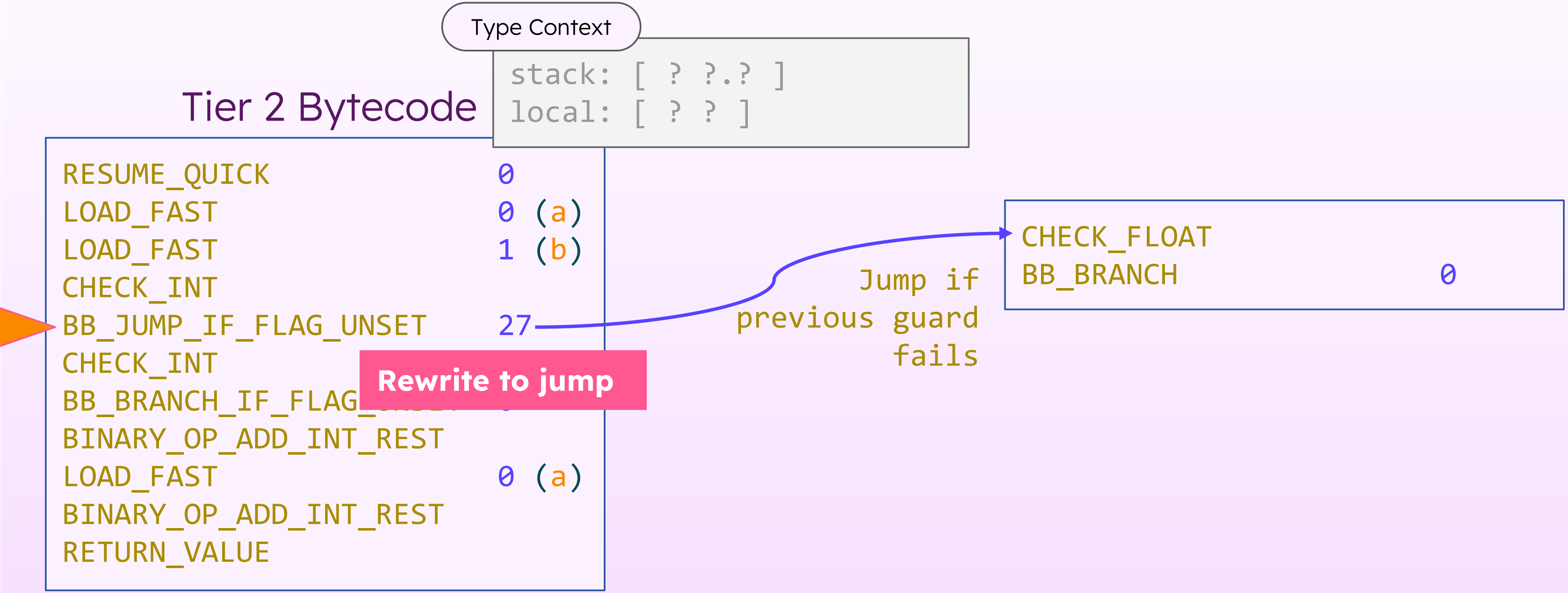
```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST                0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Generate new BB with float guard

```
CHECK_FLOAT
BB_BRANCH                0
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [ ? ?.? ]
local: [ ? ? ]
```

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
CHECK_INT
BB_JUMP_IF_FLAG_UNSET     27
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
BINARY_OP_ADD_INT_REST
LOAD_FAST                 0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

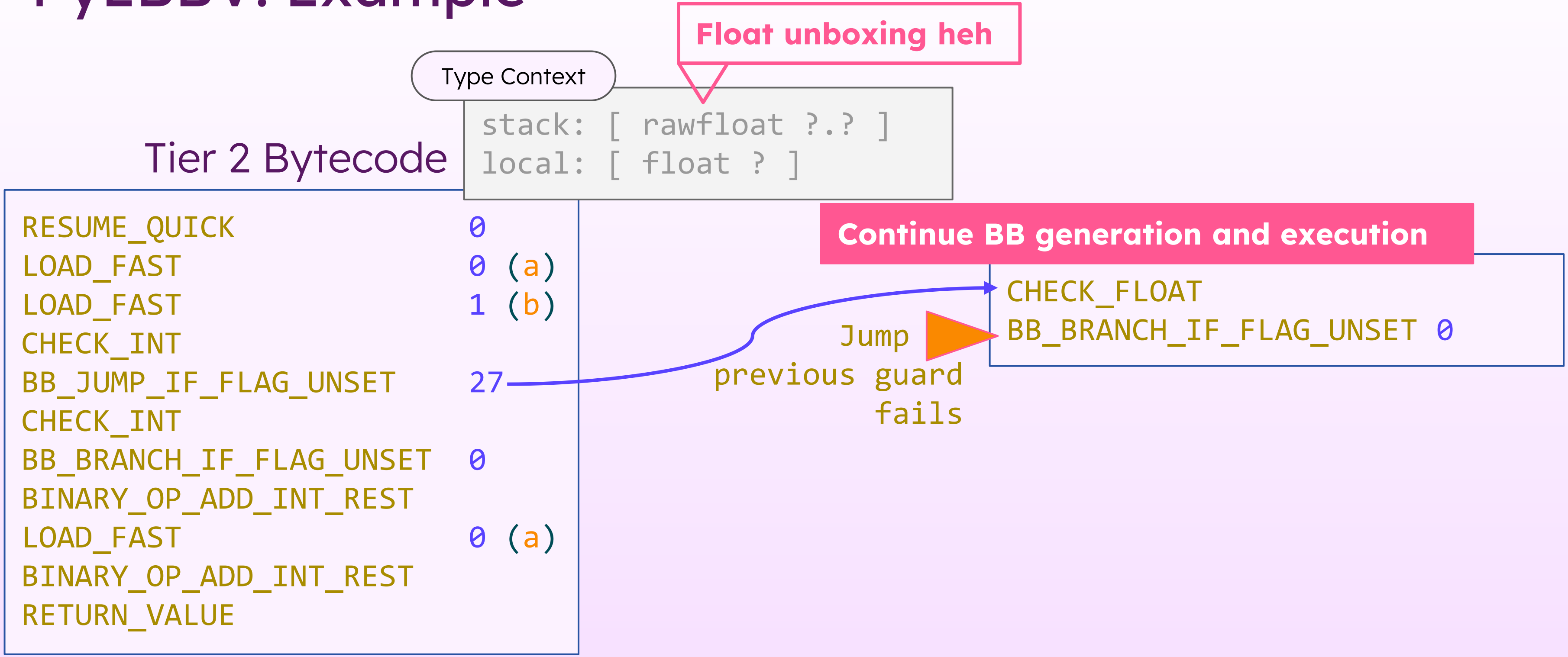
Rewrite to jump

```
   Jump if
previous guard
         fails
```

```
CHECK_FLOAT
BB_BRANCH                        0
```

* Small modifications made for explainability

PyLBBV: Example



Float unboxing heh

Type Context

