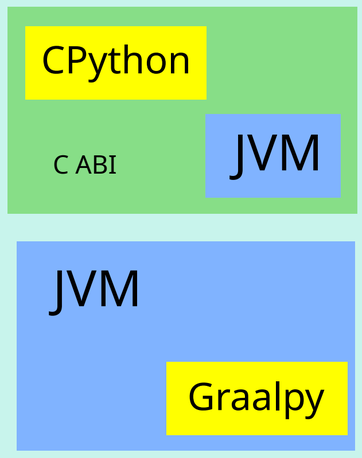
Excel@FIT 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

ORACLE

Adam Hrbáč

Improving the GraalPy Interpreter



GraalPy

- Python implementation in **Java**
- Easy **interop** with JVM
- Bytecode interpreter
- JIT compilation using the **Graal** compiler
- Loop unrolling for better compilation
- Not complete, features **missing**
  - Implemented **async** and **tracing**

Figure 1: Schemas of Java-Python interop with CPython and GraalPy.

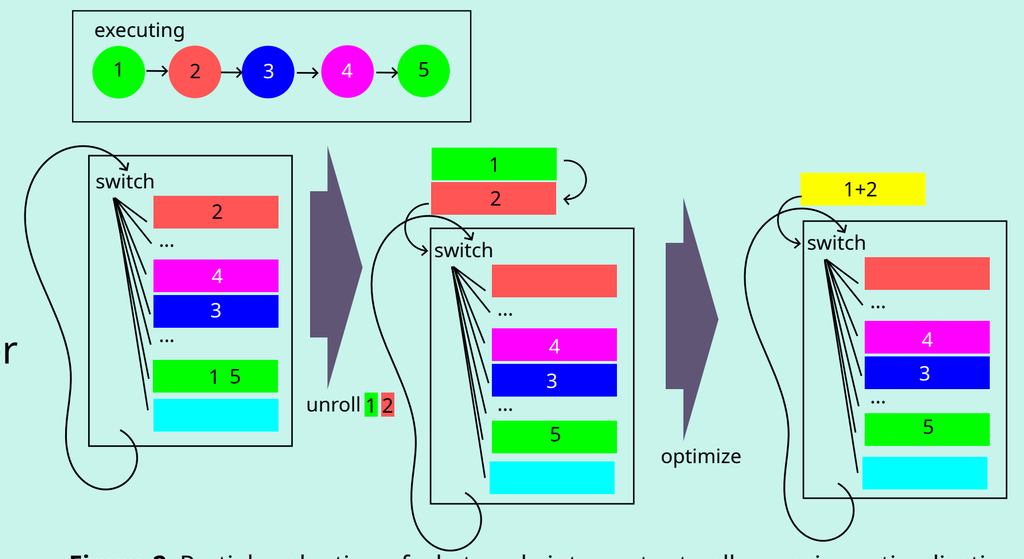


Figure 2: Partial evaluation of a bytecode interpreter to allow easier optimalization.

Async

- IO multiplexing
- **asyncio** library
  - **Event loops** using select/poll/epoll/...
- **syntax**
  - **Colored** functions
  - Alternatives to synchronous constructs - **for**, **with**

```
def fn():
    return 1

async def fn2():
    return 2

async def fn3():
    print(fn())
    print(await fn2())
```

Figure 3: Colored functions.

```
while await resps.has_next():
    process(await resps.next())

async for resp in resps:
    process(resp)
```

Figure 4: Async for loops.

```
client = Client()
await client.open()
...
await client.close()

async with Client() as client:
    ...
```

Figure 5: Async context managers.

```
def coro():
    yield future
    yield future2
    return result
```

EVENT LOOP

Figure 6: Transfer of control flow between coroutines and event loop.

Hash-Array Mapped Trie

- Variant of an **associative array**
- **Immutable** data structure - copy on write
- **Self-balacing** tree via **hashing**
- Used for **Task-local** state

Results

- **230/308** tracing tests pass
- **121/163** async syntax tests pass
- **Flask** and **httpx** work
- **pdb** can be used to debug code

This work was developed under the management of Lukas Stadler and supervision of David Kozák, as part of my internship at Oracle Labs.

