Partial Evaluation, Whole-Program Compilation

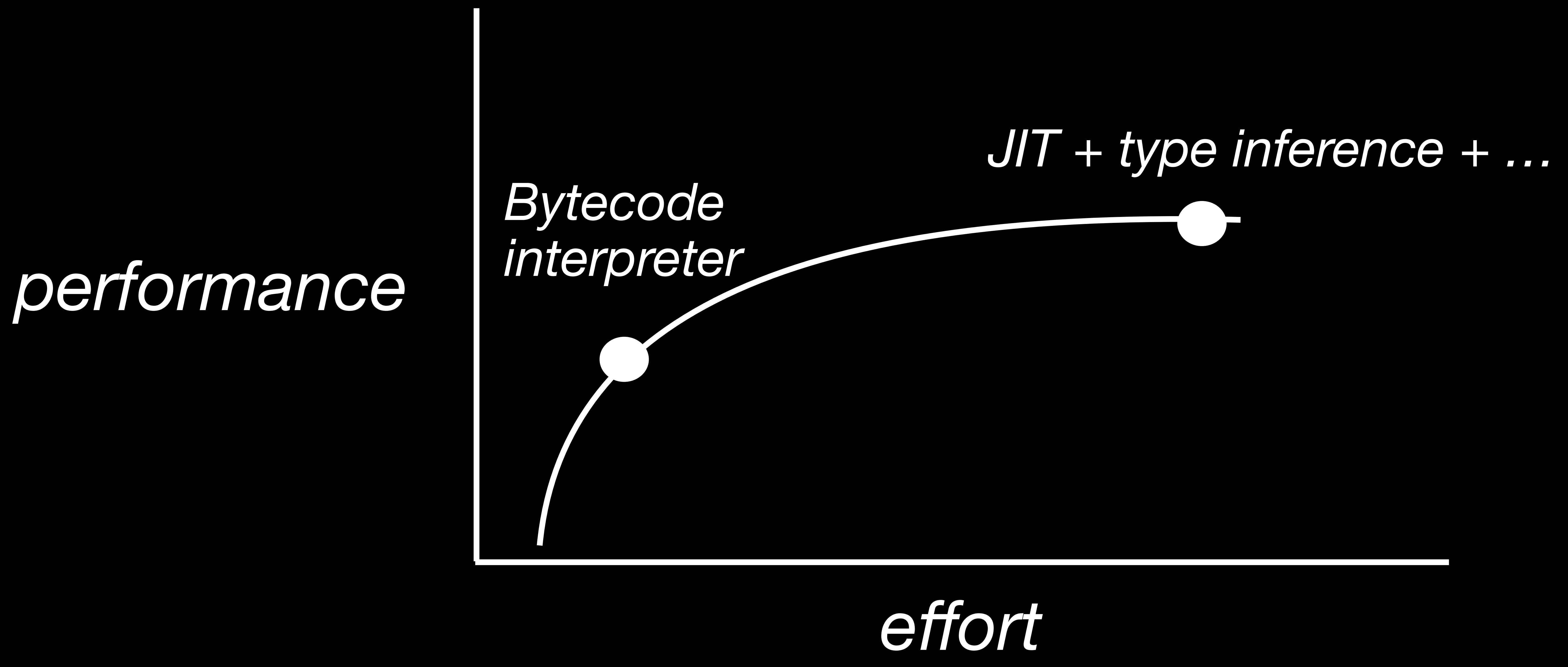
Chris Fallin (F5), Maxwell Bernstein (Recurse Center)

Why Interpreters for Dynamic Languages?

Bytecode

```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```

- Bytecode interpreters are a *sweet spot* between simplicity and production-ready performance



Why Interpreters for Dynamic Languages?

Bytecode

```
GETLOCAL 0    # x
GETLOCAL 1    # y
ADD
RETURN
```

- Bytecode interpreters are a *sweet spot* between simplicity and production-ready performance
- Interpreters are *very portable*

Why Interpreters for Dynamic Languages?

Bytecode

```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```

- Bytecode interpreters are a *sweet spot* between simplicity and production-ready performance
- Interpreters are *very portable*

Context for this project:

- ported SpiderMonkey to run *inside* a Wasm module (Function-as-a-Service platform)

Interpreter + Compiler(s)

Interpreter

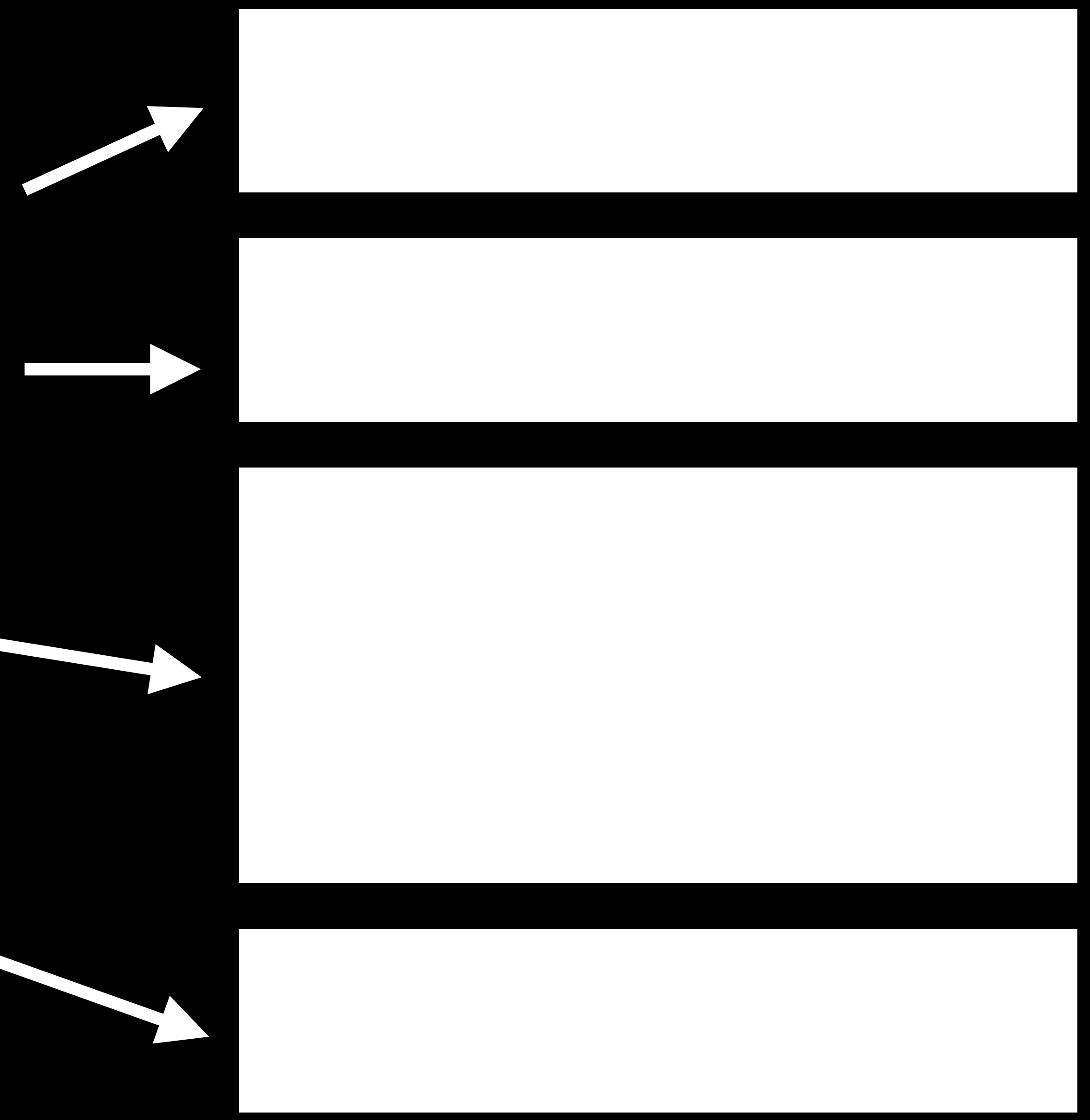
```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```

Interpreter + Compiler(s)

Interpreter

Baseline Compiler

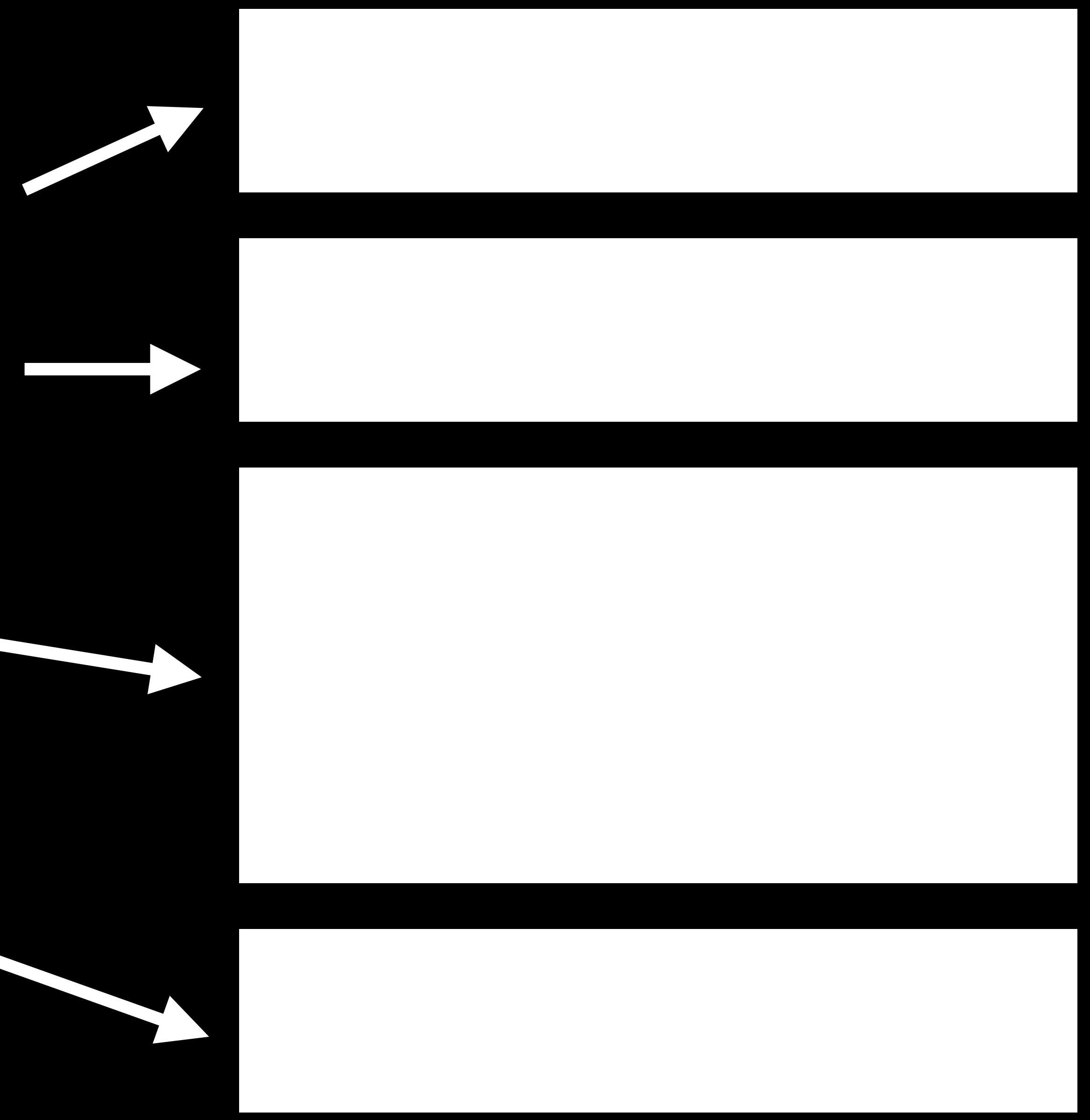
```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```



Interpreter + Compiler(s)

Interpreter *Baseline Compiler* *Optimizing Compiler*

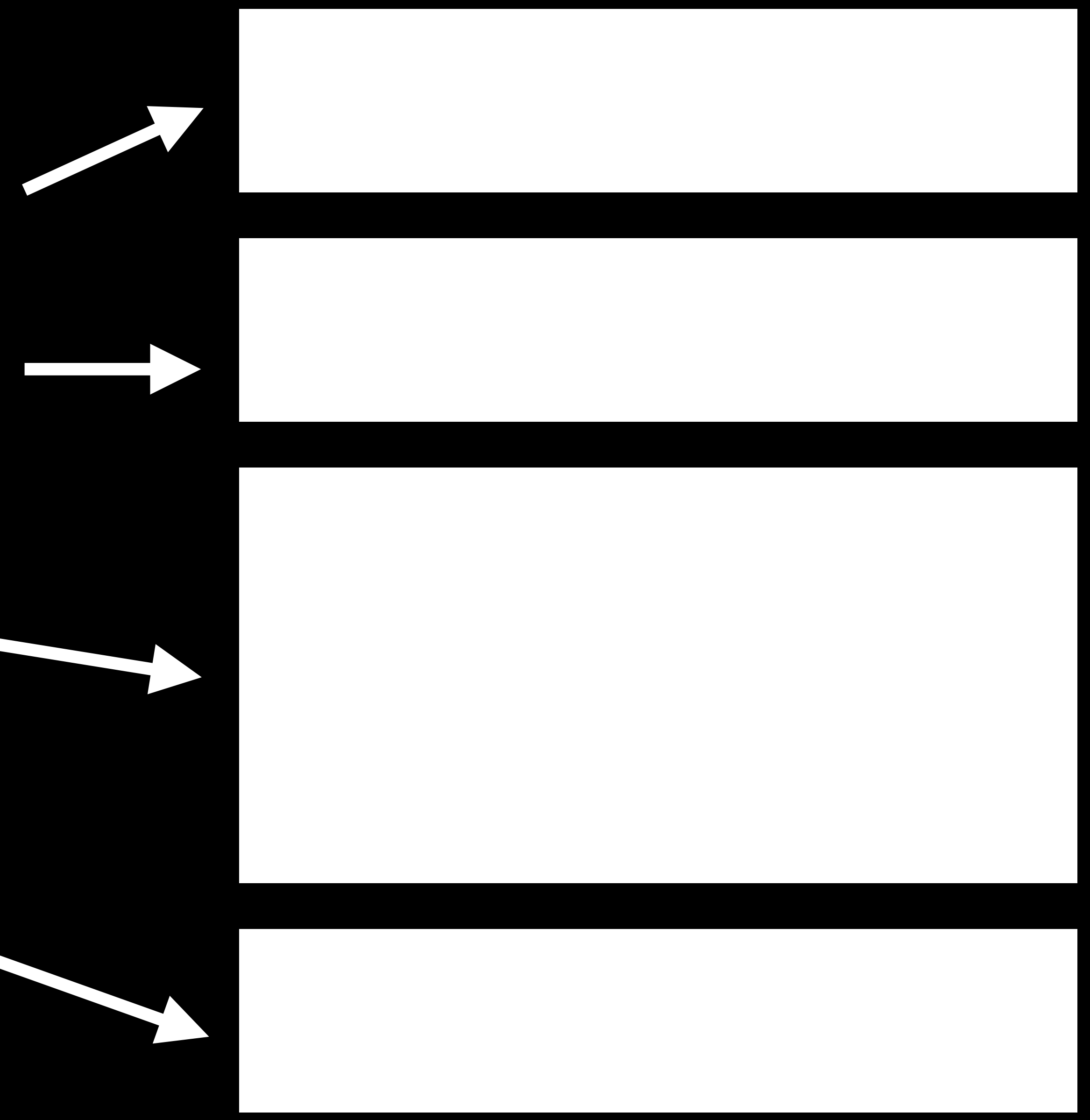
```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```



Interpreter + Compiler(s)

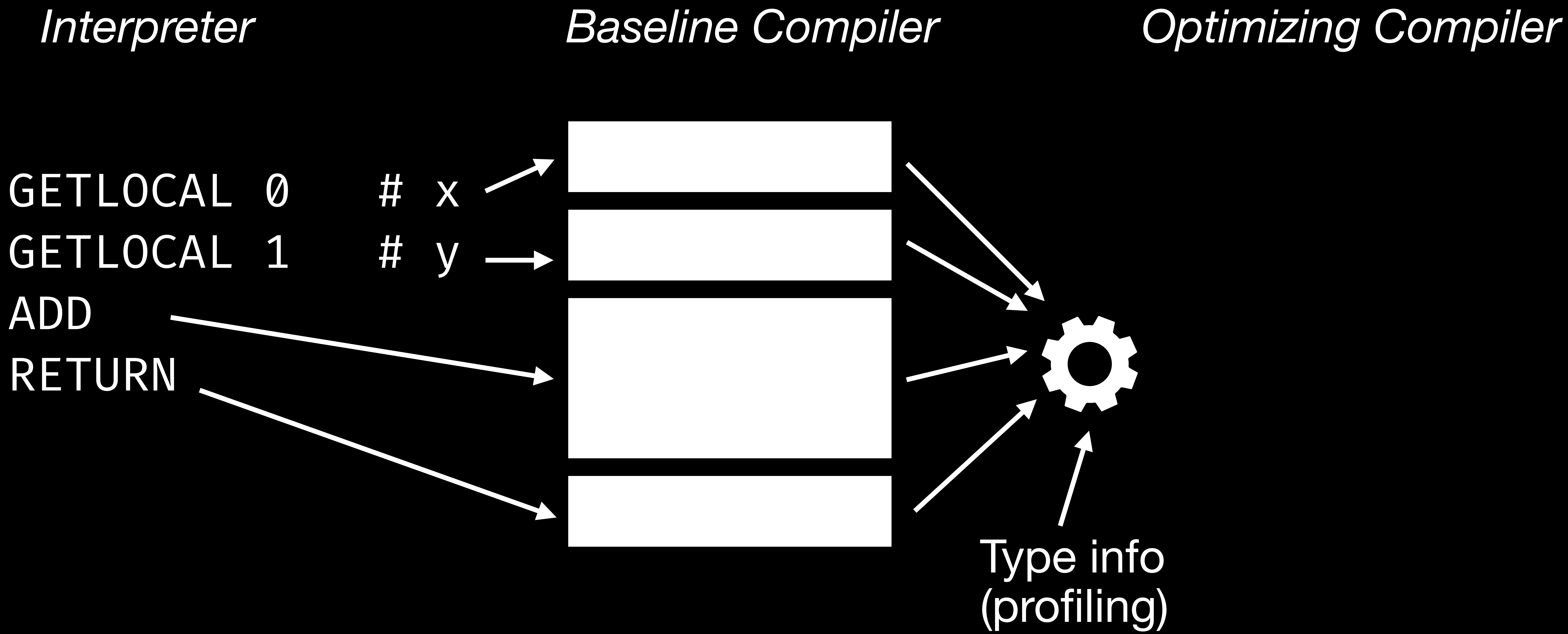
Interpreter *Baseline Compiler* *Optimizing Compiler*