```
stack: [ rawfloat ?.? ]
local: [ float ? ]
```

Tier 2 Bytecode

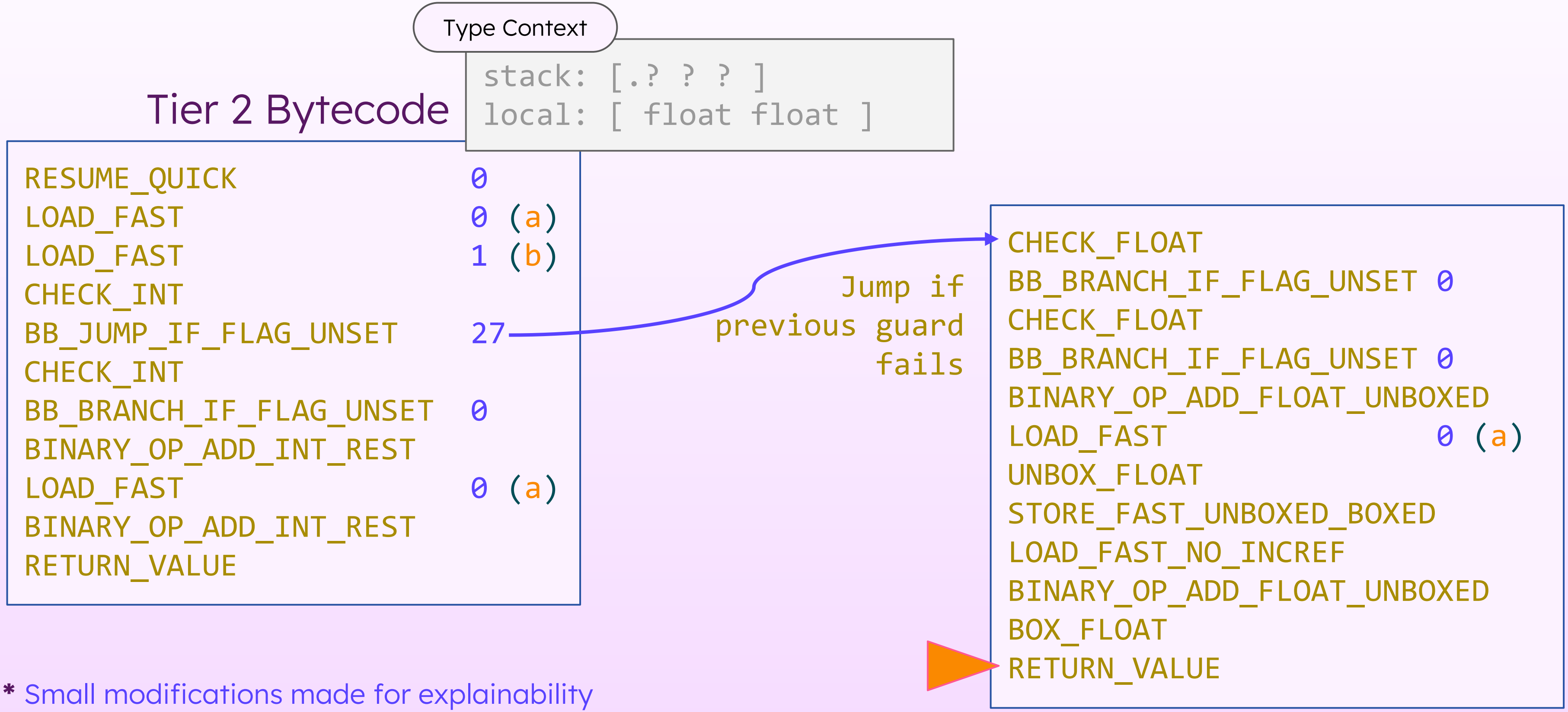
```
RESUME_QUICK             0
LOAD_FAST                0 (a)
LOAD_FAST                1 (b)
CHECK_INT
BB_JUMP_IF_FLAG_UNSET    27
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET  0
BINARY_OP_ADD_INT_REST
LOAD_FAST                0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

Continue BB generation and execution

```
CHECK_FLOAT
BB_BRANCH_IF_FLAG_UNSET 0
```

* Small modifications made for explainability

PyLBBV: Example



Type Context

```
stack: [.? ? ? ]
local: [ float float ]
```

Tier 2 Bytecode

```
RESUME_QUICK              0
LOAD_FAST                 0 (a)
LOAD_FAST                 1 (b)
CHECK_INT
BB_JUMP_IF_FLAG_UNSET     27
CHECK_INT
BB_BRANCH_IF_FLAG_UNSET   0
BINARY_OP_ADD_INT_REST
LOAD_FAST                 0 (a)
BINARY_OP_ADD_INT_REST
RETURN_VALUE
```

```
       Jump if
previous guard
         fails
```

```
CHECK_FLOAT
BB_BRANCH_IF_FLAG_UNSET 0
CHECK_FLOAT
BB_BRANCH_IF_FLAG_UNSET 0
BINARY_OP_ADD_FLOAT_UNBOXED
LOAD_FAST               0 (a)
UNBOX_FLOAT
STORE_FAST_UNBOXED_BOXED
LOAD_FAST_NO_INCREF
BINARY_OP_ADD_FLOAT_UNBOXED
BOX_FLOAT
RETURN_VALUE
```

* Small modifications made for explainability

PyLBBV: LBBV in Python

What we get:

- Guard Elimination
- Float Unboxing { suboptimal }
- Free Deadcode Elimination
- Free Loop Unpeeling
  - Exploit type stability in loops even if initial iterations are unstable

PyLBBV: Type Propagator

When PyLBBV was written, there were **207 instructions**.

- **Problem:** Manually writing the Type Propagator is extremely tedious

Right before PyLBBV, a **DSL** was introduced to specify Python's bytecode interpreter.

- **Solution:** Modify **DSL** to specify Type Propagator semantics, generate Type Propagator automatically.
    - Type propagator is < 1000 lines of handwritten C

{ Later, in CPython 3.13 a similar approach is used to generate the abstract interpreter }

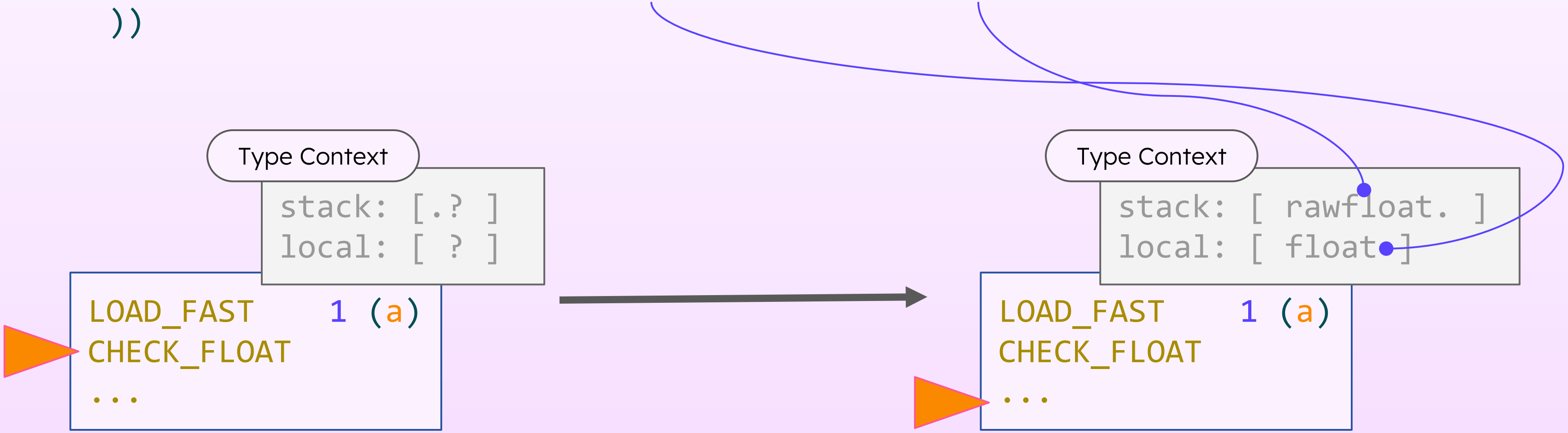
PyLBBV: Type Propagator

Added type semantics

```
inst(CHECK_FLOAT, (
         maybe_float, unused[oparg]
         --
         unboxed_float : {<<= PyFloat_Type, PyRawFloat_Type}, unused[oparg]
     ))
{
    assert(cframe.use_tracing == 0);
    char is_successor = PyFloat_CheckExact(maybe_float);
    frame->bb_test = BB_TEST(is_successor, 0);

    if (is_successor) {
        unboxed_float = *((PyObject **)(&(((PyFloatObject *)maybe_float)-
>ob_fval)));
        DECREF_INPUTS();
    }
    else {
        unboxed_float = maybe_float;
    }
}
```

PyLBBV: Type Propagator

```
inst(CHECK_FLOAT, (
           maybe_float, unused[oparg]
           --
           unboxed_float : {<<= PyFloat_Type, PyRawFloat_Type}, unused[oparg]
     ))
```



Type Context

```
stack: [.? ]
local: [ ? ]
```

```
LOAD_FAST      1 (a)
CHECK_FLOAT
...
```

Type Context

```
stack: [ rawfloat. ]
local: [ float ]
```

```
LOAD_FAST      1 (a)
CHECK_FLOAT
...
```

PyLBBV: Challenges

Really difficult to support all of Python.

A lot of eventual work done on **Uops** in CPython 3.13 was not done while we were writing PyLBBV

- A lot of Tier 2 instructions were “ad-hoc”
- A lot of annoying edge case
    - e.g., `FOR_ITER` had runtime dependent stack effect

PyLBBV: Results

PyLBBV:

- `39.8%` speedup in arithmetic. **No speedup at all elsewhere.**
- PyLBBV does not support a huge portion of Python, so we were unable properly profile it.

We later slapped on Brandt Bucher's Copy-and-Patch JIT Compiler:

PyLBBV + JIT:

- `12.0%` speedup in arithmetic. (a slowdown!)
- PyLBBV's basic blocks are too short for JIT to be helpful.