Tracing

- **CPython** debugger API
- De-facto standard for **Python debugging**
- Also used for **coverage**
- Callback for each executed line
- Need to detect line execution from **bytecode**

```
if False:
    print(0)
print(1)
```

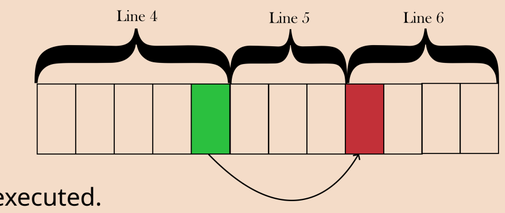


Figure 7: Detecting a new line being executed.

```
while cond(): body()
```

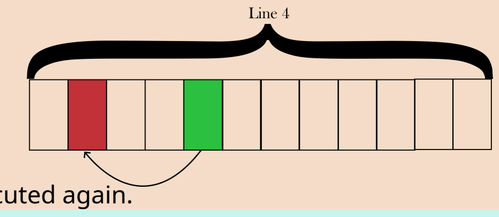


Figure 8: Detecting the same line executed again.

```
if True:
    print(1)
else:
    print(0);return None
```

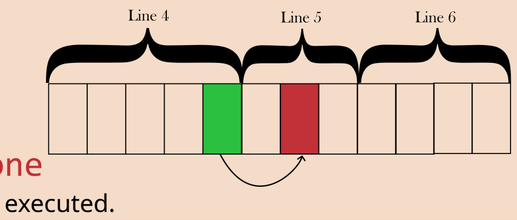


Figure 9: Avoiding lines not actually executed.

```
with open(PATH) as file:
    print("About to run", file=file)
    for name in names:
        print("Can't jump here", file=file)
    print("Can jump here", file=file)
log.info("Or here")
```

Figure 10: Detecting jumps that do not cause a stack underflow.

begin
- $Q \longleftarrow \{0\}$
- $R \longleftarrow \{(0, \bot)\}$
- **while** $Q \neq \emptyset$ **do**
  - $i \longleftarrow pop(Q)$
  - $assert\ \forall S'.iRS' \Rightarrow S = S'$
  - **if** $(i, \times, h, l) \in H$ **then**
    - $assert\ l \leq length(S)$
    - create S' by removing items from S until $length(S) = l$
    - $S' \longleftarrow (Exc, S')$
    - $R \longleftarrow \{(h, S')\} \cup R$
    - $Q \longleftarrow Q \cup \{h\}$
  - **if** $op(B, i) =$ *SWAP* **then**
    - $(a, (b, S)) \longleftarrow S$
    - $R \longleftarrow \{(next(B, i), (b, (a, S)))\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
  - **else if** $op(B, i) =$ *GET_ITER* **then**
    - $R \longleftarrow \{(next(B, i), (Iter, S))\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
  - **else if** ... **then**
    - other operations with special handling
  - **else if** $hasNext(B, i)$ **then**
    - create S' by removing $nRemoved(B, i)$ items from S
    - **for** $\times \leftarrow 1$ **to** $nAdded(B, i)$ **do**
      - $S' \leftarrow (Obj, S')$
    - $R \longleftarrow \{(next(B, i), S')\} \cup R$
    - $Q \longleftarrow Q \cup \{next(B, i)\}$
    - **if** $hasJump(B, i)$ **then**
      - create S' by removing $nRemovedWithJump(B, i)$ items from S
      - **for** $\times \leftarrow 1$ **to** $nAddedWithJump(B, i)$ **do**
        - $S' \leftarrow (Obj, S')$
      - $R \longleftarrow \{(nextWithJump(B, i), S')\} \cup R$
      - $Q \longleftarrow Q \cup \{nextWithJump(B, i)\}$
- **return** R

Algorithm 1: Analysis of bytecode for the types of stack items.

Jumps

- **Non-deterministic** control flow
- Used by debuggers
- Could **crash** the interpreter
- Avoid by checking source and destination