```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```

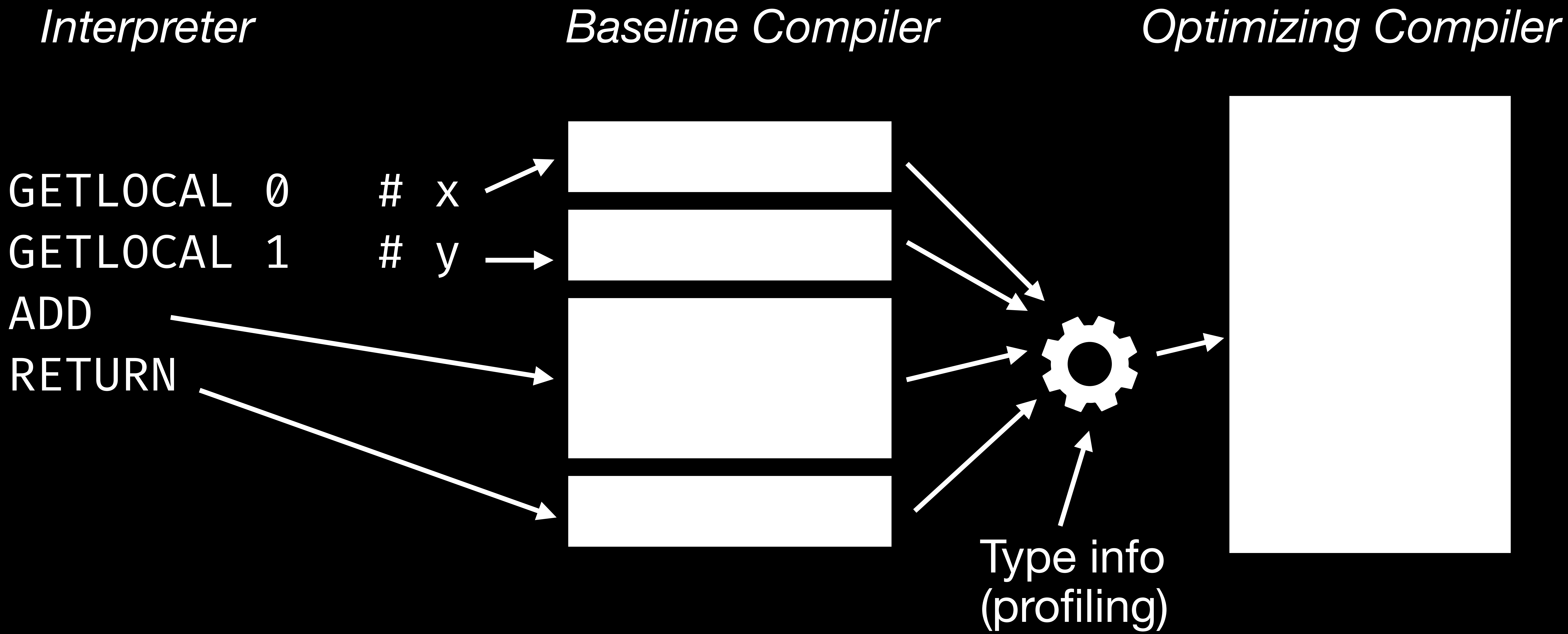


Type info
(profiling)

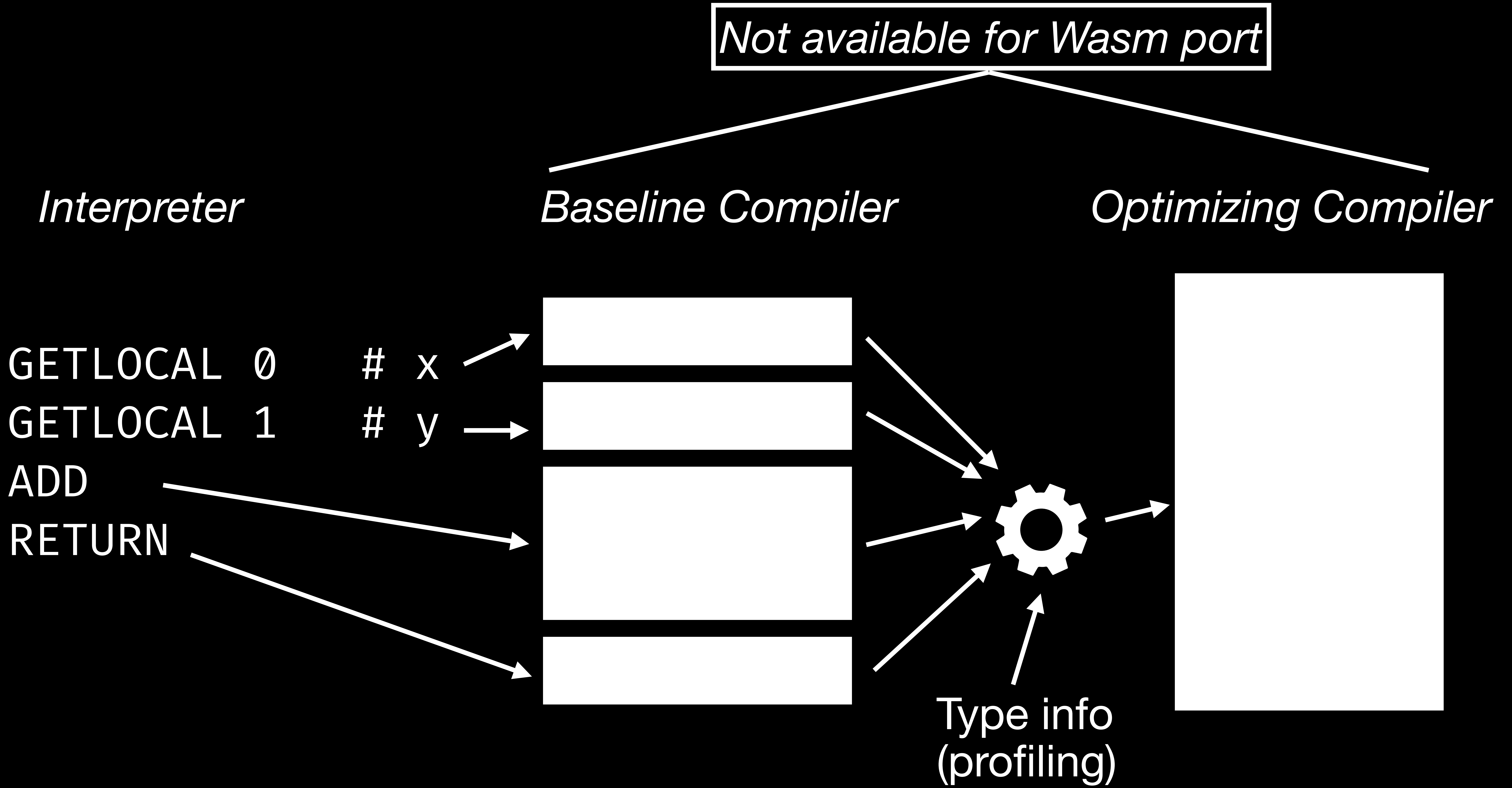
Interpreter + Compiler(s)



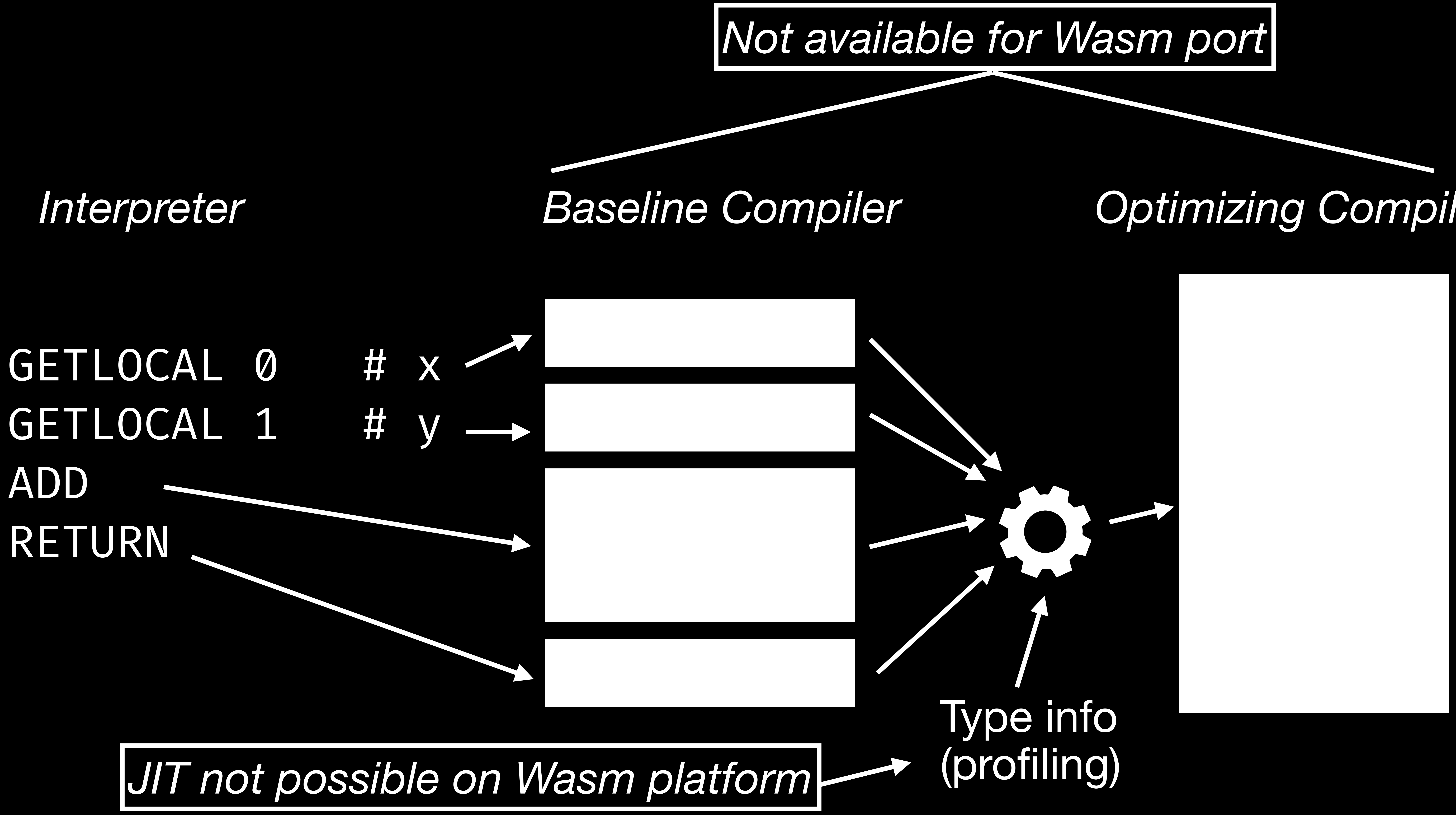
Interpreter + Compiler(s)



Interpreter + Compiler(s)



Interpreter + Compiler(s)



Interpreter + Compiler(s)

Interpreter

???

```
GETLOCAL 0   # x
GETLOCAL 1   # y
ADD
RETURN
```

Need to build new AOT compiler backend separate from existing JIT infrastructure

Opportunity: is there a better way?

Interpreter + Compiler(s)

Interpreter

Direct semantics

↓

Execute

Interpreter + Compiler(s)

Interpreter	*Baseline Compiler*
Direct semantics	Instructions implementing semantics
↓	↓
	Emit
	↓
Execute	Execute

Interpreter + Compiler(s)

Interpreter

Direct semantics

Execute

Baseline Compiler

Instructions implementing semantics

Emit

Execute

Optimizing Compiler

IR nodes implementing semantics

Optimize

Emit

Execute

Interpreter + Compiler(s)

Interpreter	*Baseline Compiler*	*Optimizing Compiler*
Direct semantics	Instructions implementing semantics	IR nodes implementing semantics

Repeated encodings of language semantics!

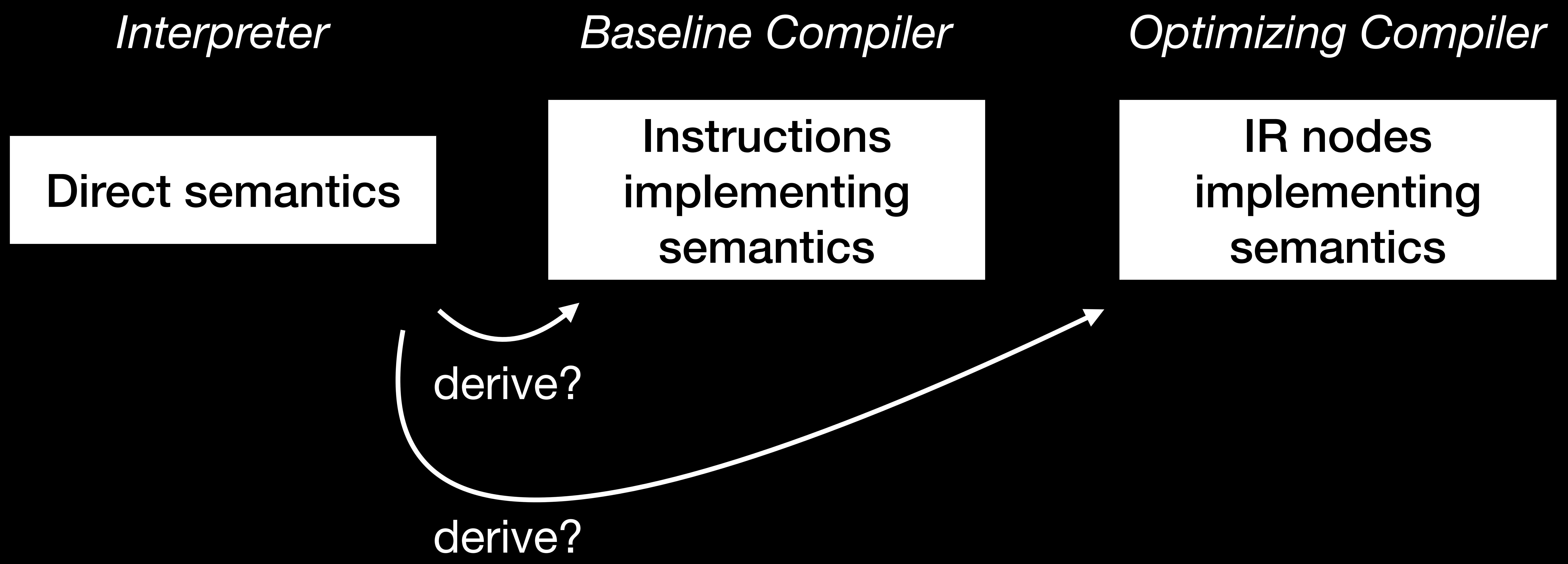
Ideal: Generate Compiler from Interpreter

Interpreter	*Baseline Compiler*	*Optimizing Compiler*
Direct semantics	Instructions implementing semantics	IR nodes implementing semantics

derive?

derive?

Ideal: Generate Compiler from Interpreter



- Single source of truth: reduces engineering effort substantially; eliminates mismatches by construction

Ideal: Generate Compiler from Interpreter

Interpreter	*Baseline Compiler*	*Optimizing Compiler*
Direct semantics	Instructions implementing semantics	IR nodes implementing semantics

derive?

derive?

(First) Futamura Projection: *partially evaluate* an interpreter with a program to perform a compilation

Past Work

- Partial evaluation: **Truffle/GraalVM** [PLDI'17]
  - Works by *constant-propagating AST nodes* and *inlining* (up to boundary)
  - Requires rewriting interpreter into Truffle framework
  - Currently works only at run-time; unclear if AOT is possible

Past Work

- Partial evaluation: **Truffle/GraalVM** [PLDI'17]
    - Works by *constant-propagating AST nodes* and *inlining* (up to boundary)
    - Requires rewriting interpreter into Truffle framework
    - Currently works only at run-time; unclear if AOT is possible
- Metatracing: **PyPy/RPython** [ICOOOLPS'09]
    - Works by *tracing interpreter over bytecode* and *compiling the trace*
    - Requires rewriting interpreter into RPython
    - Can only operate at run-time; requires observing interpretation

Goal

Goal: practical full-method, AOT partial evaluation for *existing bytecode interpreters*.

- **Our contribution**: a tool (*weval*) that partially evaluates interpreters compiled to WebAssembly; and an adaptation of SpiderMonkey to do AOT compilation using this tool.

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
switch(*pc) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    pc++;
    break;
  case RET:
    return pop();
}
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
switch(*pc) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    pc++;
    break;
  case RET:
    return pop();
}
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
switch(ADD) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    pc++;
    break;
  case RET:
    return pop();
}
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
switch(ADD) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    pc++;
    break;
  case RET:
    return pop();
}
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

+ branch folding

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
auto a = pop();
auto b = pop();
push(a + b);
pc++;
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

+ branch folding

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
auto a = pop();
auto b = pop();
push(a + b);
pc++;
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