CPython 3.13 and onwards

CPython 3.11 → 3.13 and onwards

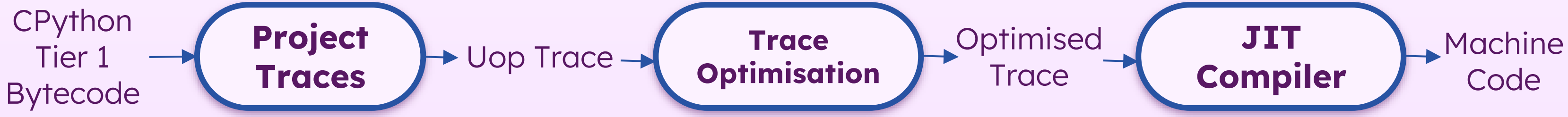
CPython 3.11:

- Specialising Interpreter optimizes across one to two bytecode

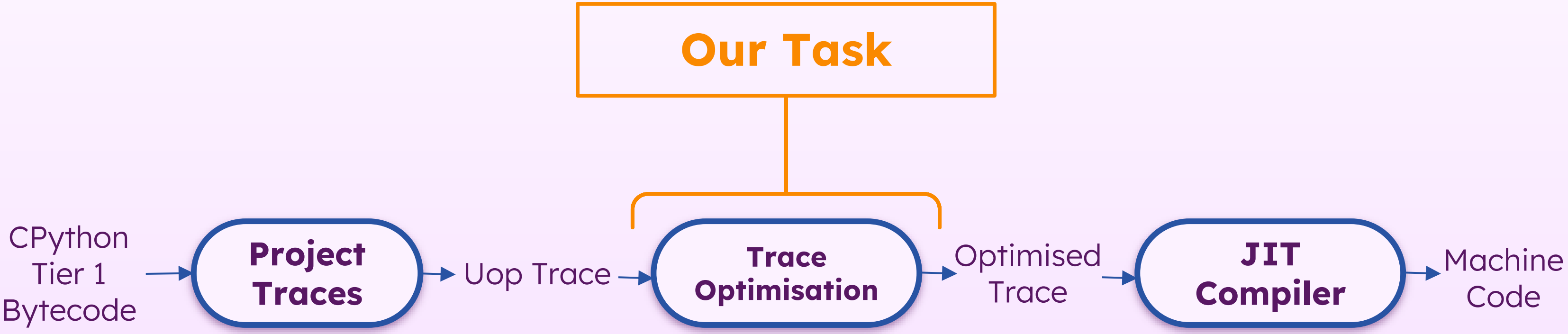
CPython 3.13 and onwards:

- 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

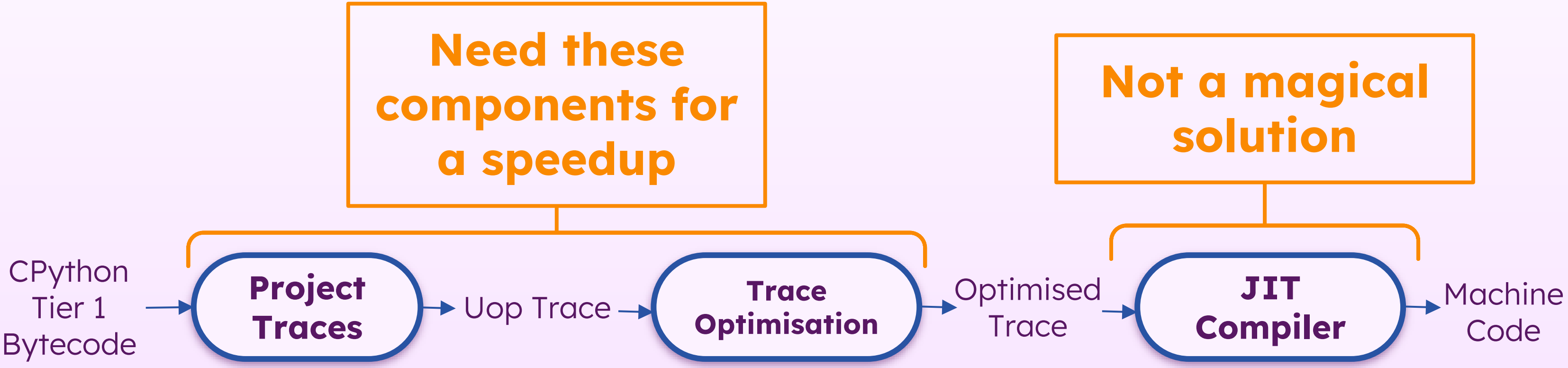
Overview of CPython 3.13 Optimisation Pipeline

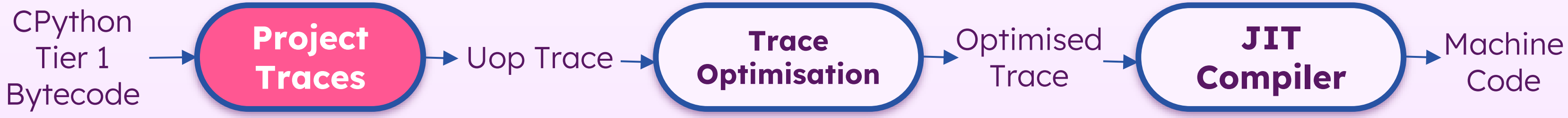


Overview of CPython 3.13 Optimisation Pipeline



Overview of CPython 3.13 Optimisation Pipeline




CPython
Tier 1
Bytecode
Project
Traces
Uop Trace
Trace
Optimisation
Optimised
Trace
JIT
Compiler
Machine
Code

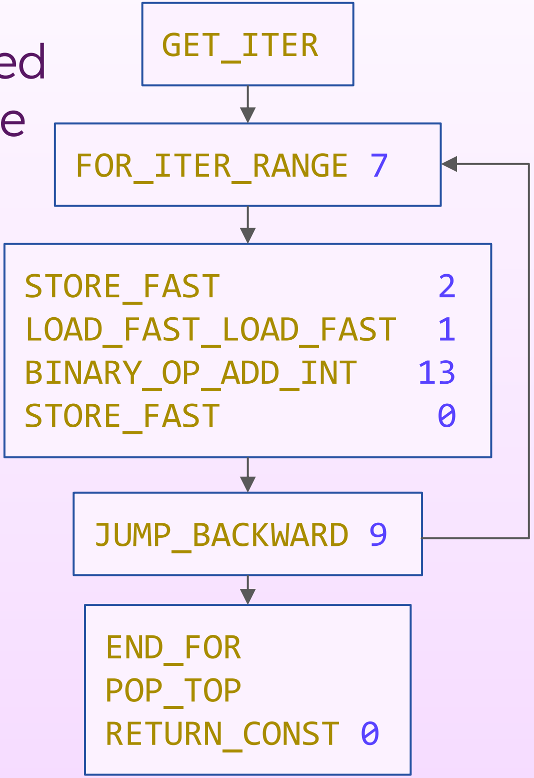
CPython 3.13

Project Traces

Python Source

```
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
```

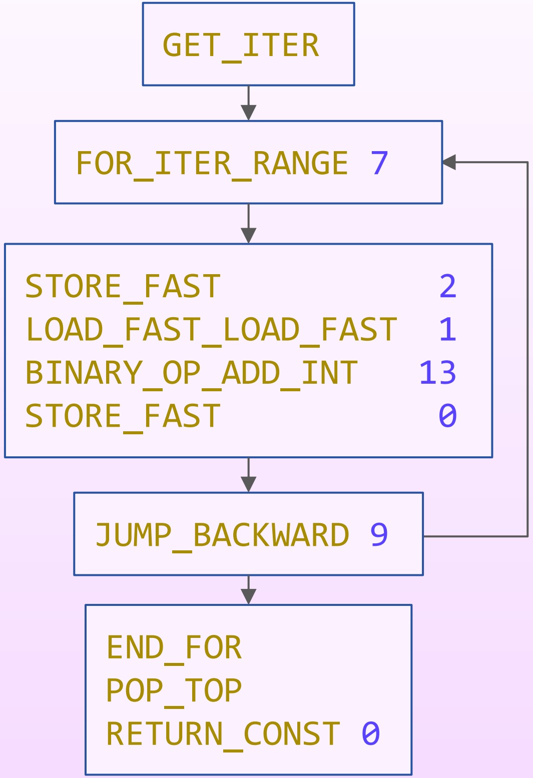
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted



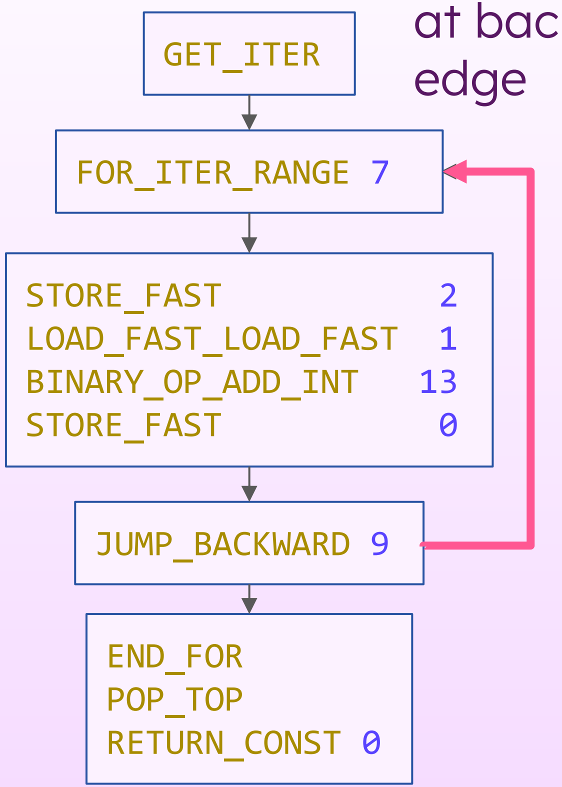
CPython 3.13

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

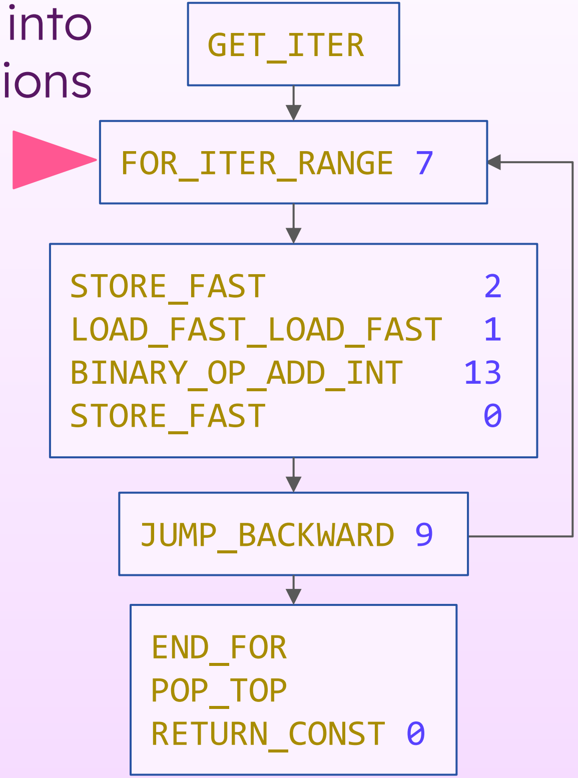
Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
```

Note:
`_CHECK_VALIDITY_AND_SET_IP` omitted

Bytecode gets broken down into smaller operations



```
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
```

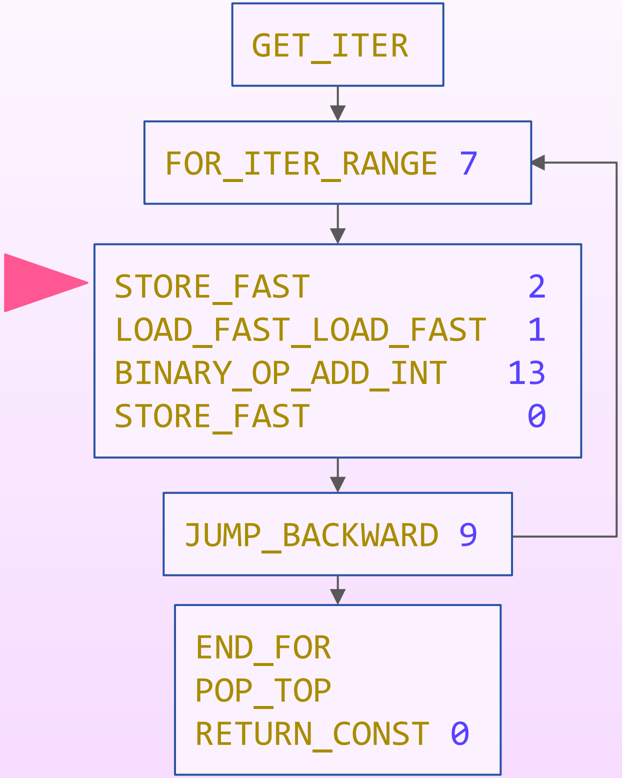
CPython 3.13

Project Traces

Tracing

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

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
```

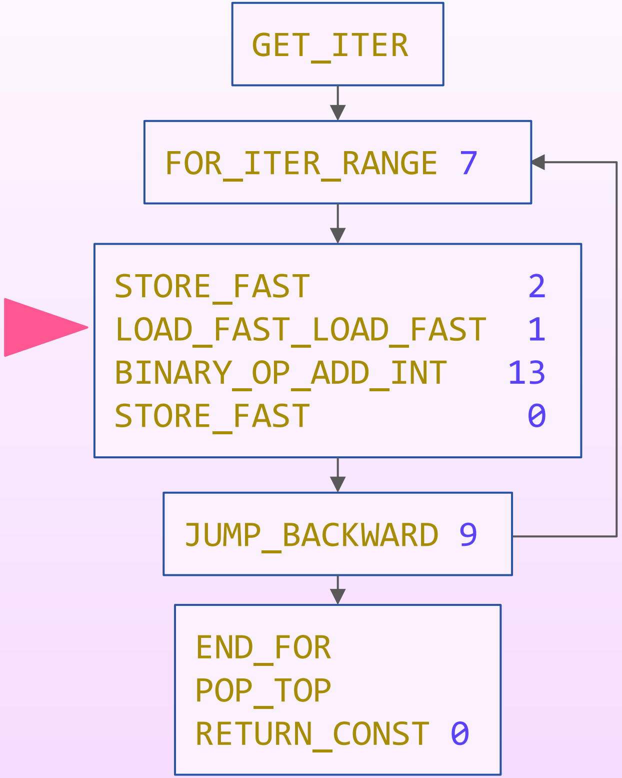
CPython 3.13

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



```
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
```

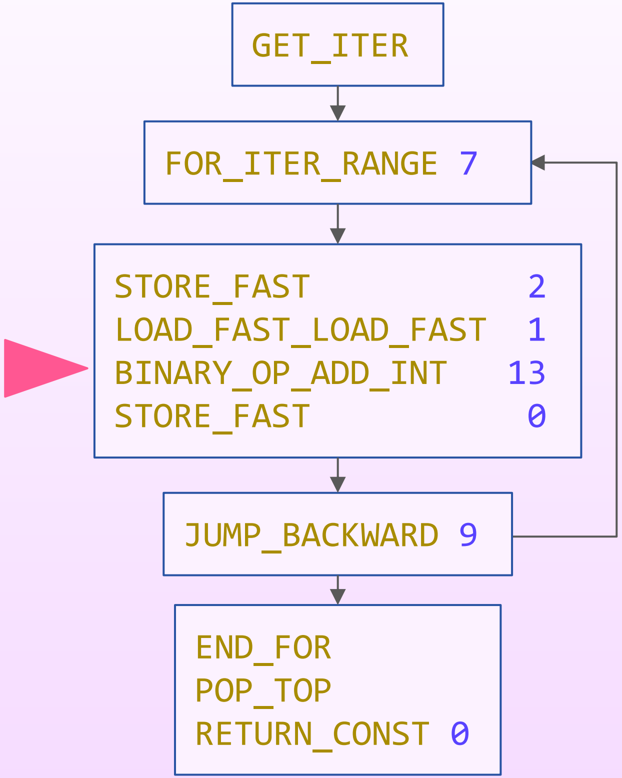
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_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



```
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
```