+ branch folding

✅ *Partial evaluation works!*

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
switch(*pc) {
  case ADD:
    auto a = pop();
    auto b = pop();
    push(a + b);
    pc++;
    break;
  case RET:
    return pop();
}
```

Constant propagation

with known state:

`*pc` is `Constant(ADD)`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

Constant propagation

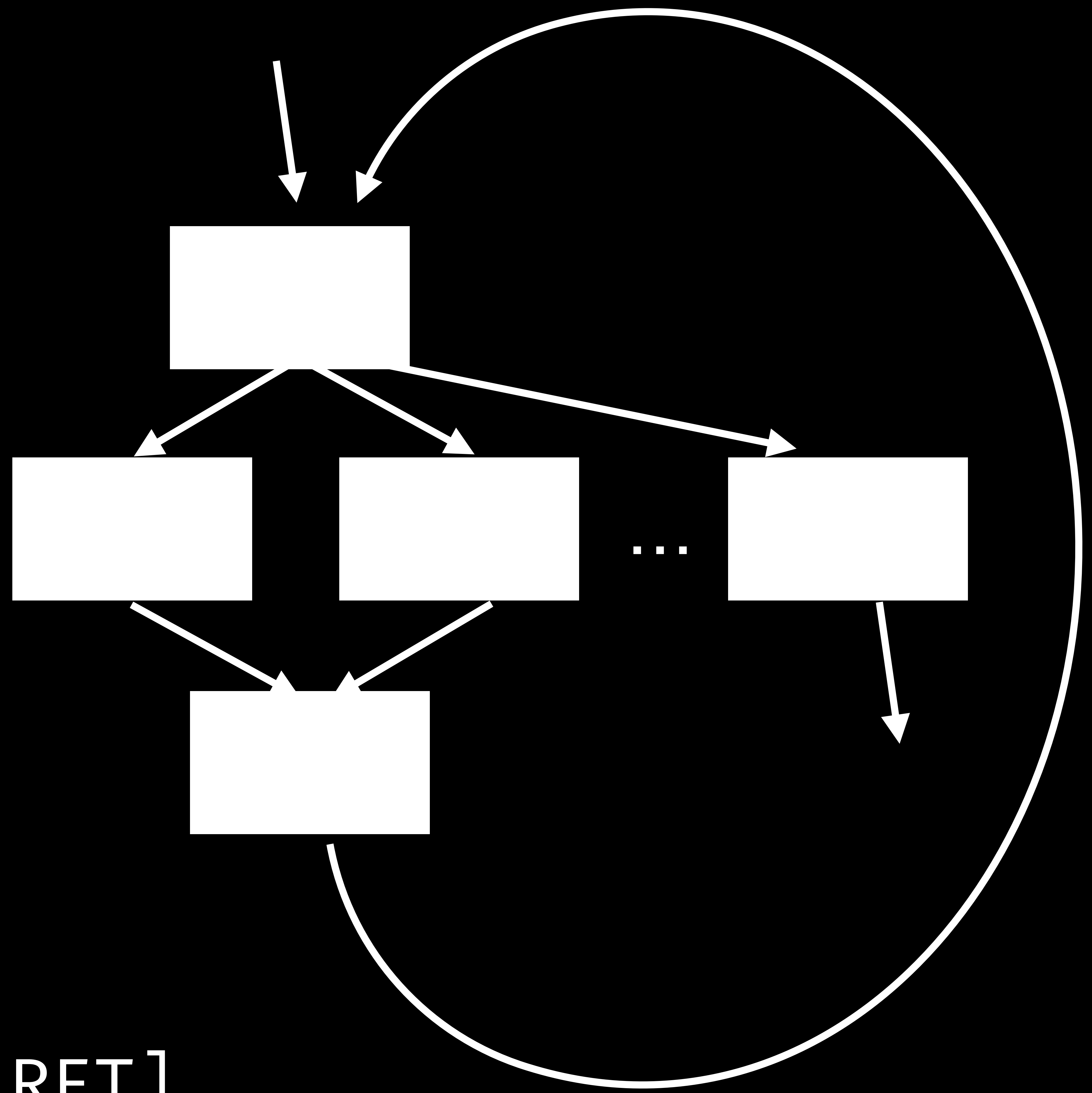
with known state:

code is `[ADD, RET]`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```



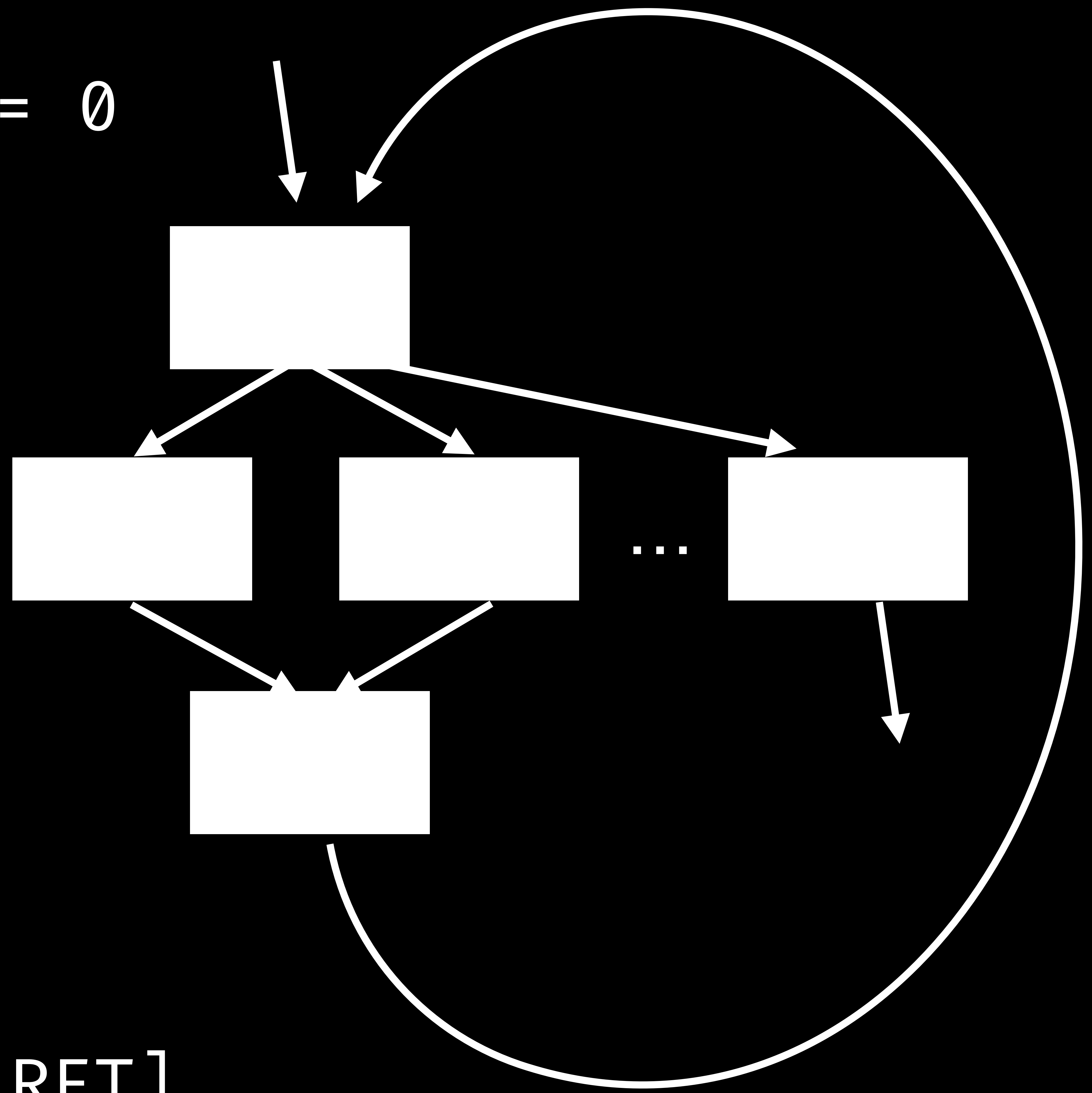

```
code = [ADD, RET]
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

`pc = 0`

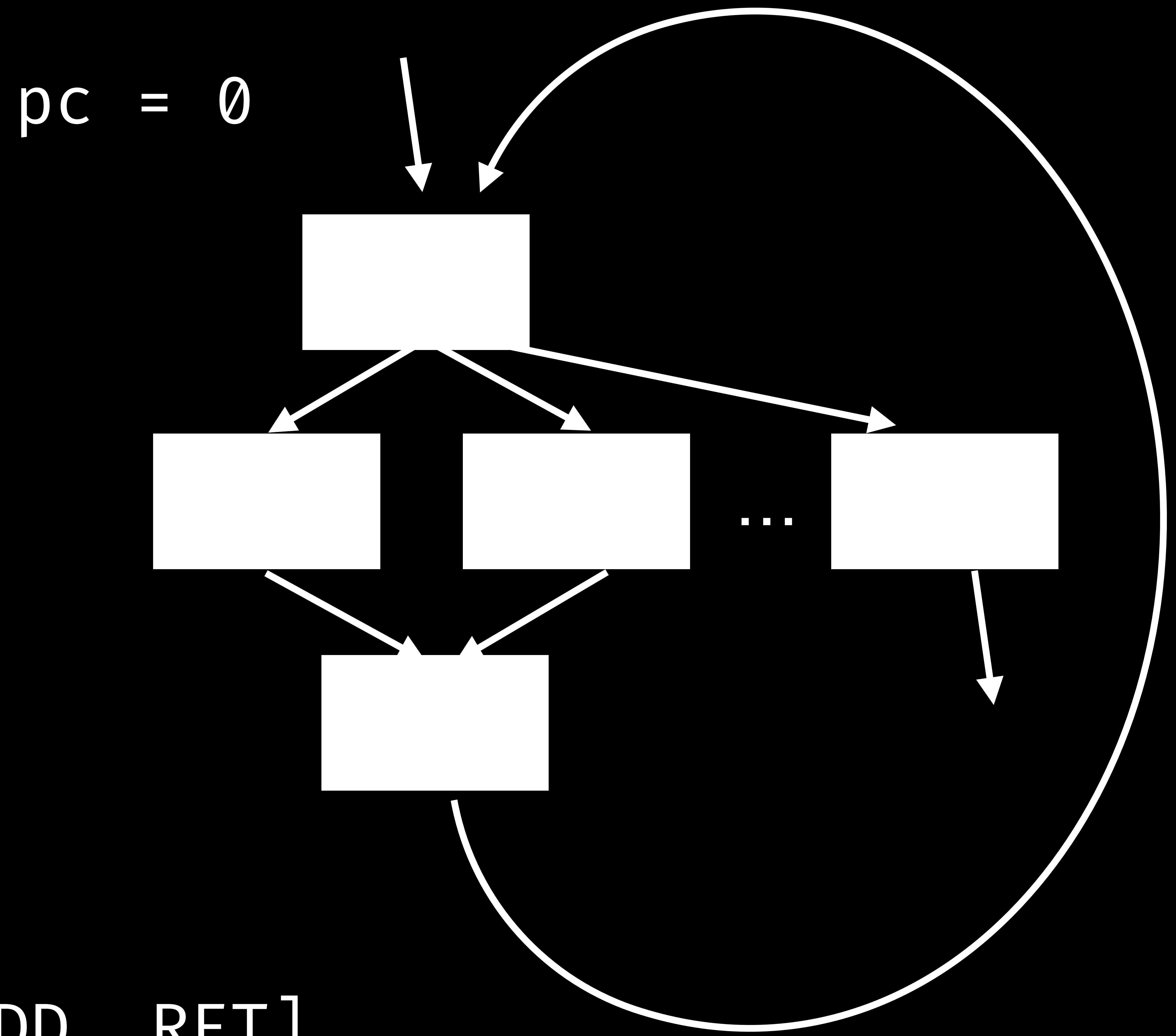


`code = [ADD, RET]`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

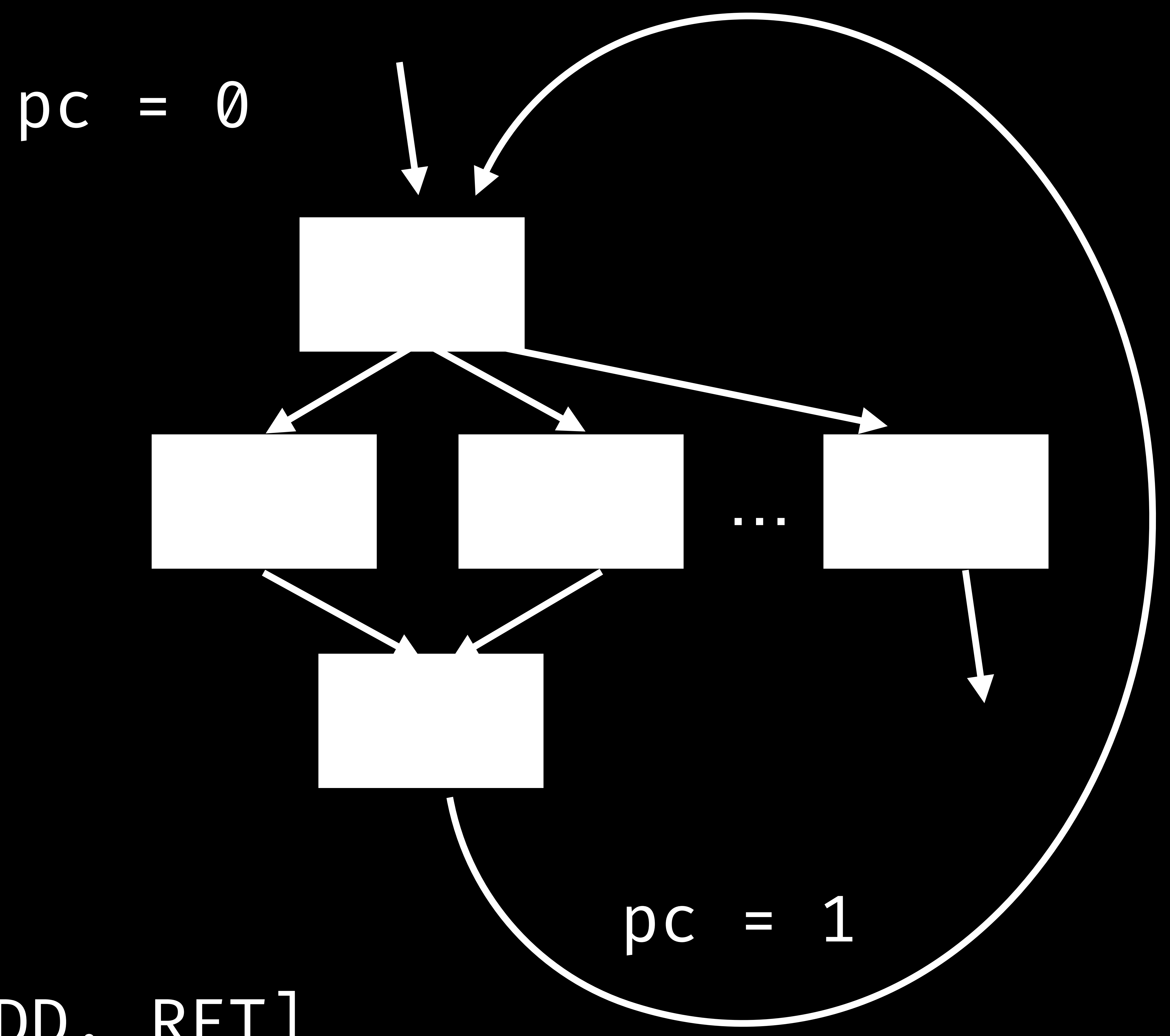


code = [ADD, RET]

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

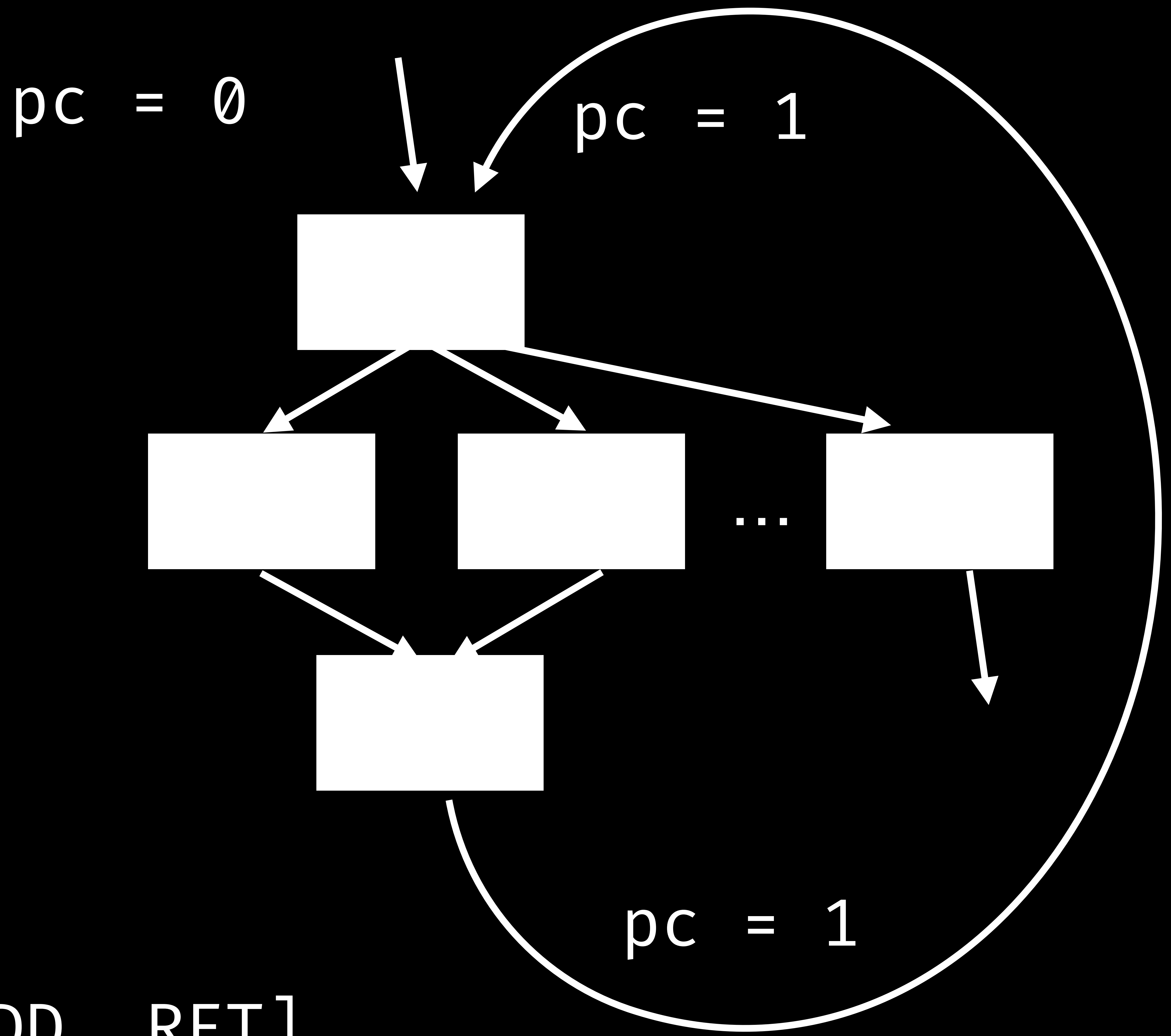


code = [ADD, RET]

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

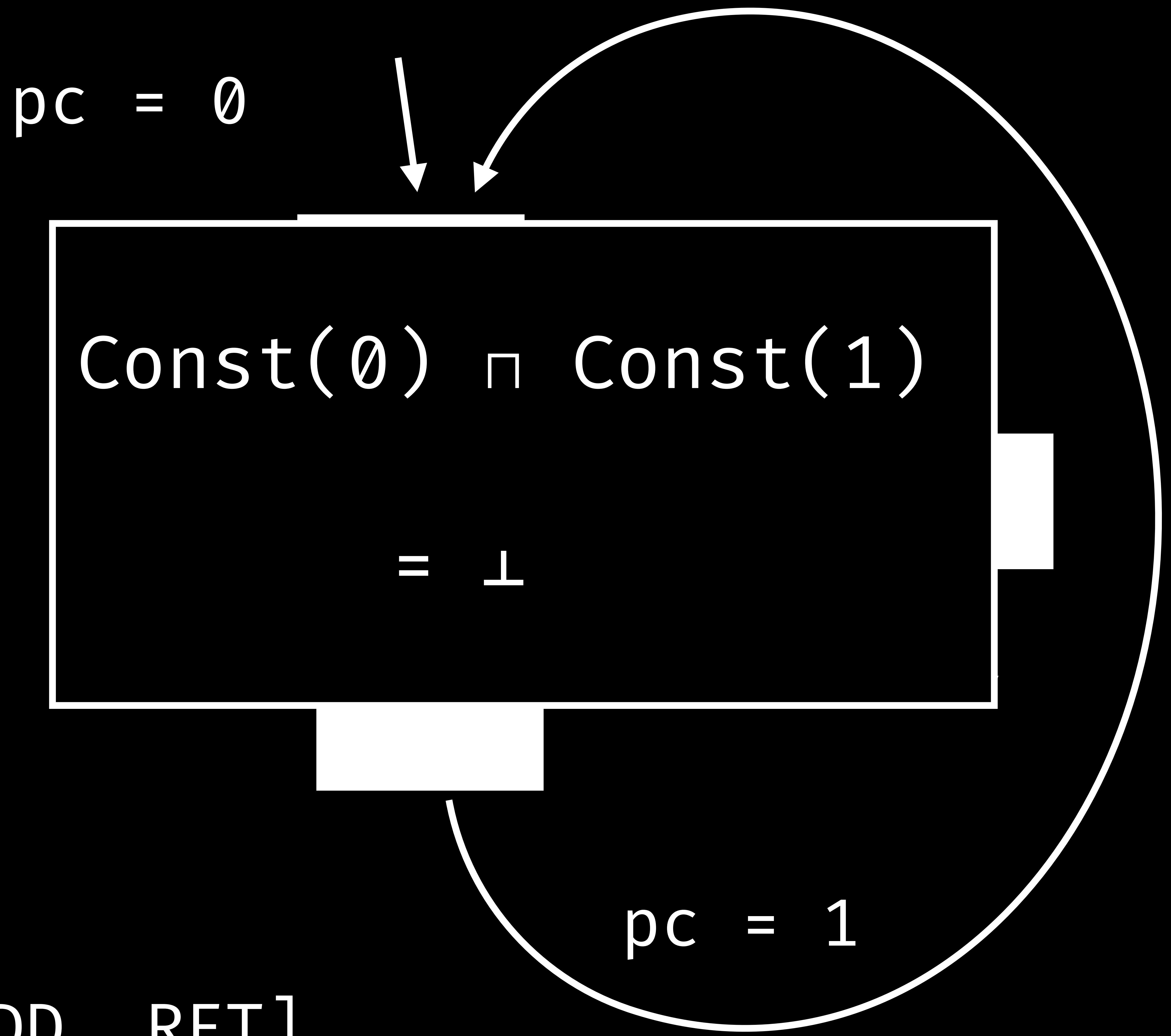


code = [ADD, RET]