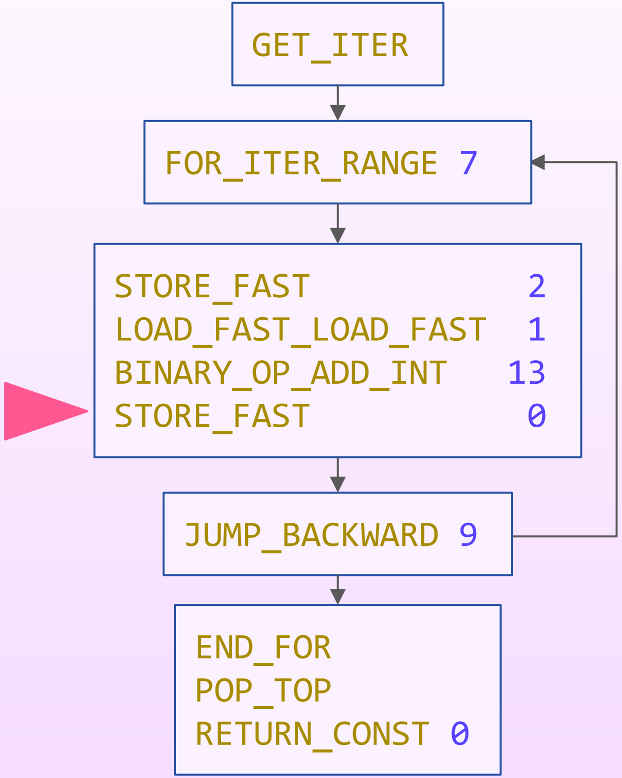
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_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



```
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

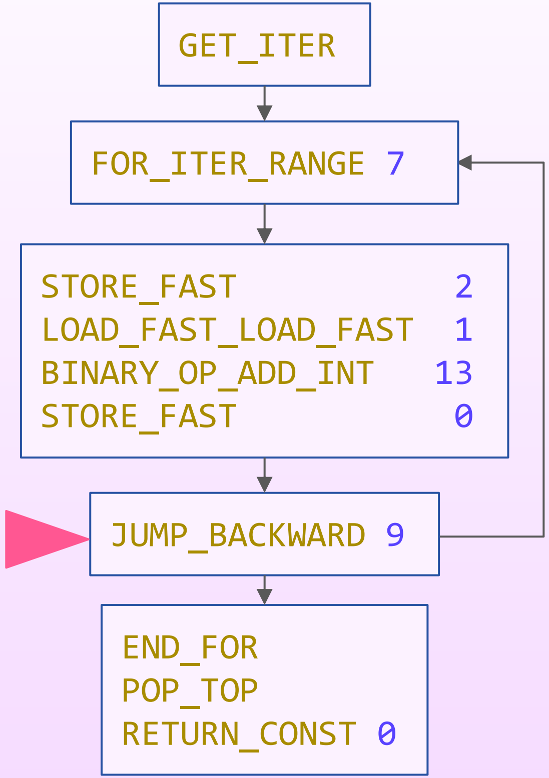
Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_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



```
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
```

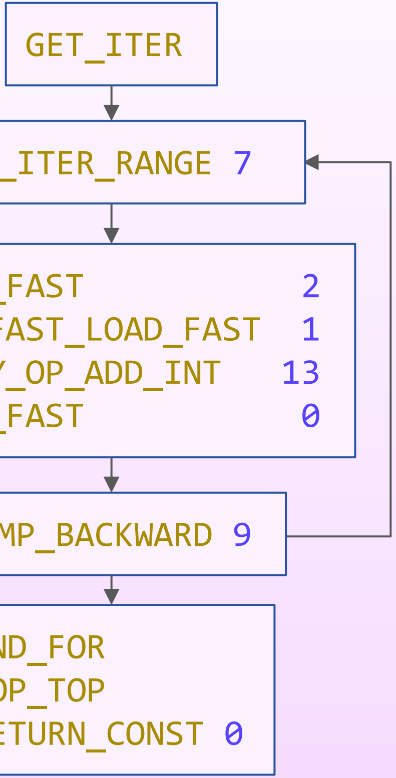
CPython 3.13

Project Traces

Tracing

```
_START_EXECUTOR
_TIER2_RESUME_CHECK
_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



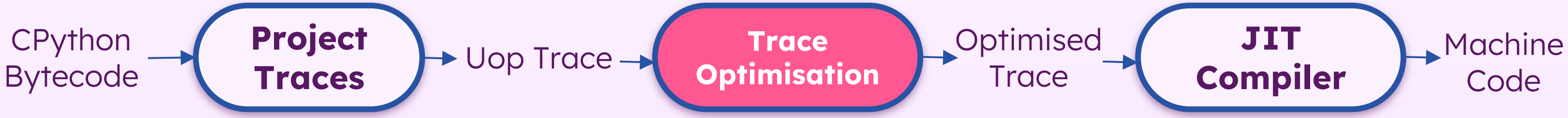
```
GET_ITER
```

```
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 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

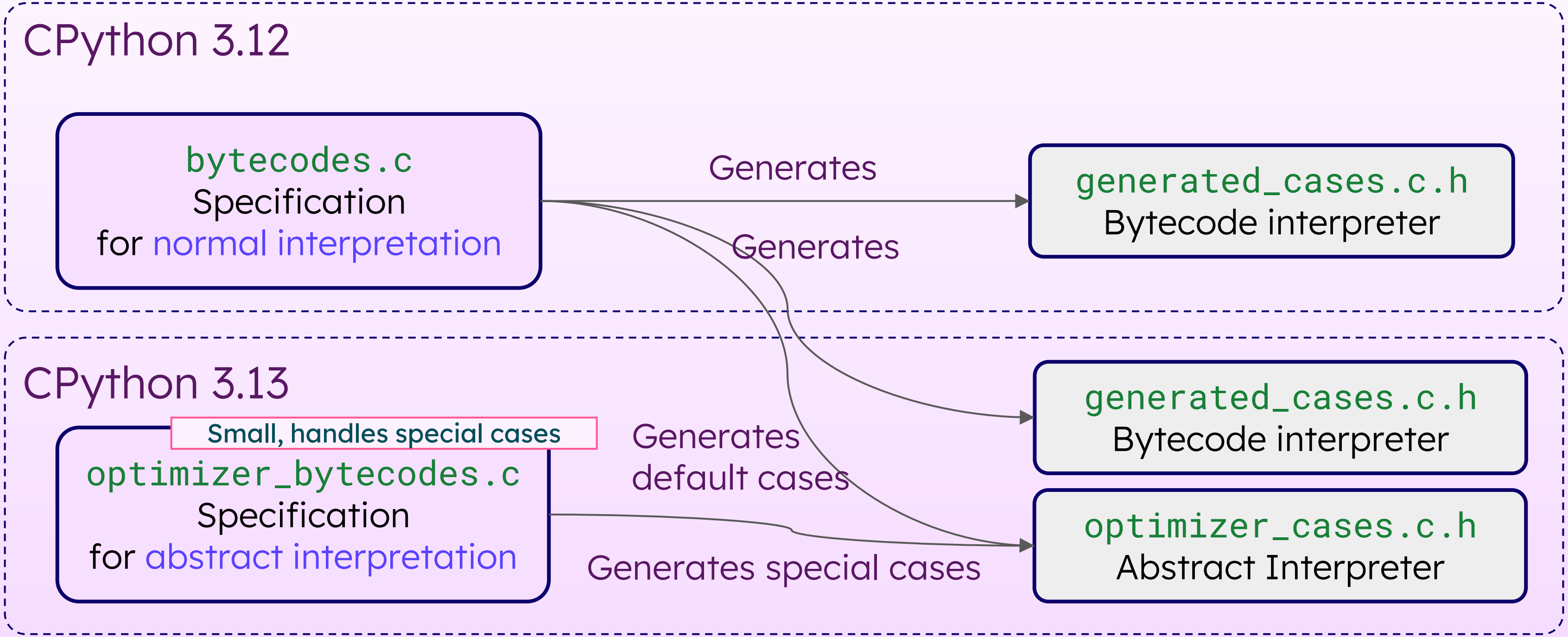
Solution:

CPython 3.12 introduced a DSL to specify the operations of bytecode

CPython 3.13 and Beyond

Trace Optimisation

Second pass: Abstract Interpretation

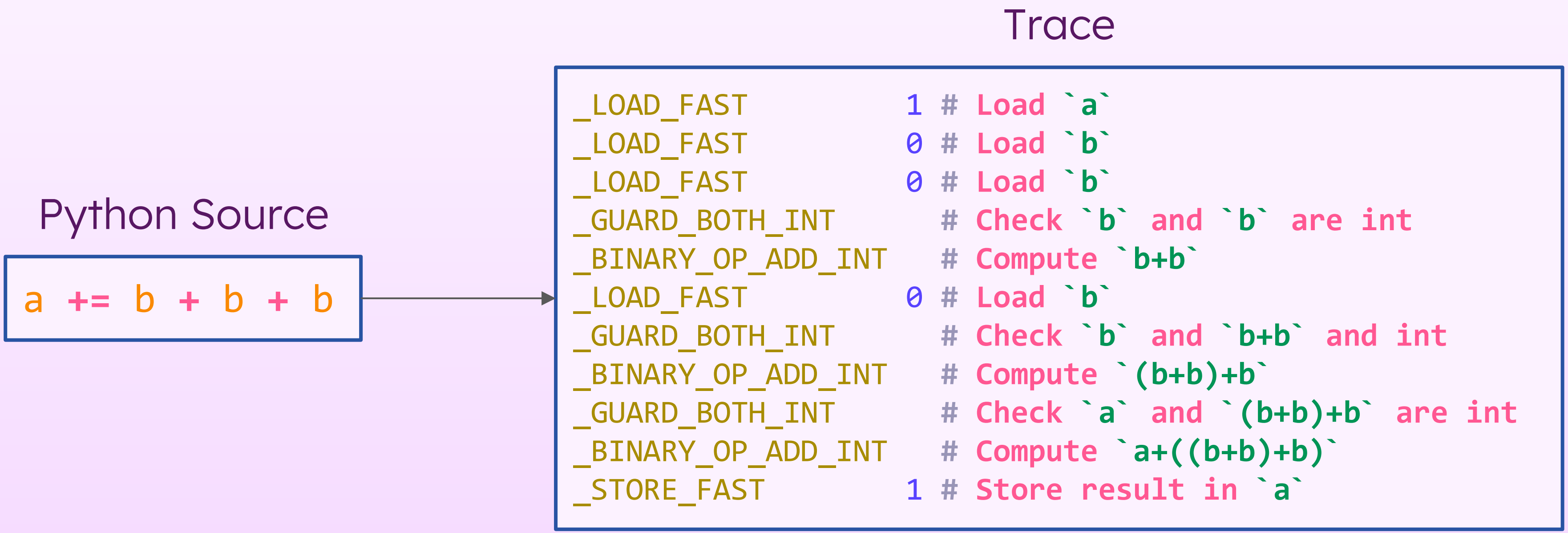


CPython 3.13 and Beyond

Trace Optimisation

Second pass: Guard Elimination (3.13 partially implemented)

Abstract Interpretation learns the types of each variable



Python Source

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

Trace

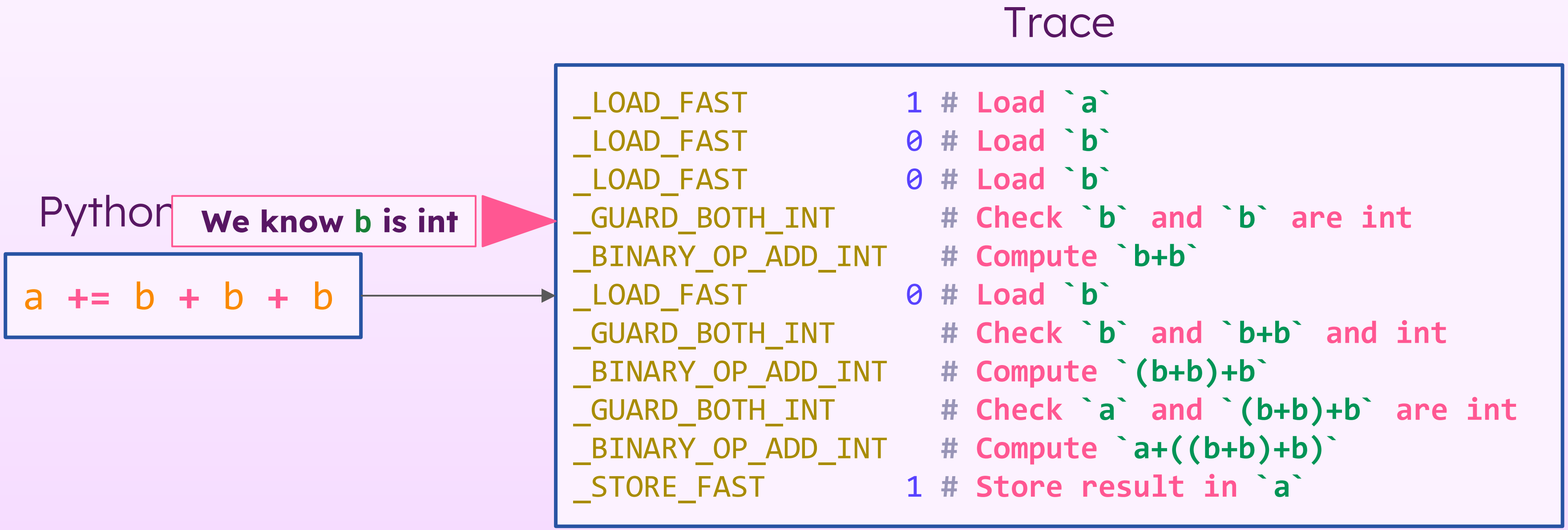
```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT    # Compute `(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

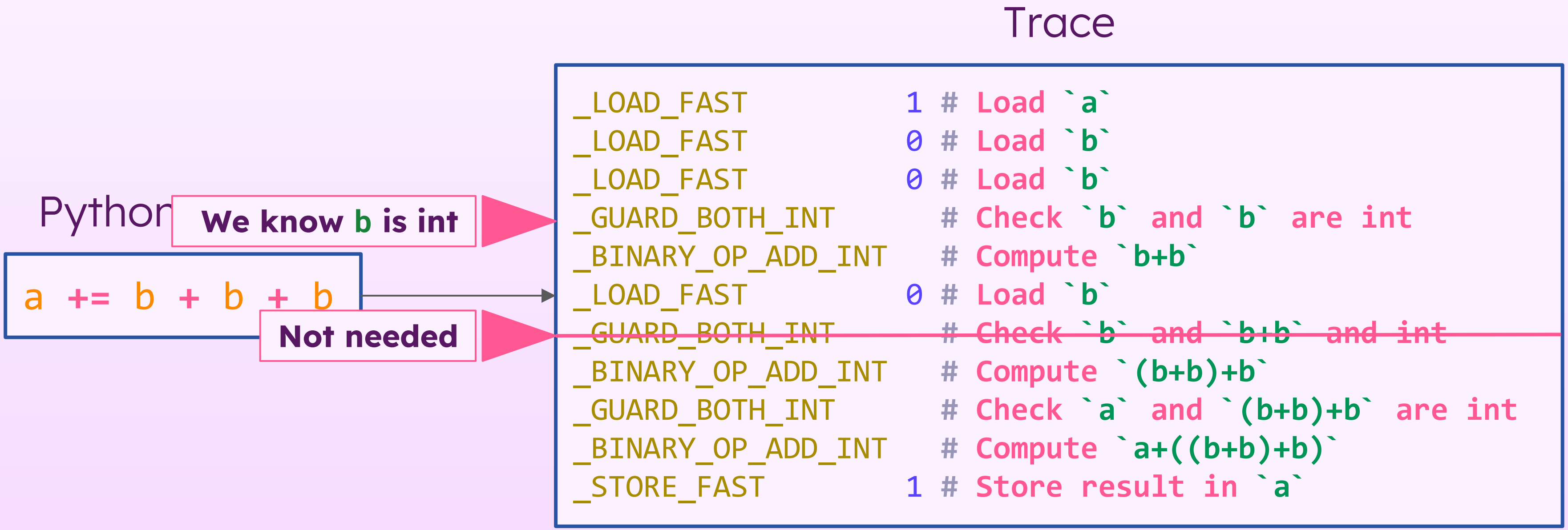
```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b+b` and int
_BINARY_OP_ADD_INT    # Compute `(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

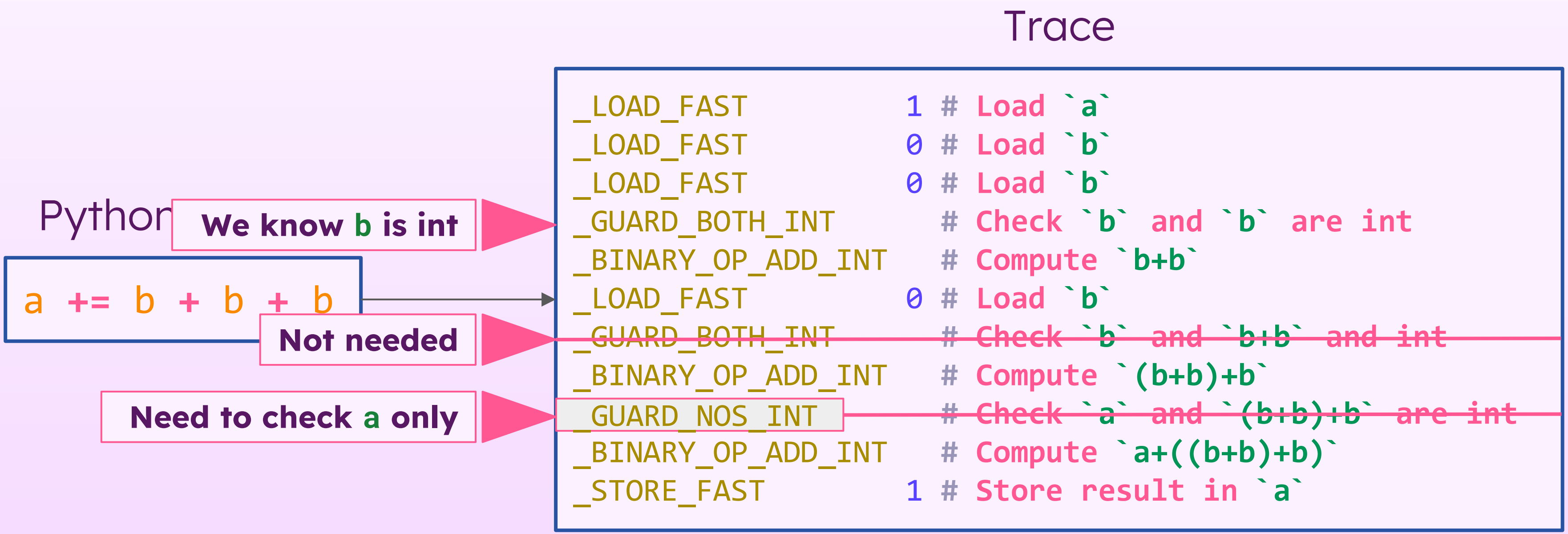
```
_LOAD_FAST          1 # Load `a`
_LOAD_FAST          0 # Load `b`
_LOAD_FAST          0 # Load `b`
_GUARD_BOTH_INT       # Check `b` and `b` are int
_BINARY_OP_ADD_INT    # Compute `b+b`
_LOAD_FAST          0 # Load `b`
~~_GUARD_BOTH_INT       # Check `b` and `b+b` and int~~
_BINARY_OP_ADD_INT    # 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

```
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)

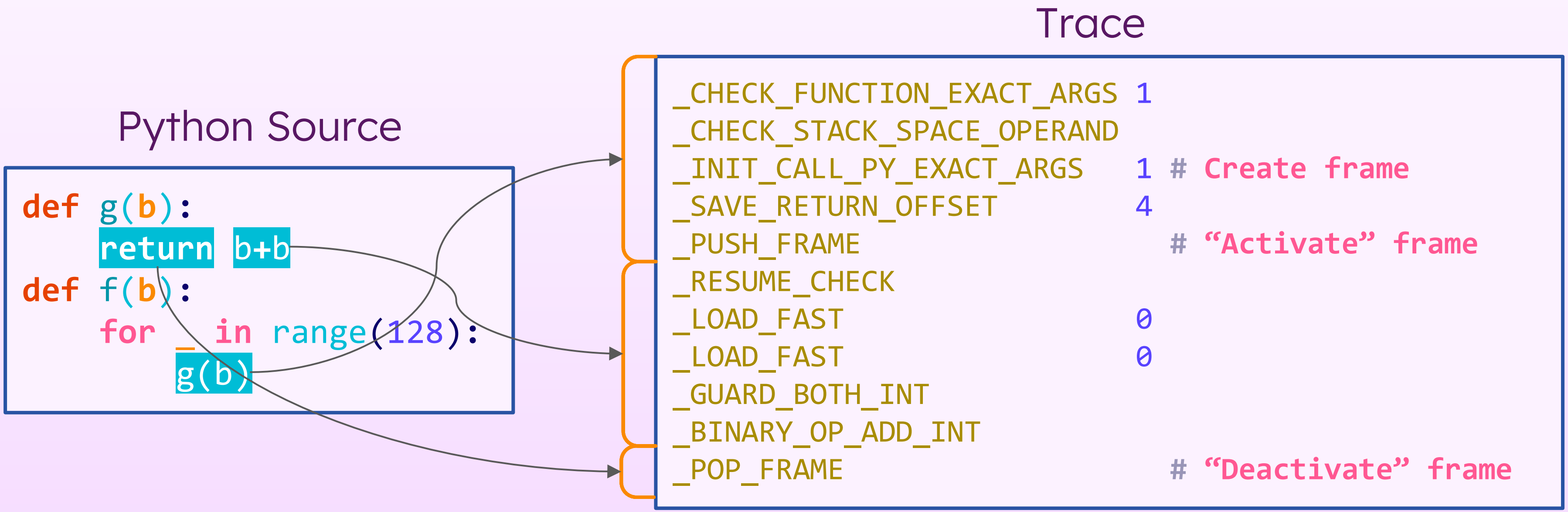
Idea:

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)
```