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```



code = [ADD, RET]

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

`pc` is “not constant” when *meeting over all paths* for a static summary of the program point

```
code = [ADD, RET]
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

`pc` is “not constant” when *meeting over all paths* for a static summary of the program point

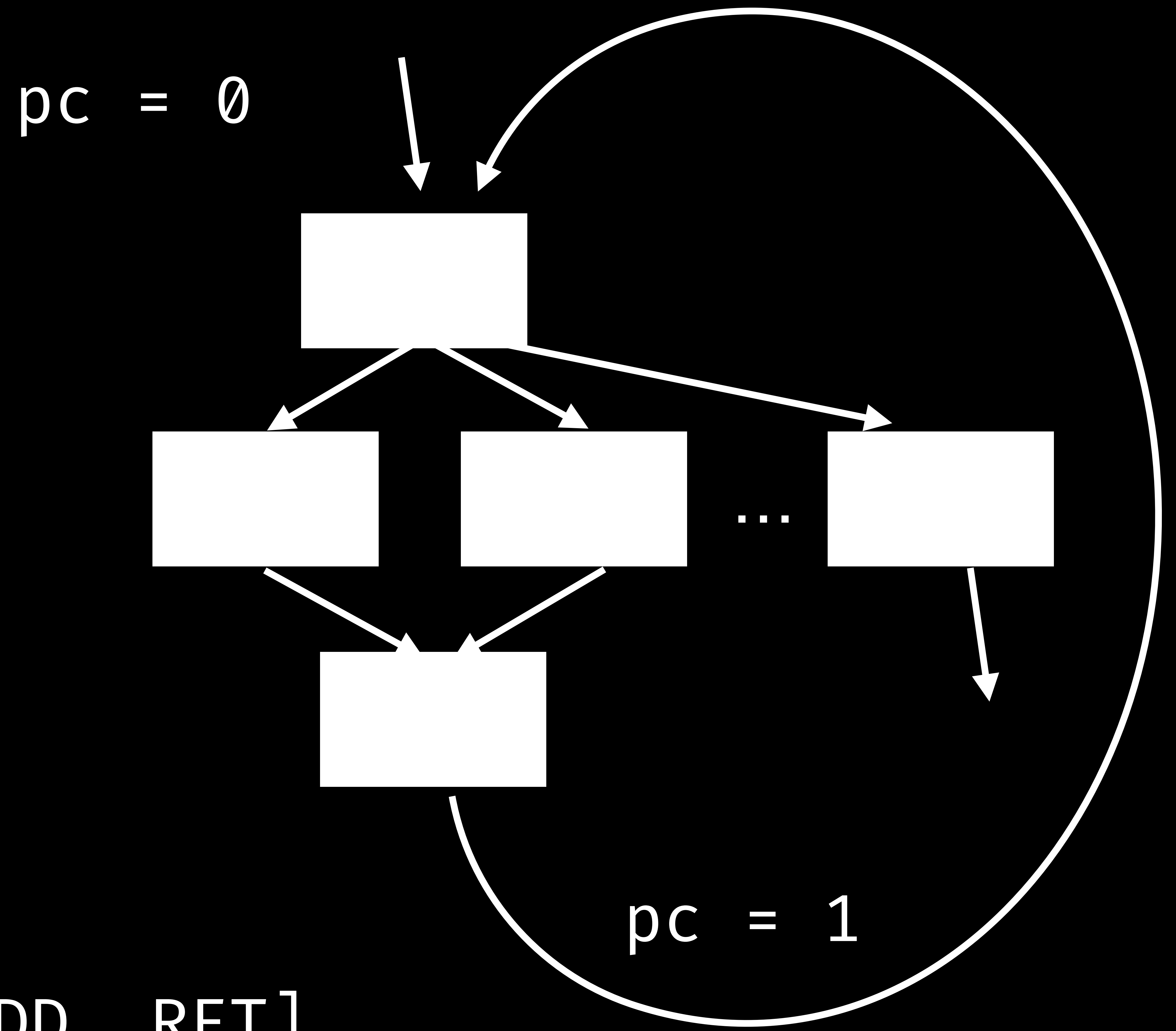
→ cannot constant-propagate or branch-fold on bytecode

```
code = [ADD, RET]
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```



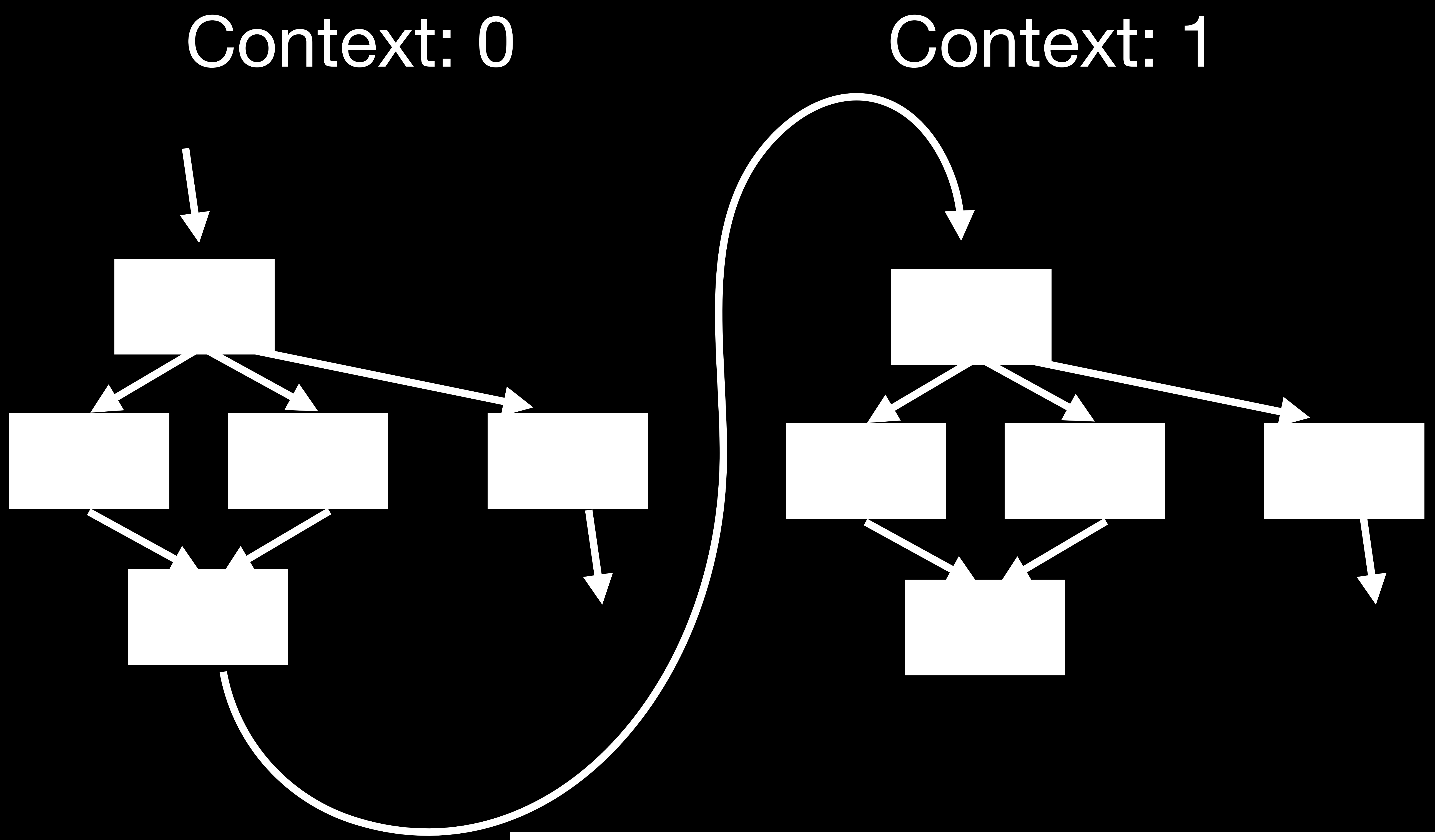

```
code = [ADD, RET]
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

```
code = [ADD, RET]
```

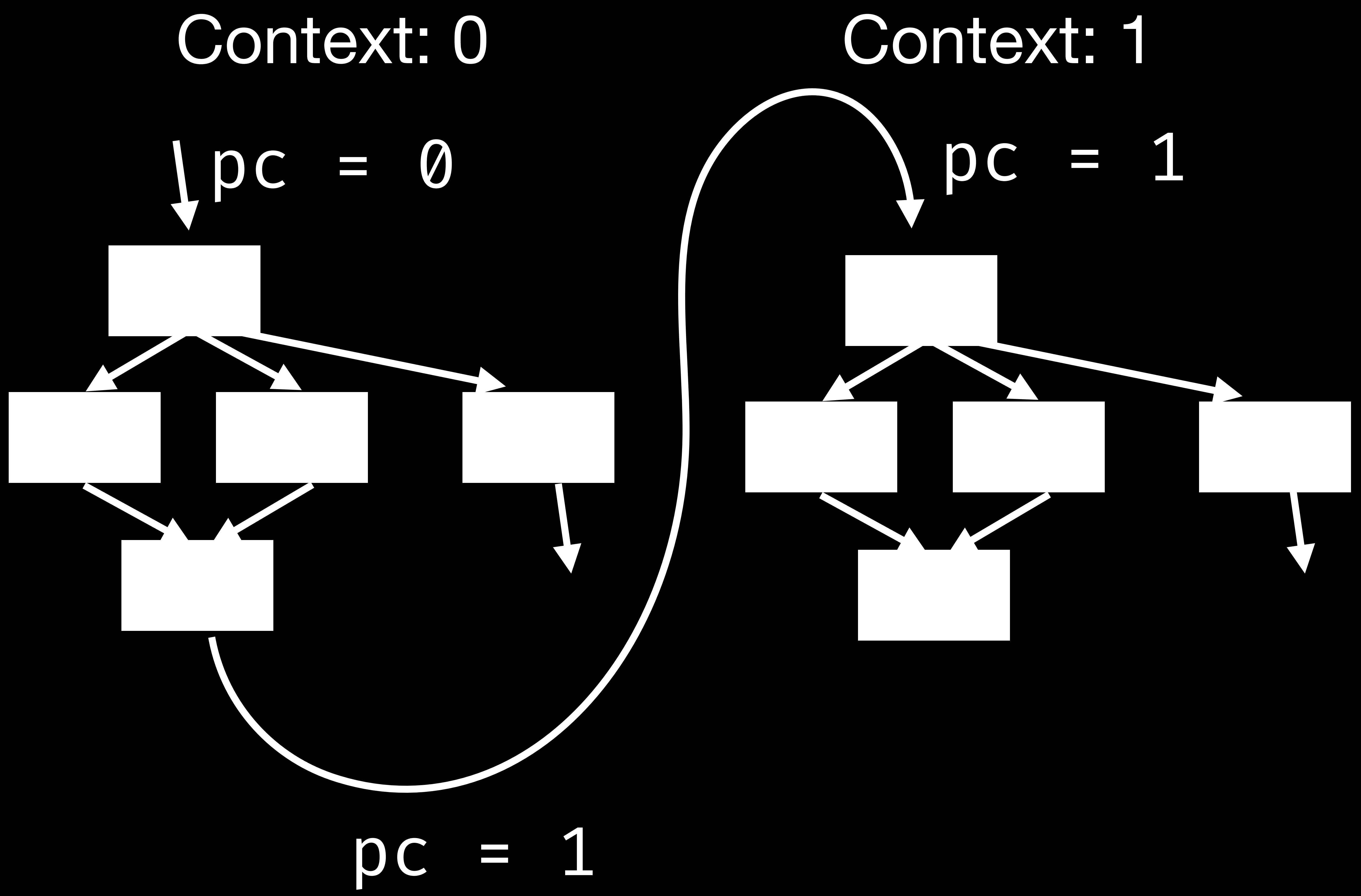


separate contexts to prevent const-prop from collapsing

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```

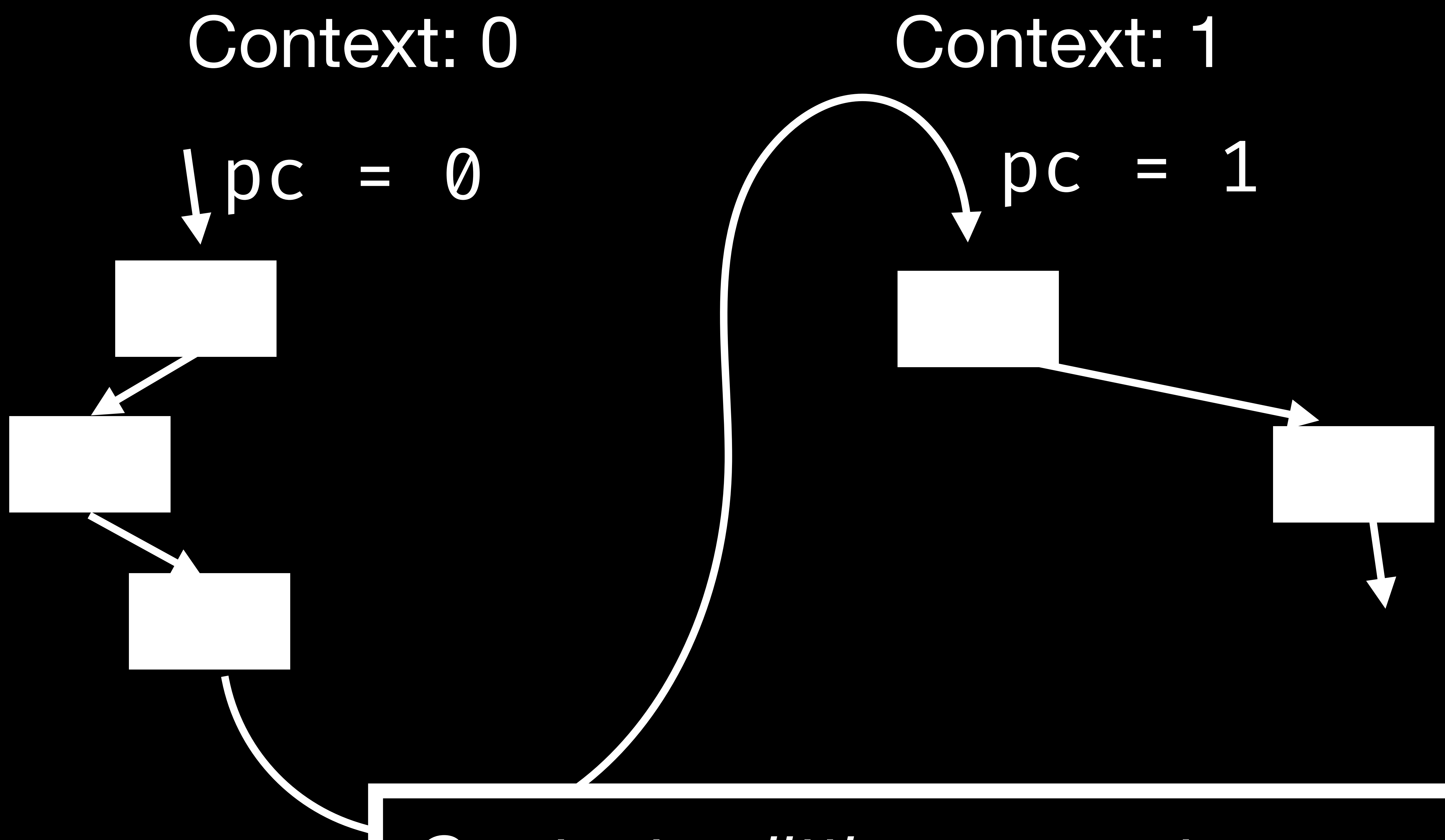


`code = [ADD, RET]`

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```



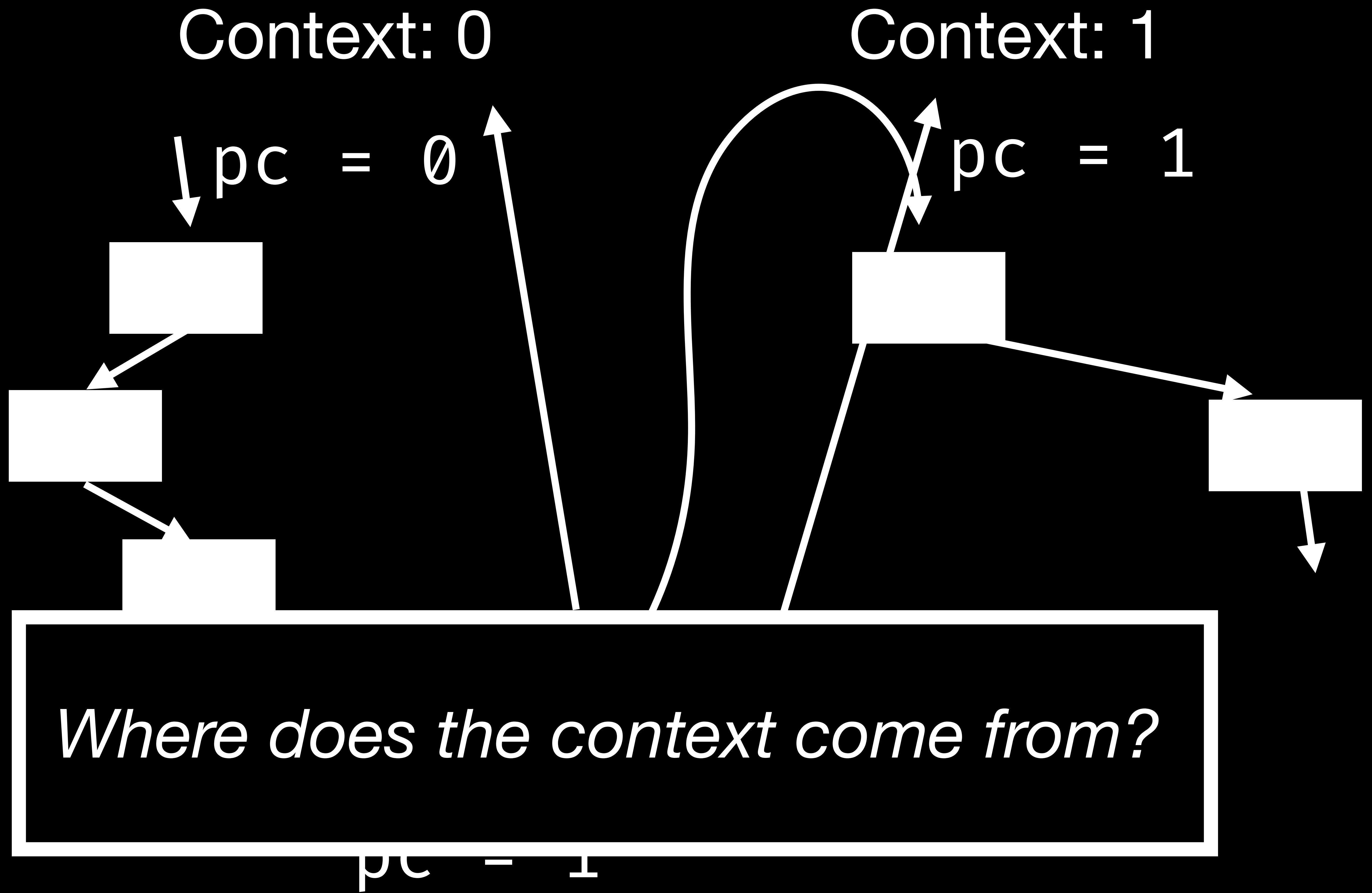
Context splitting + const-prop + branch folding is equivalent to a template compiler for free!

```
code = [ADD, RET]
```

A Practical First Futamura Projection…

… of an existing bytecode interpreter loop?

```
while (true) {
  switch(code[pc]) {
    case ADD:
      auto a = pop();
      auto b = pop();
      push(a + b);
      pc++;
      break;
    case RET:
      return pop();
  }
}
```




```
code = [ADD, RET]
```

Specialization Intrinsics

```
void interp(bytecode* pc) {

  while (true) {
    switch (*pc++) {
      case OP1:
        …

        break;
      case OP2:
        …

        break;
    }
  }
}
```

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        weval::update_context(pc);
        break;
      case OP2:
        …
        weval::update_context(pc);
        break;
    }
  }
}
```

Specialization Intrinsics

```
void interp(bytecode* pc) {

  while (true) {
    switch (*pc++) {
      case OP1:
        …

        break;
      case OP2:
        …

        break;
    }
  }
}
```

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        weval::update_context(pc);
        break;
      case OP2:

              date_context(pc);


    }
  }
}
```

Insight: intrinsics can drive context during abstract interpretation!

Specialization Intrinsics

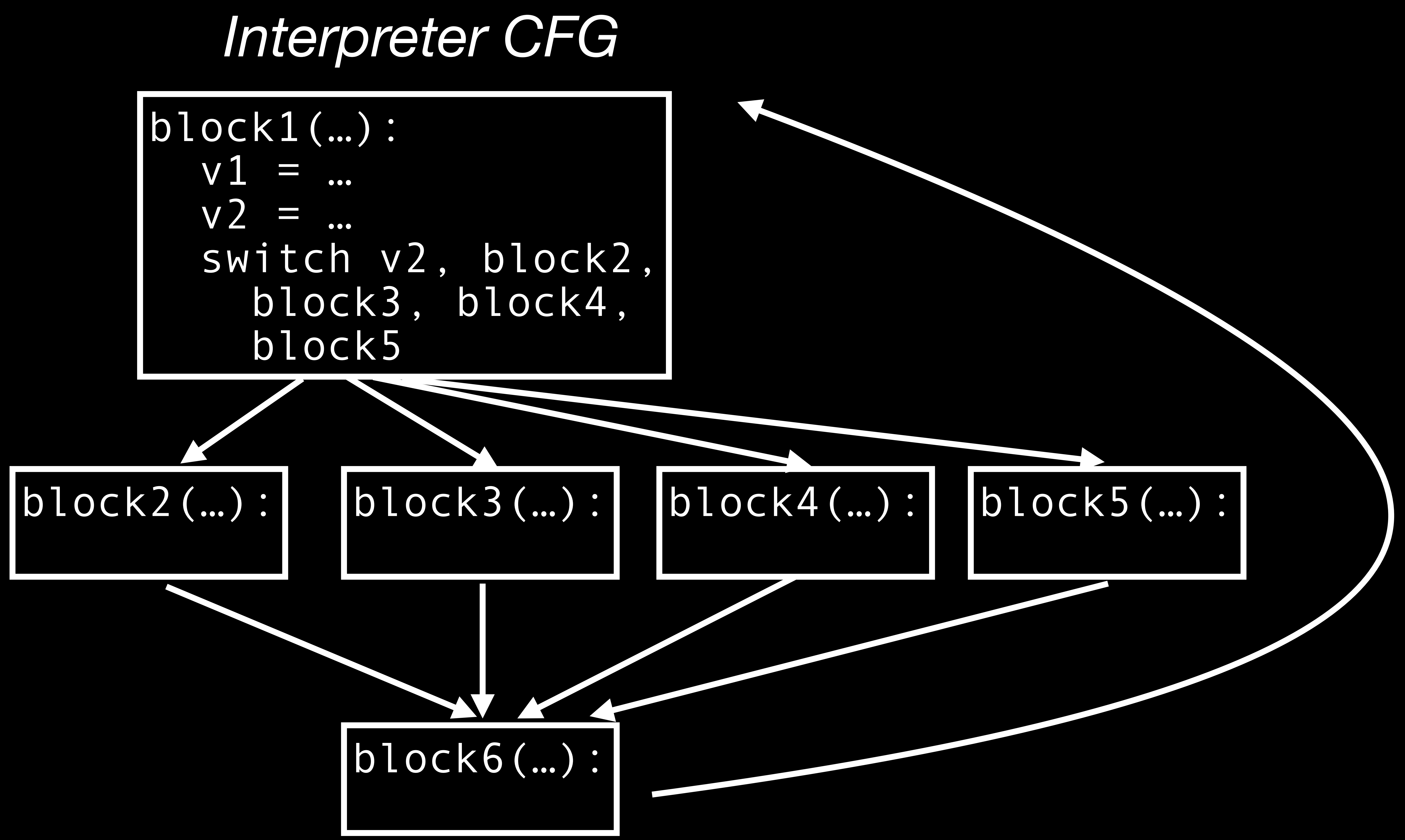
```
void interp(bytecode* pc) {

  while (true) {
    switch (*pc++) {
      case OP1:
        …

        break;
      case OP2:
        …

        break;
    }
  }
}
```

```
void interp(bytecode* pc) {
  weval::push_context(pc);
  while (true) {
    switch (*pc++) {
      case OP1:
        …
        weval::update_context(pc);
        break;
      case OP2:
        …
        date_context(pc);
```

Insight: intrinsics can drive context during abstract interpretation!

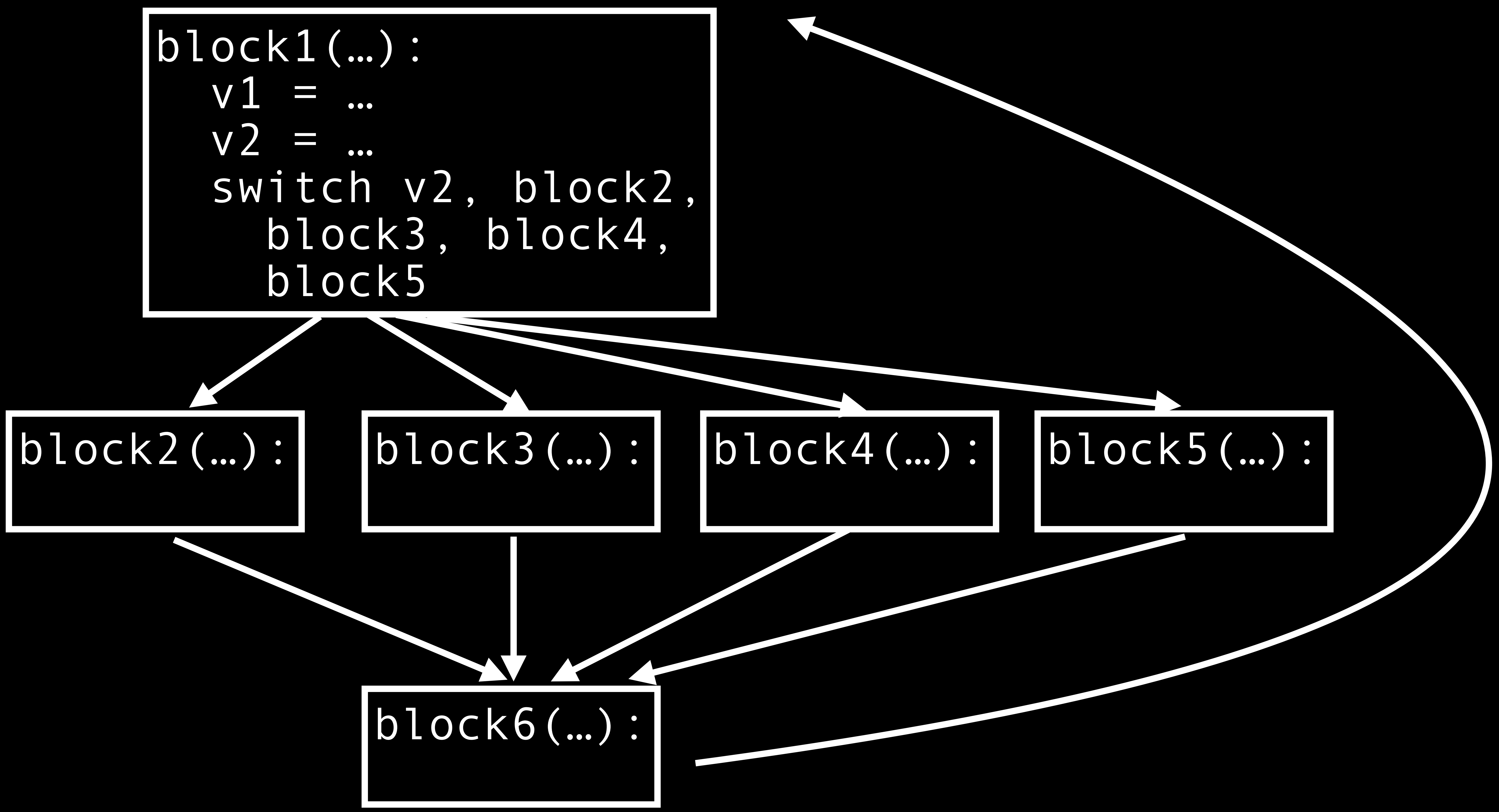
(not load-bearing for correctness, only analysis precision!)

The weval Transform



The weval Transform

Interpreter CFG

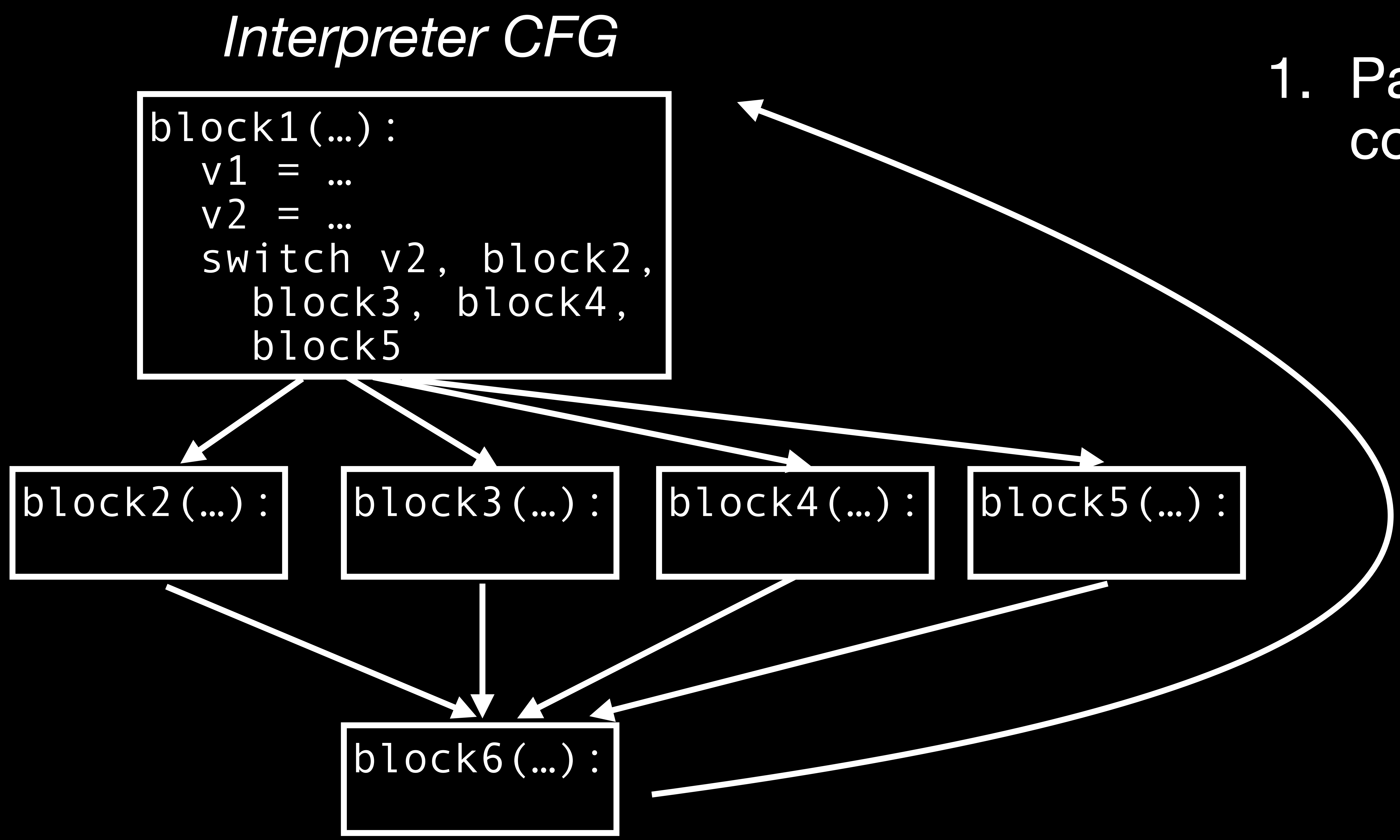


Generic

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Specialized

The weval Transform



1. Partially evaluate a block using constant propagation

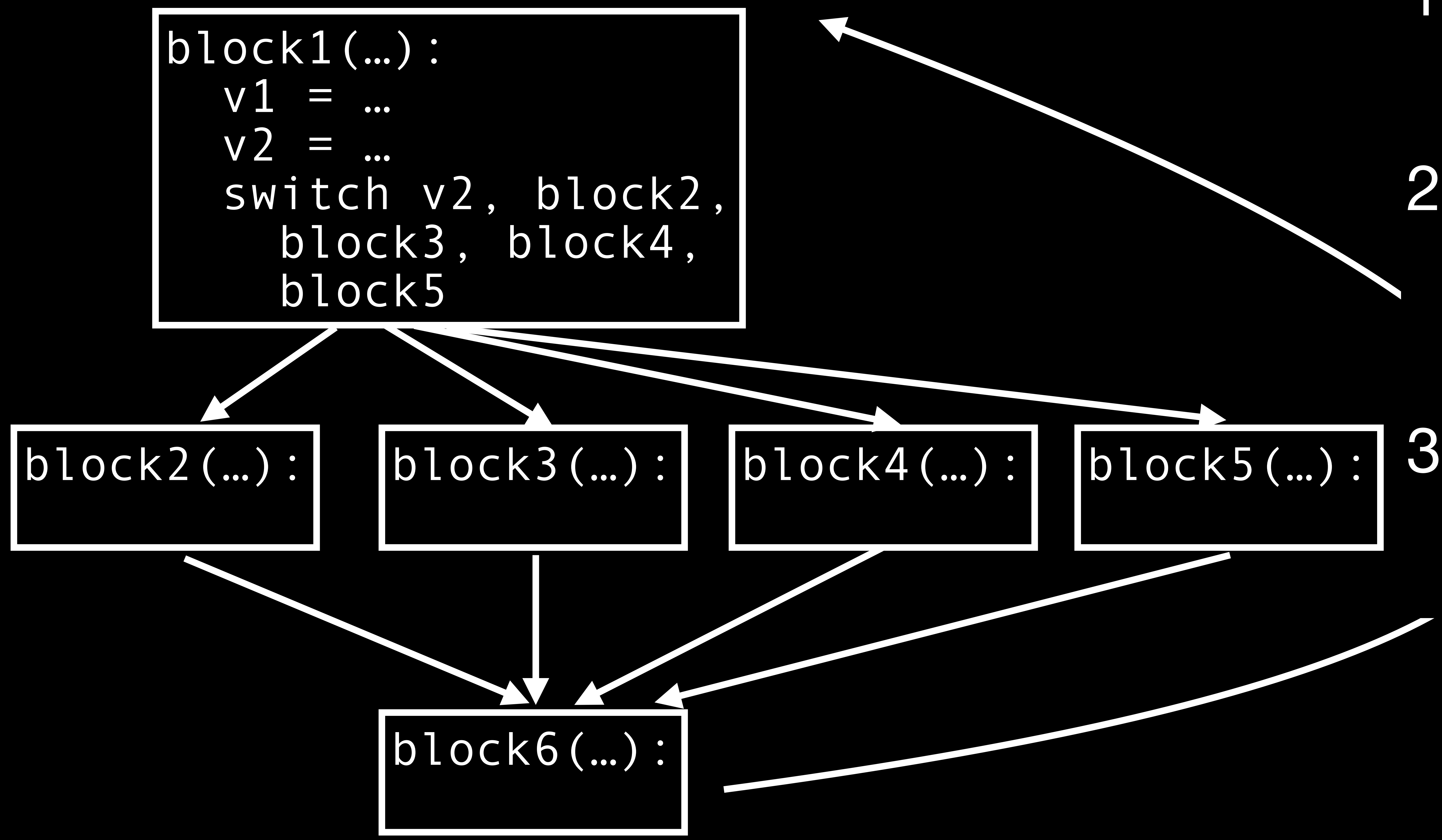
blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

Specialized

The weval Transform

Interpreter CFG



1. Partially evaluate a block using constant propagation
2. Track *context* as part of flow-sensitive state; update at intrinsics
3. At branches, enqueue targets

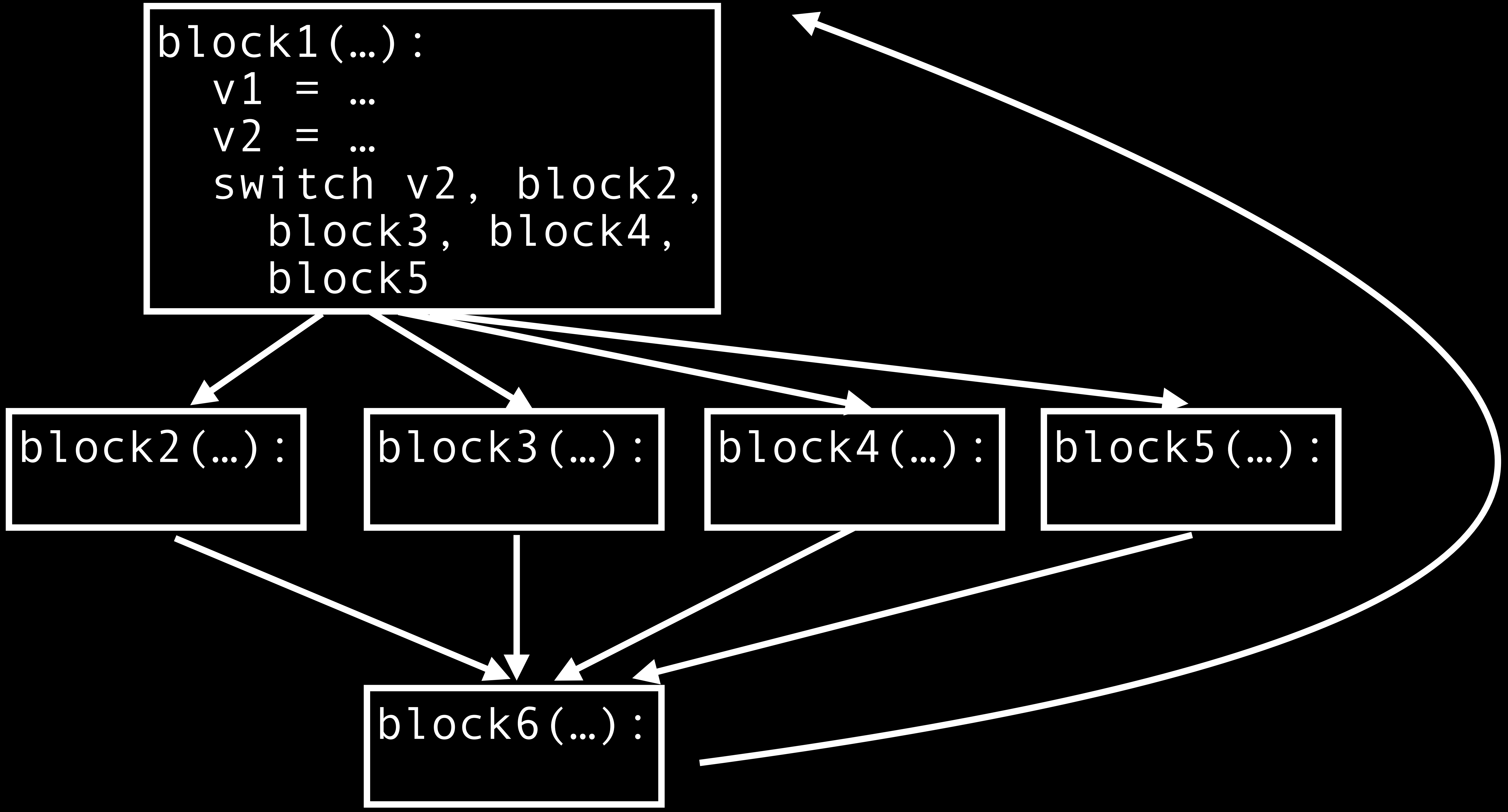
blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

Specialized

The weval Transform

Interpreter CFG



Generic

Context: PC 0

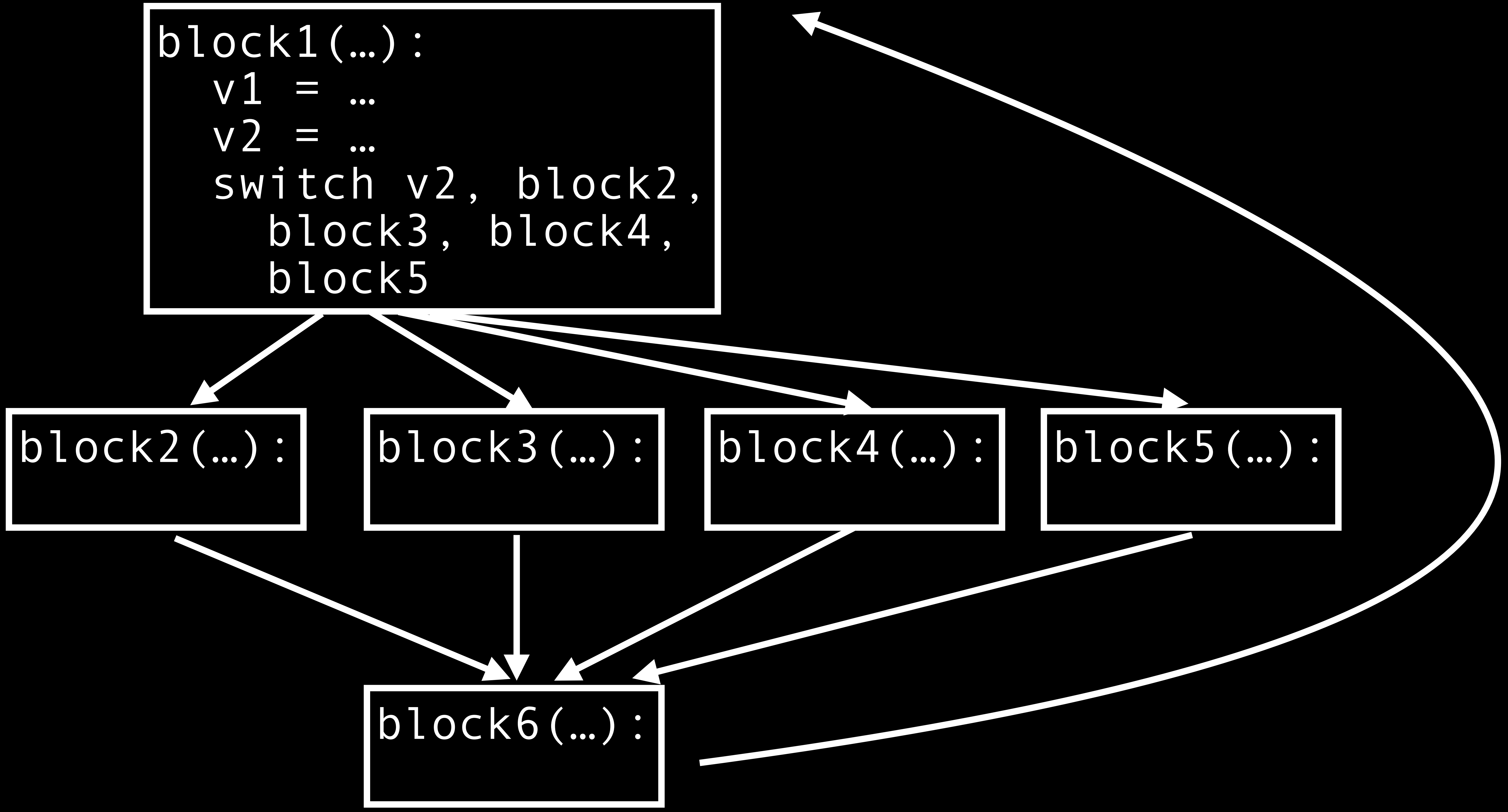
```
block1(…):
  v1 = …
  v2 = (load opcode)
  switch v2, block2,
    block3, block4,
    block5
```

Specialized

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

The weval Transform

Interpreter CFG



Generic

Context: PC 0

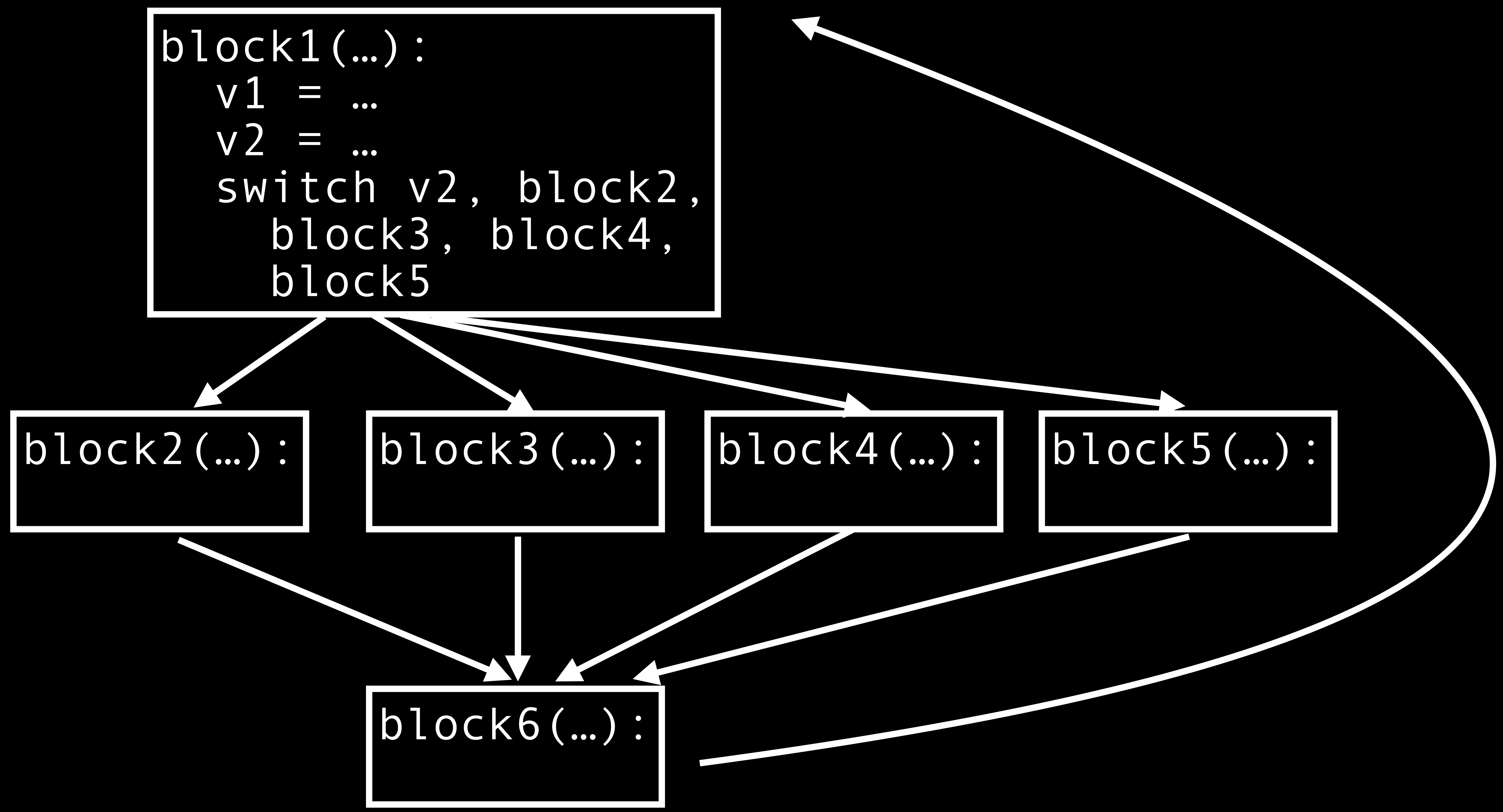
```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5
```

Specialized

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

The weval Transform

Interpreter CFG



Context: PC 0