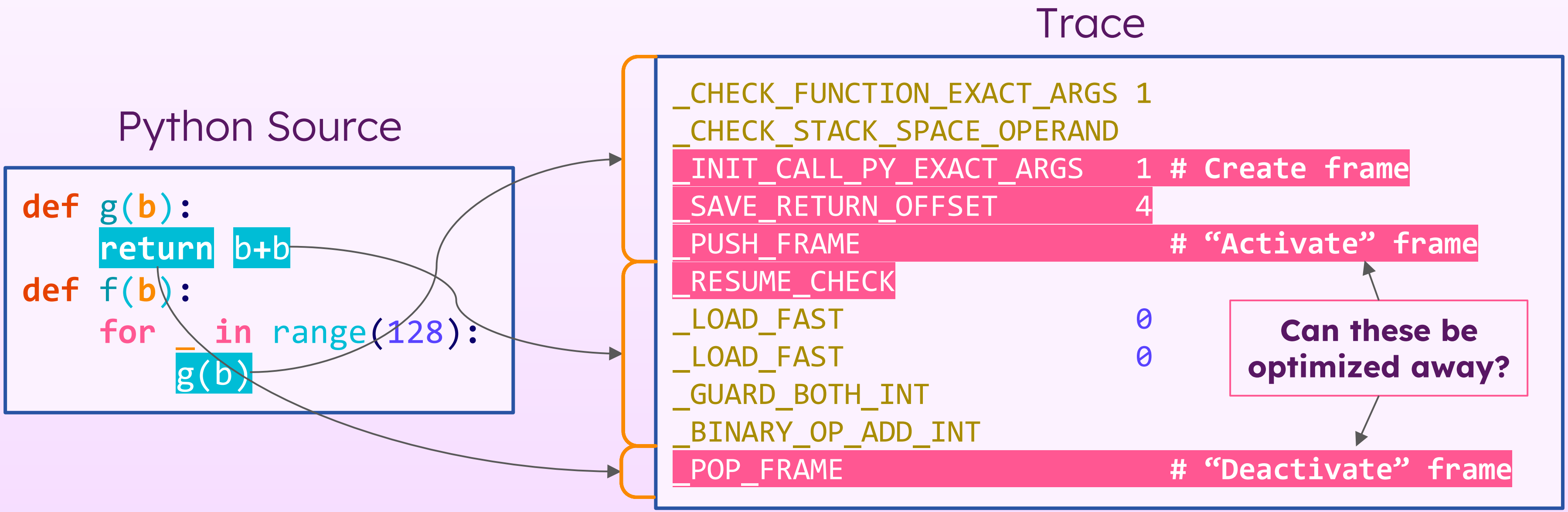
Trace

```
_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

```
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)

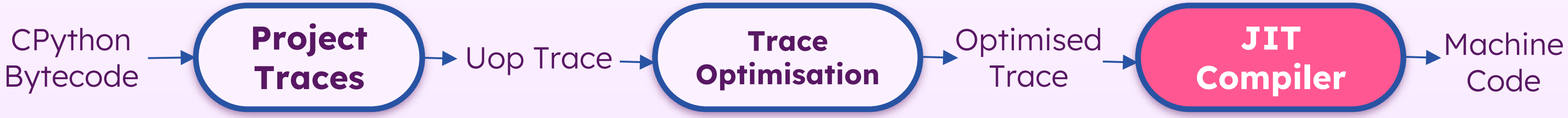
Currently worked on by Brandt Bucher.