```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5
```

```
block5(…):
```

```
block6(…):
  update_context(…)
```

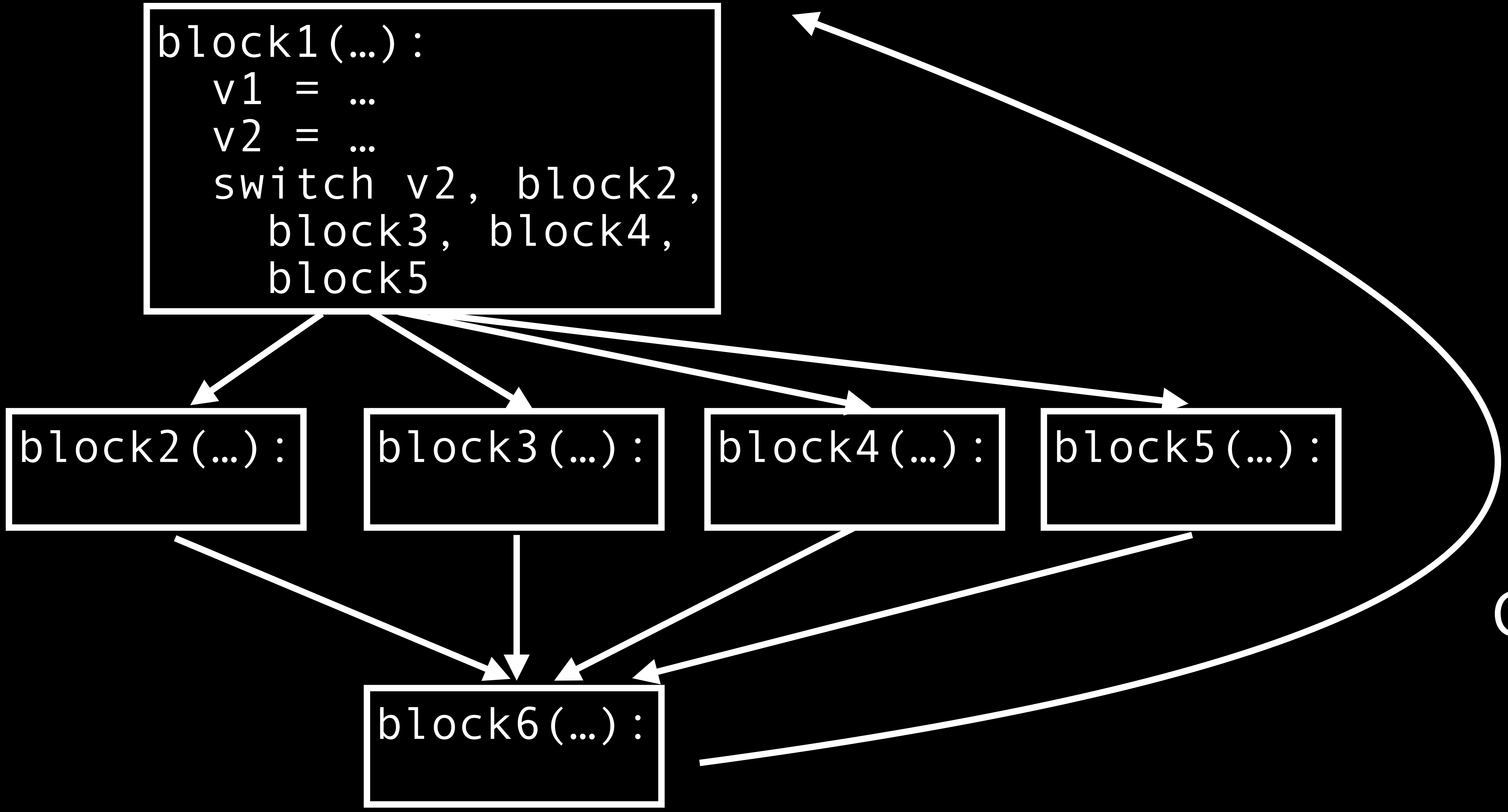
Generic

blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

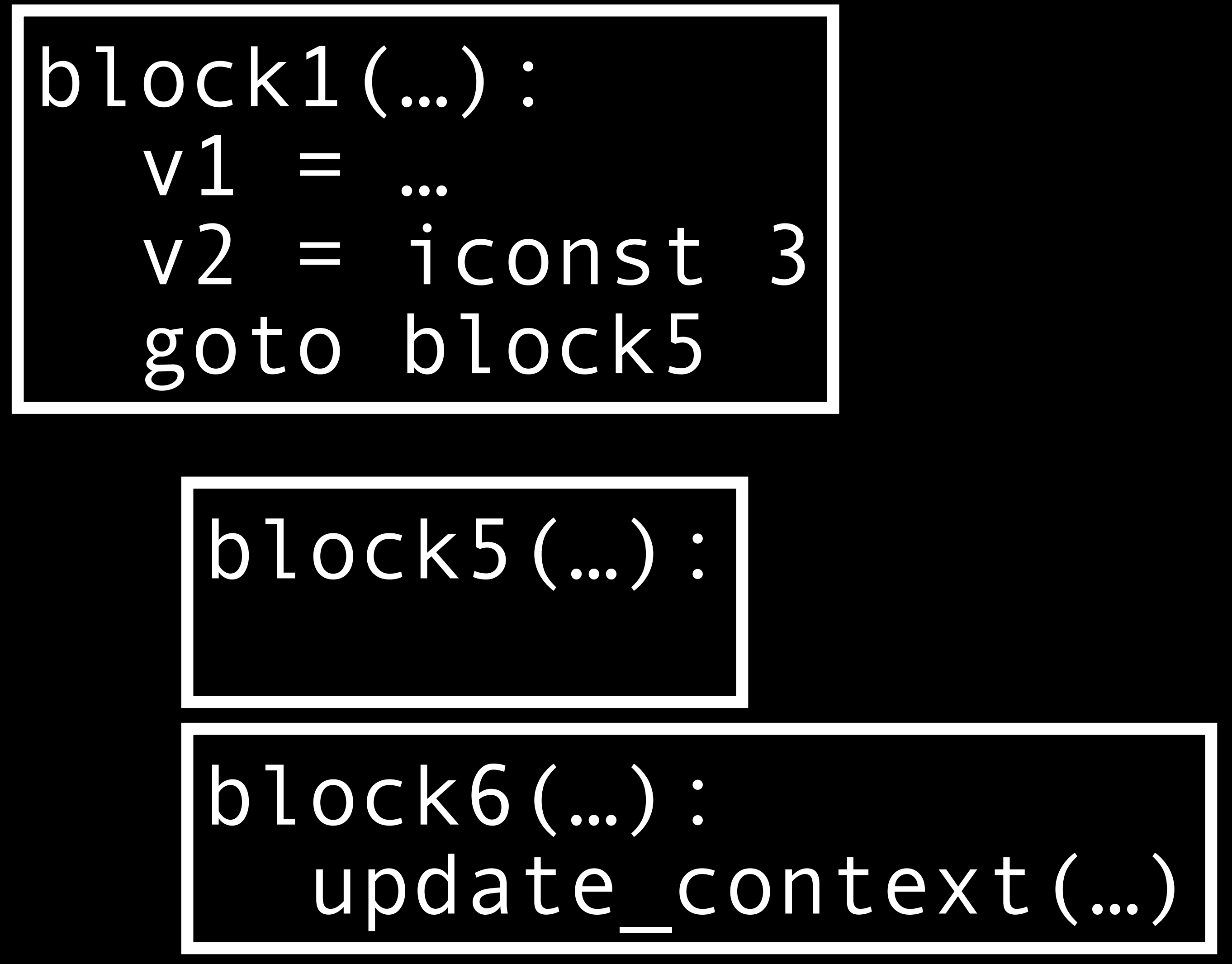
Specialized

The weval Transform

Interpreter CFG



Context: PC 0



Context: PC 1

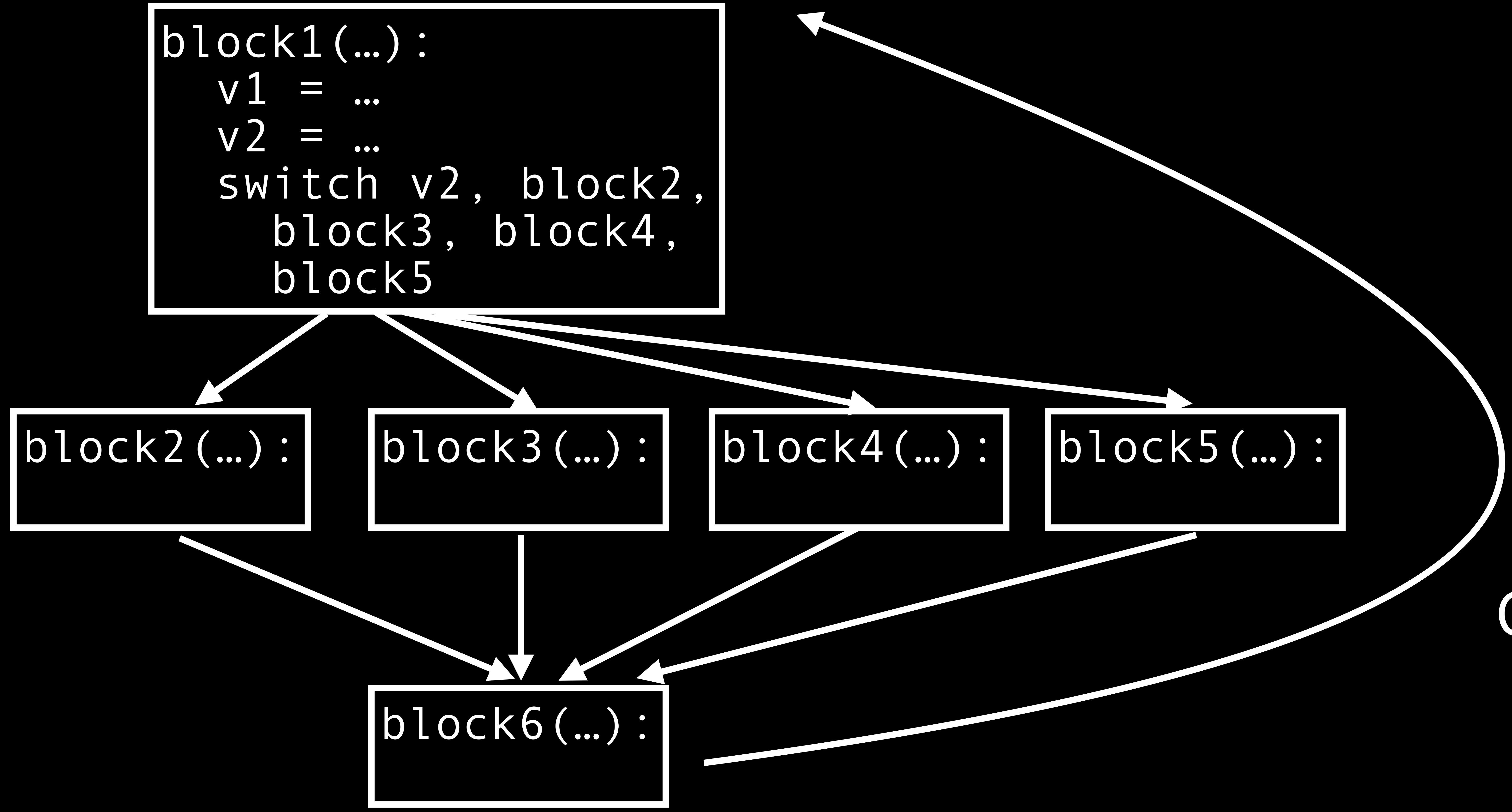
blocks: (Context, Block) -> Block
values: (Context, Value) -> Value
workqueue: (Context, Block)

Generic

Specialized

The weval Transform

Interpreter CFG



Generic

Context: PC 0

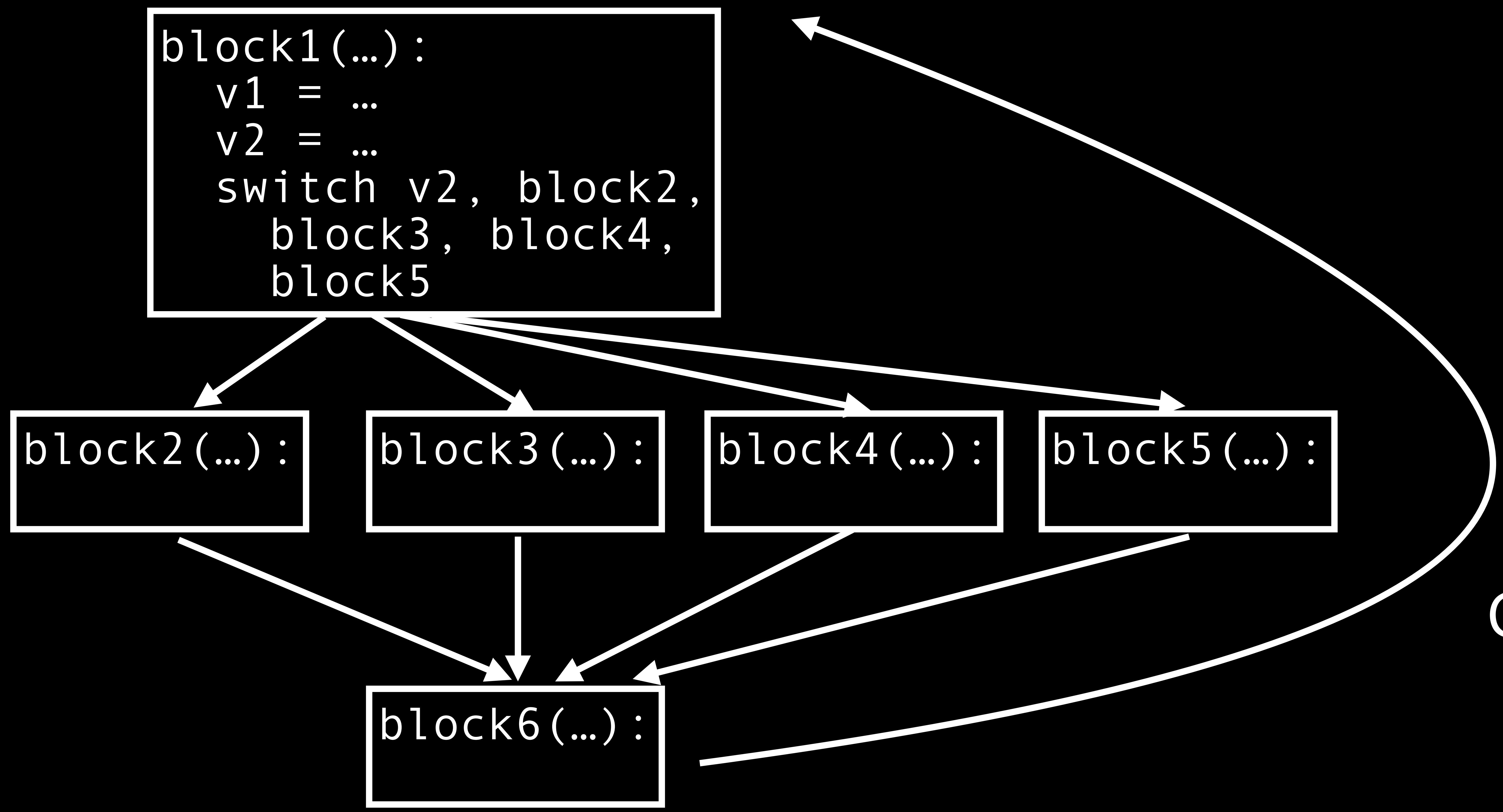
```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5

block5(…):

block6(…):
  update_context(…)
```

Context: PC 1

```
block1(…):
  …

block4(…):
  …

block6(…):
  update_context(…)
```

The weval Transform

Interpreter CFG



Generic

PC 0: ADD
PC 1: GOTO 0

Context: PC 0

```
block1(…):
  v1 = …
  v2 = iconst 3
  goto block5
```

```
block5(…):
```

```
block6(…):
  update_context(…)
```

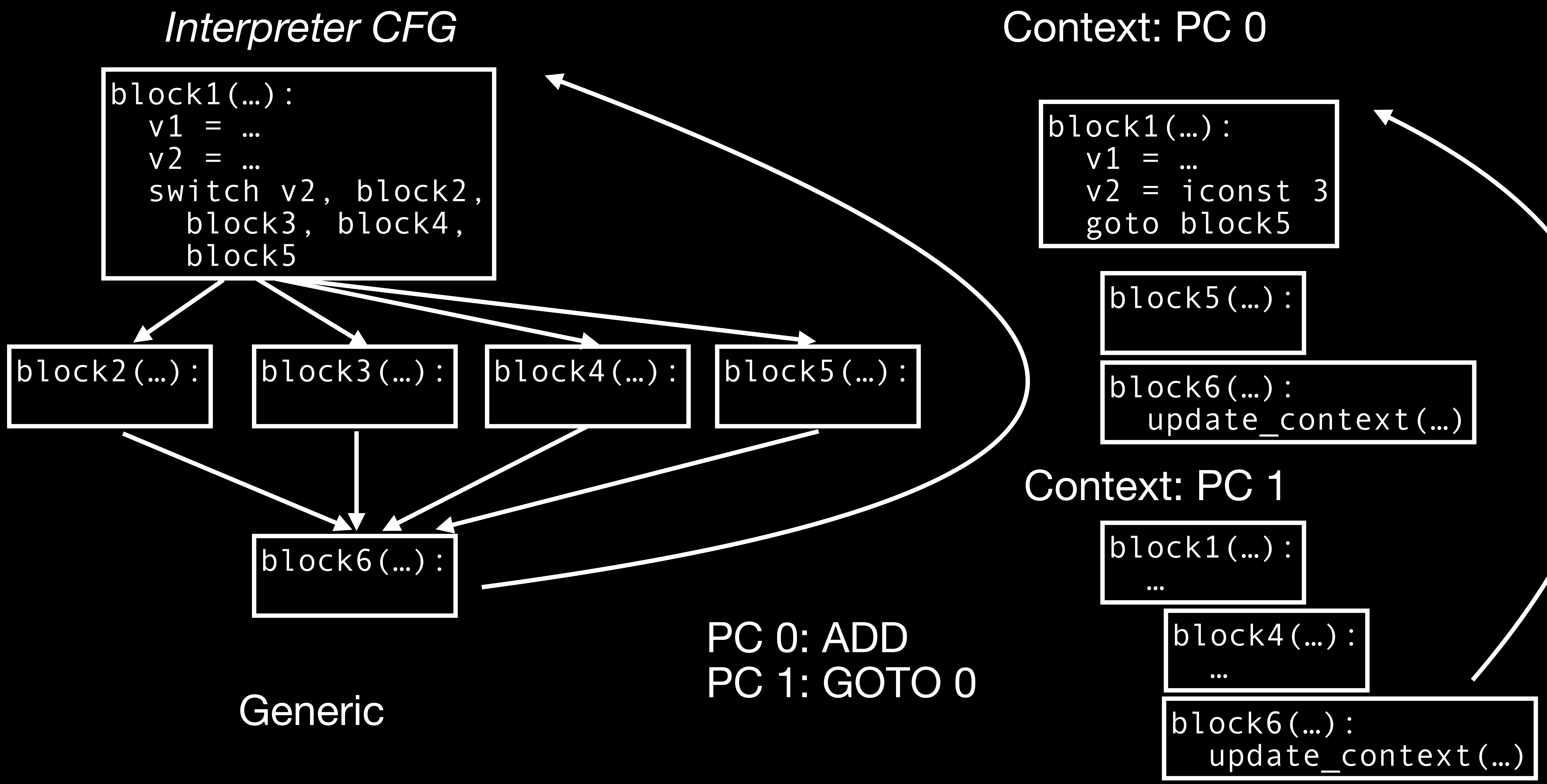
Context: PC 1

```
block1(…):
  …
```

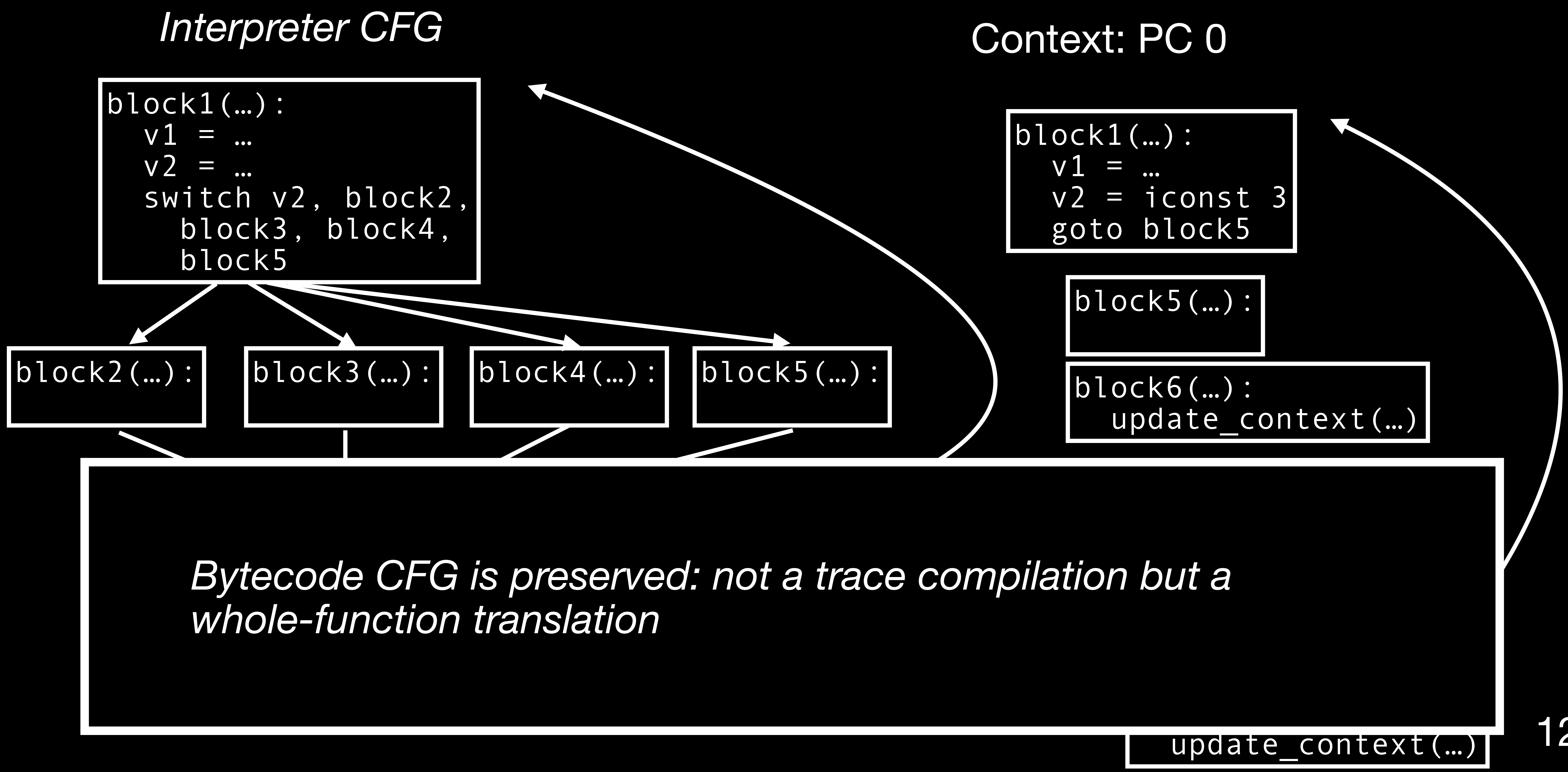
```
block4(…):
  …
```