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

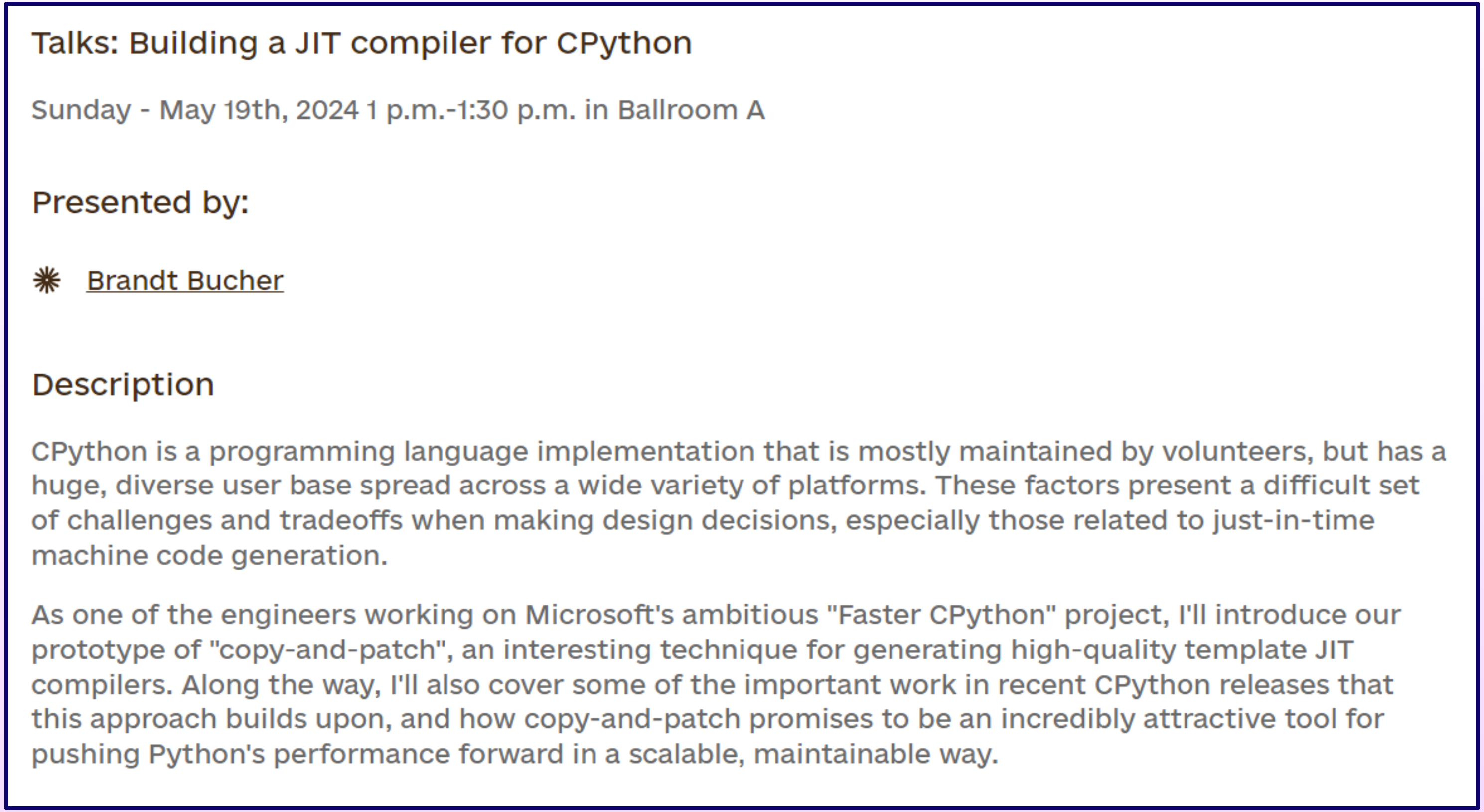
Optimised Trace

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)

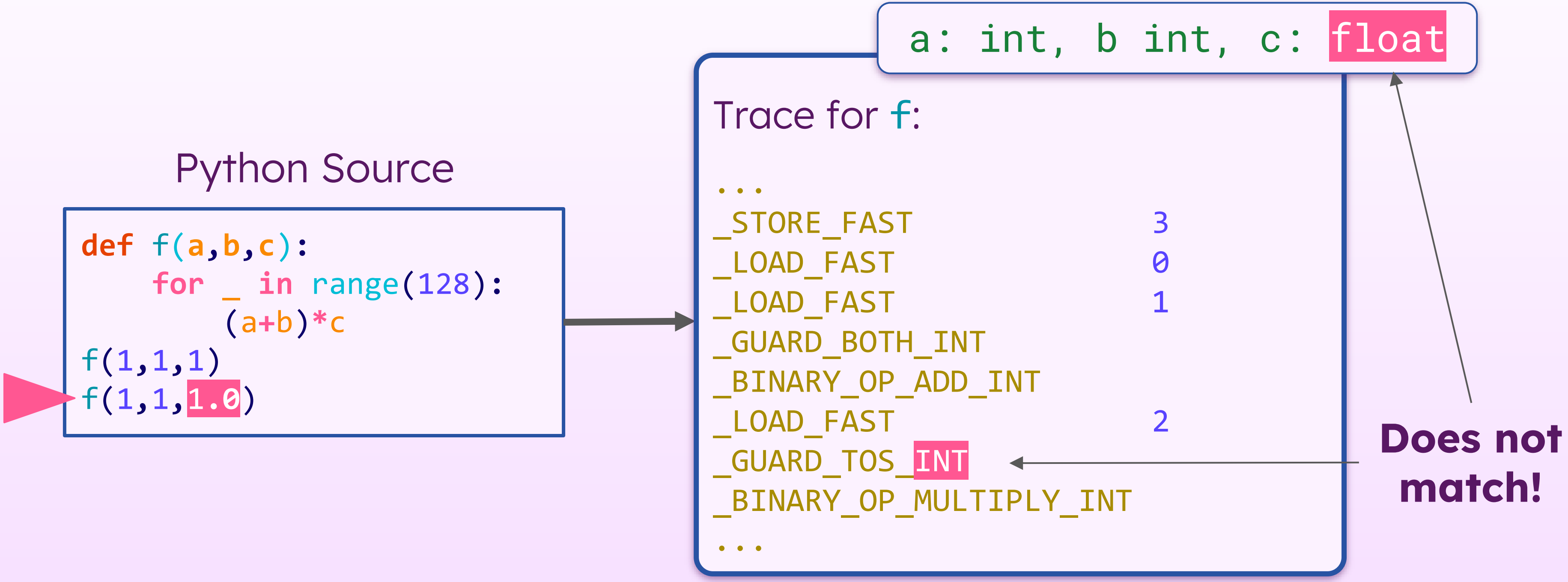
CPython 3.13

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)
```

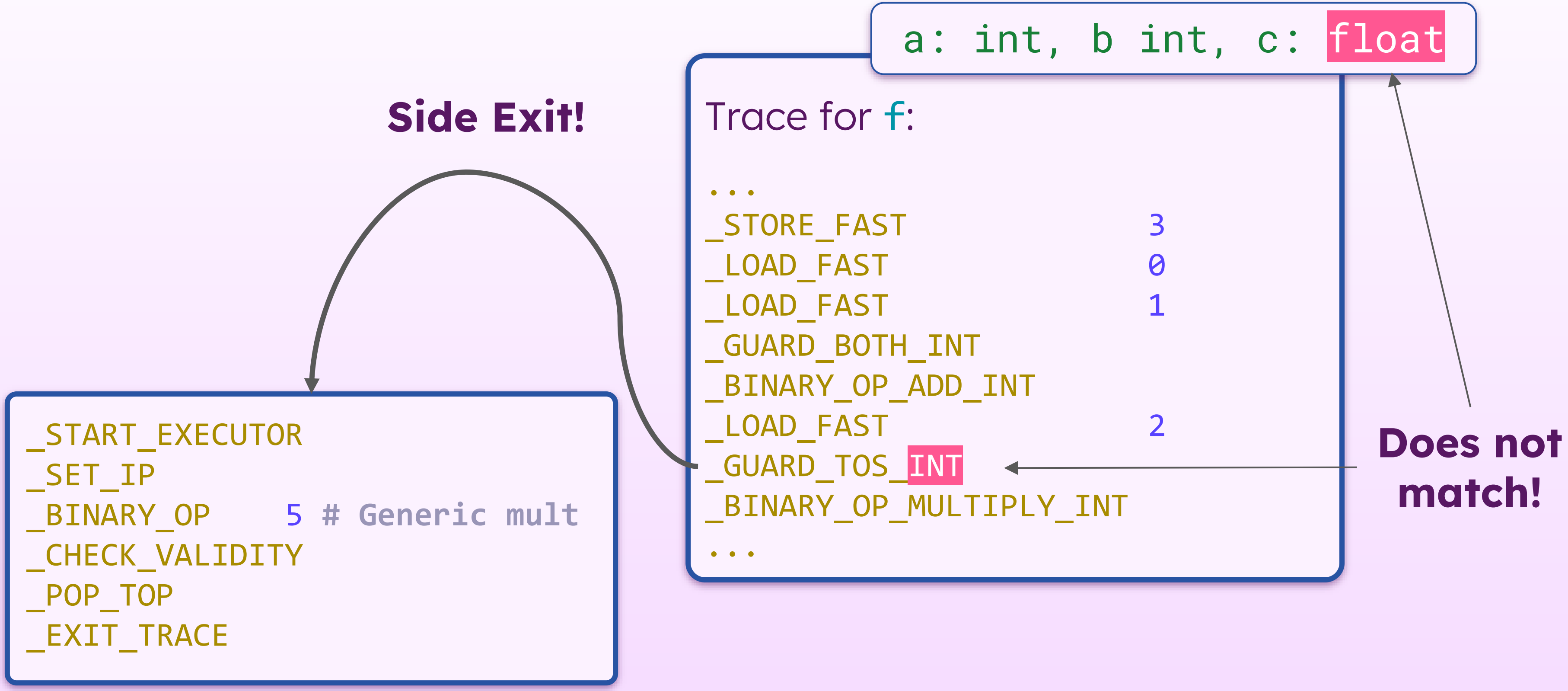
a: int, b int, c: float

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!

```
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!

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

CPython 3.13

De-Opts:

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

- Drop back to Tier 1

Thank You!

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