```
block6(…):
  update_context(…)
```

The weval Transform



The weval Transform



Bytecode CFG is preserved: not a trace compilation but a whole-function translation

In the Paper

- Algorithm and other implementation details
- Intrinsics for optimizing interpreter state (lifting dynamic loads/stores to SSA)
- Maintaining SSA (static single assignment) form with phi/blockparam insertion
- Value-specialization for handling “switch” opcodes

Realistic Interpreter: SpiderMonkey

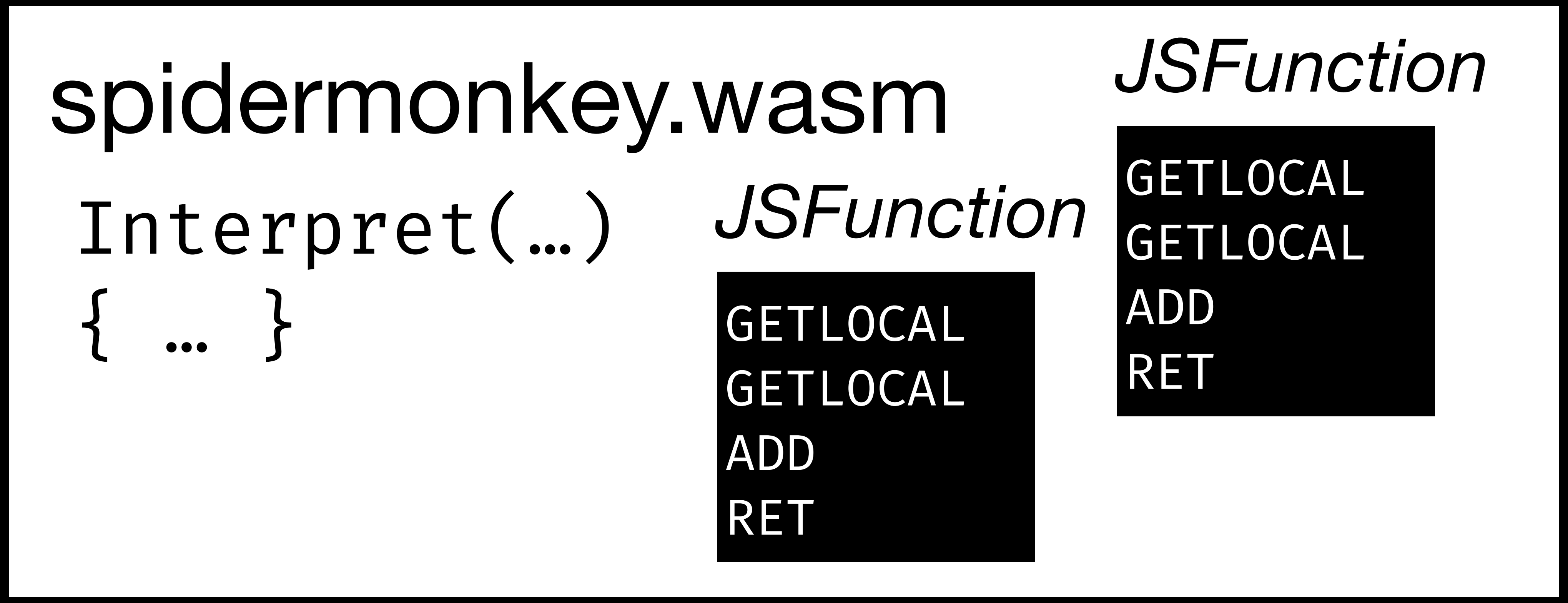
- SpiderMonkey (Firefox’s JavaScript engine) runs inside a Wasm module in *interpreter-only mode*. Useful for Function-as-a-Service and plugin scenarios.

Realistic Interpreter: SpiderMonkey

- SpiderMonkey (Firefox’s JavaScript engine) runs inside a Wasm module in *interpreter-only mode*. Useful for Function-as-a-Service and plugin scenarios.
- weval was developed to AOT-compile its JS bytecode to Wasm bytecode.

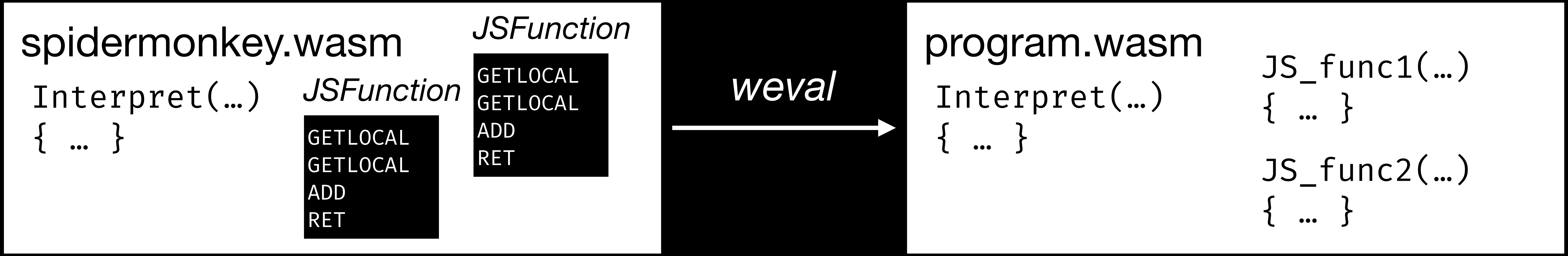
Realistic Interpreter: SpiderMonkey

- SpiderMonkey (Firefox's JavaScript engine) runs inside a Wasm module in *interpreter-only mode*. Useful for Function-as-a-Service and plugin scenarios.
- weval was developed to AOT-compile its JS bytecode to Wasm bytecode.



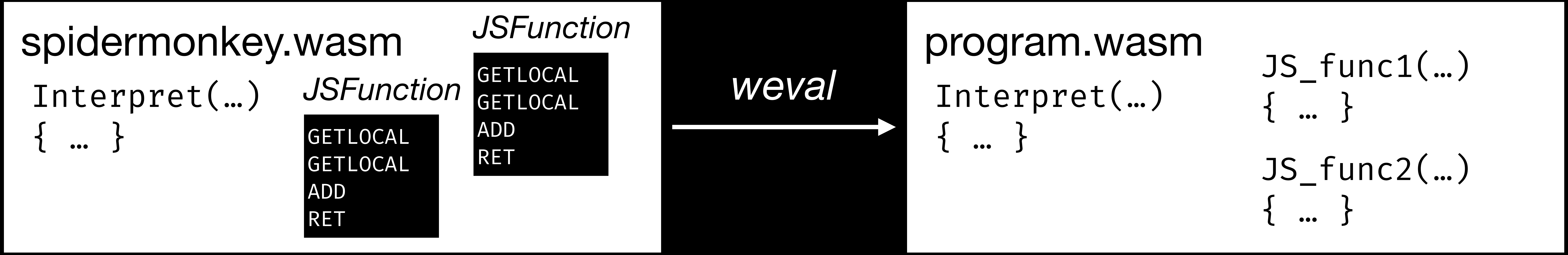
Realistic Interpreter: SpiderMonkey

- SpiderMonkey (Firefox's JavaScript engine) runs inside a Wasm module in *interpreter-only mode*. Useful for Function-as-a-Service and plugin scenarios.
- weval was developed to AOT-compile its JS bytecode to Wasm bytecode.



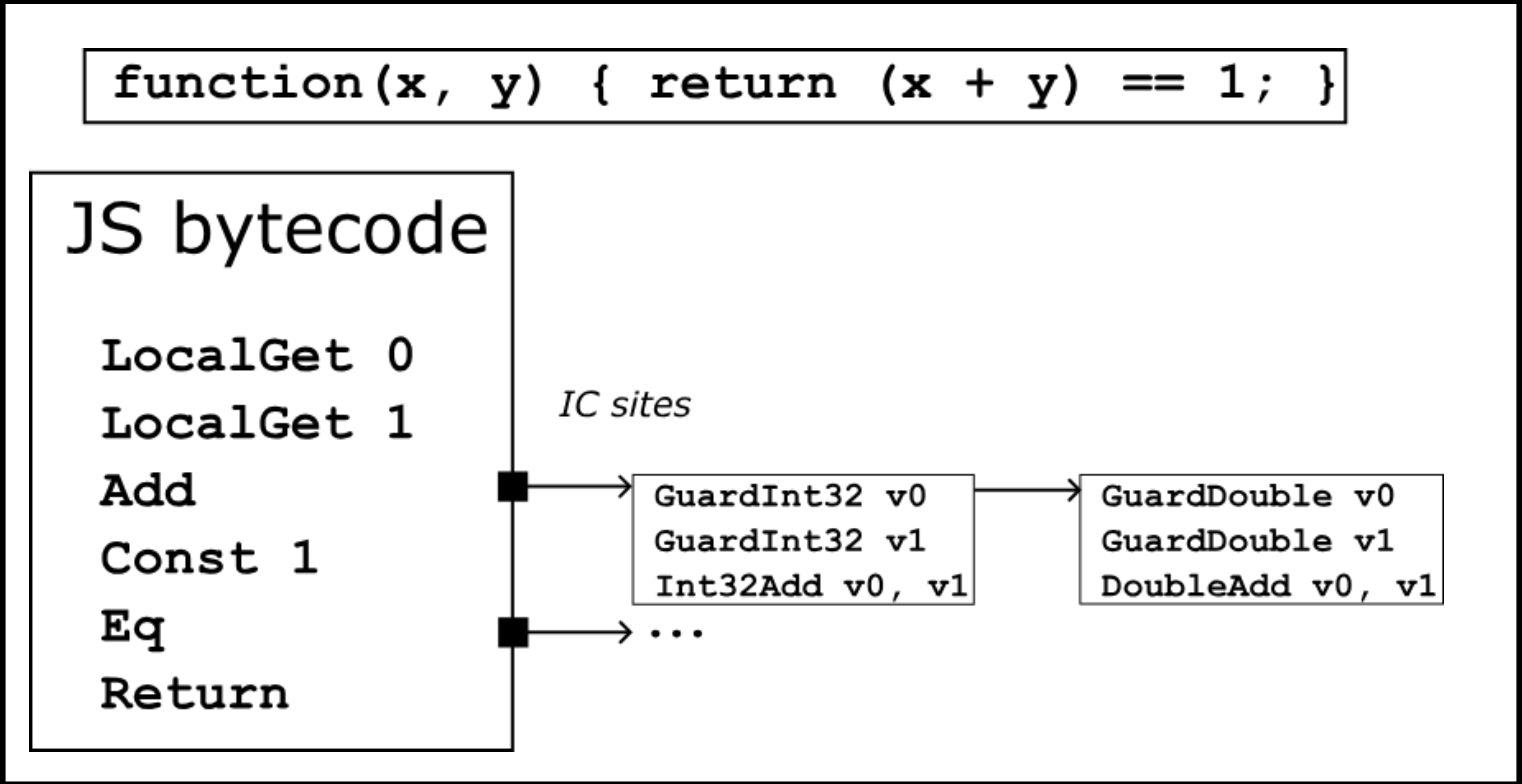
Realistic Interpreter: SpiderMonkey

- SpiderMonkey (Firefox's JavaScript engine) runs inside a Wasm module in *interpreter-only mode*. Useful for Function-as-a-Service and plugin scenarios.
- weval was developed to AOT-compile its JS bytecode to Wasm bytecode.

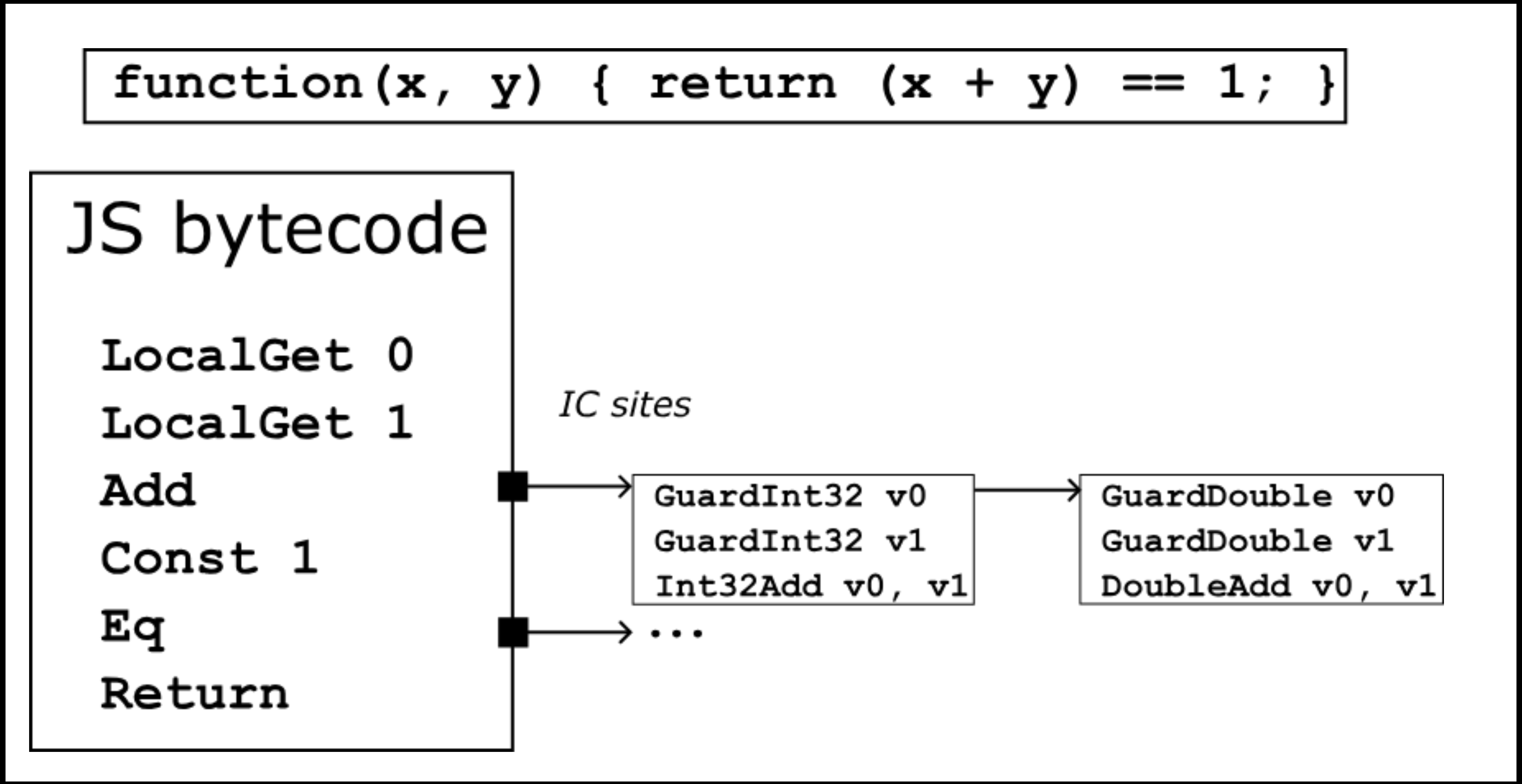


- How is AOT compilation of a dynamic language possible?

AOT Baseline Compilation

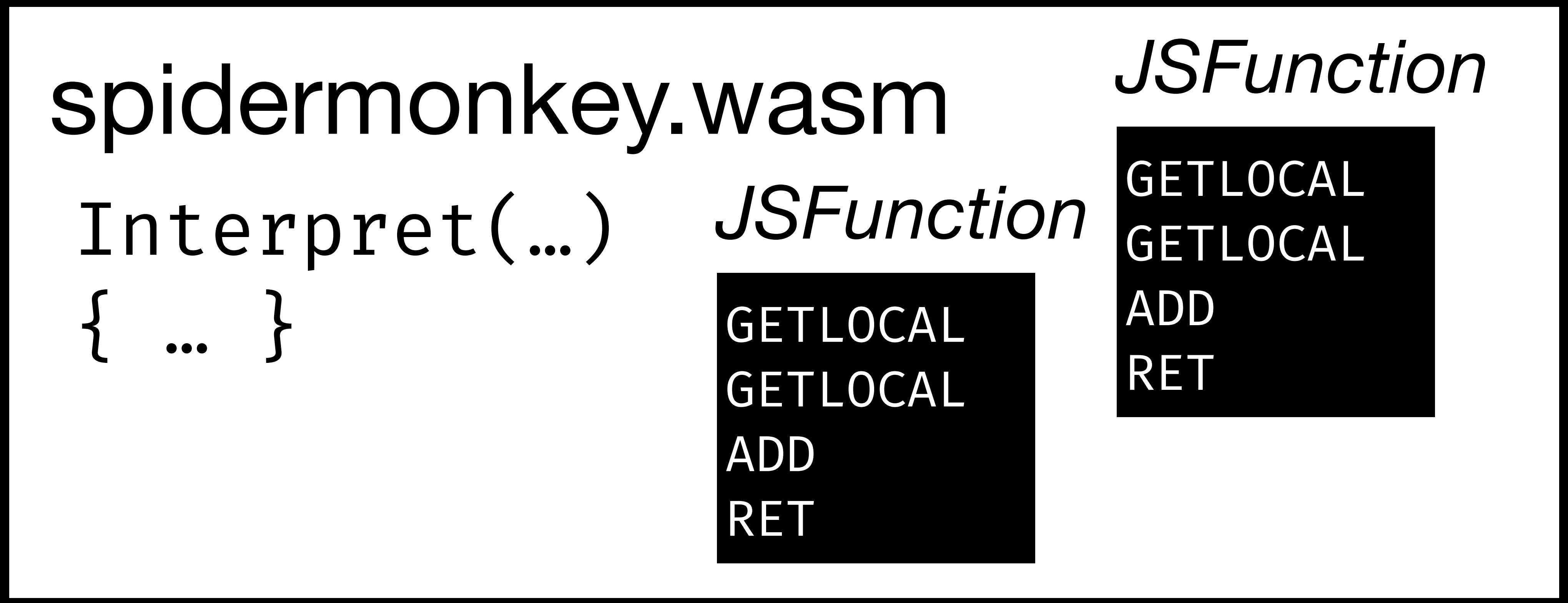


AOT Baseline Compilation

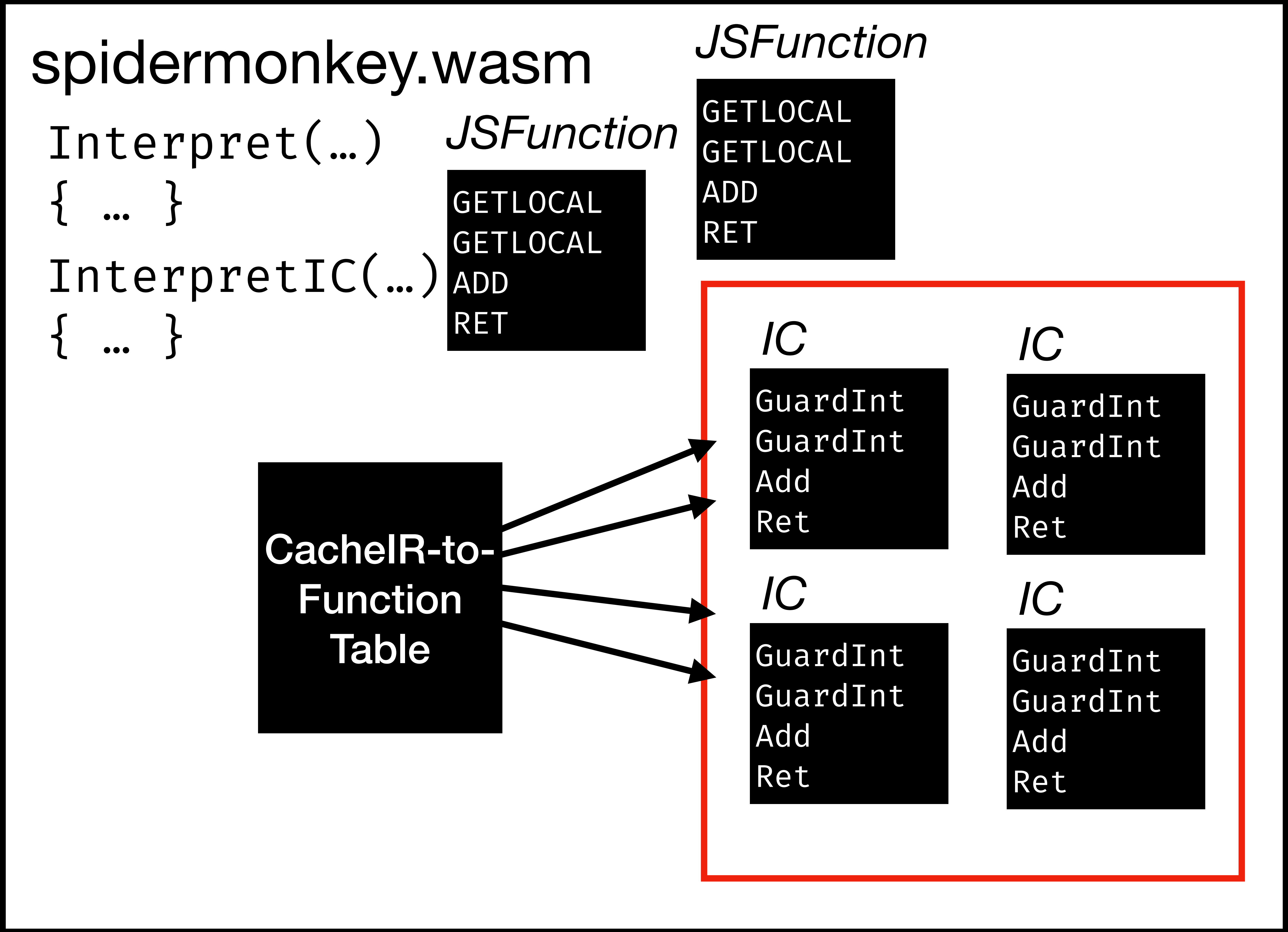


Key idea: *late binding* for execution semantics (dynamic types) becomes *late binding* in compilation strategy (indirect call via IC head)

AOT Baseline Compilation

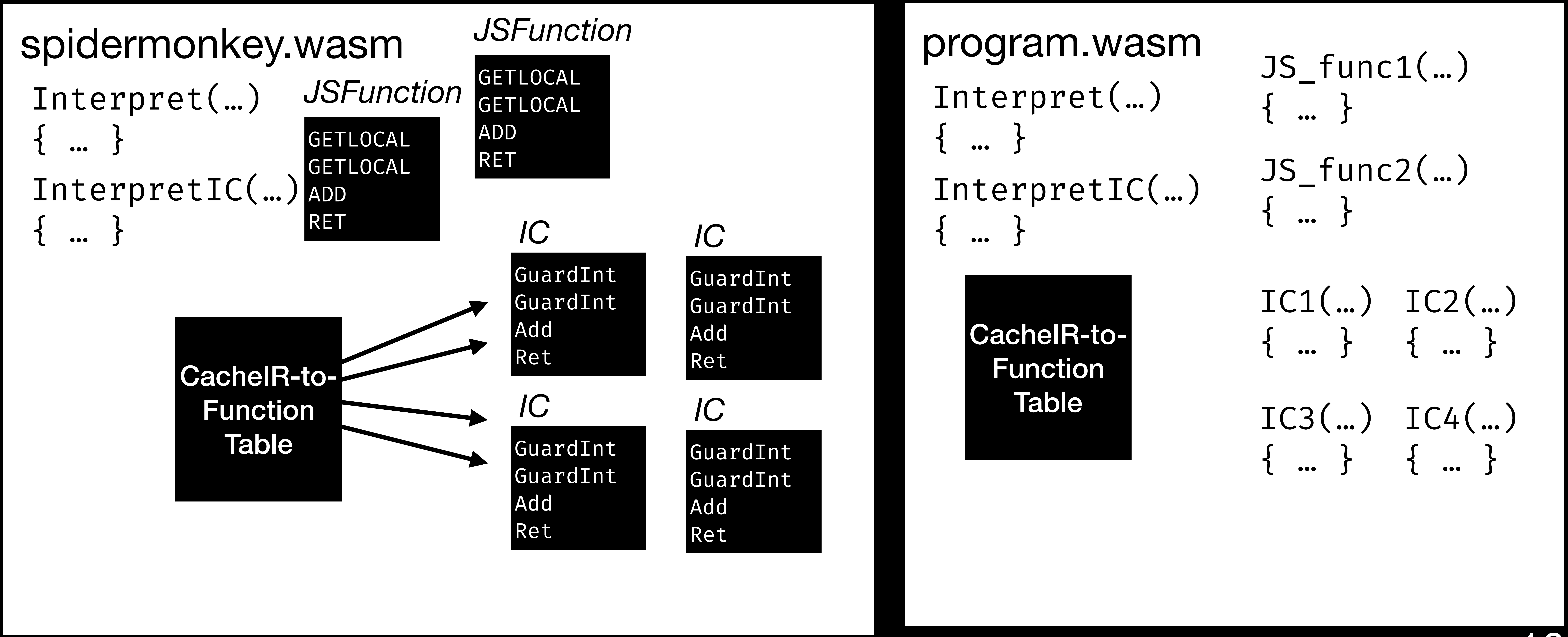


AOT Baseline Compilation

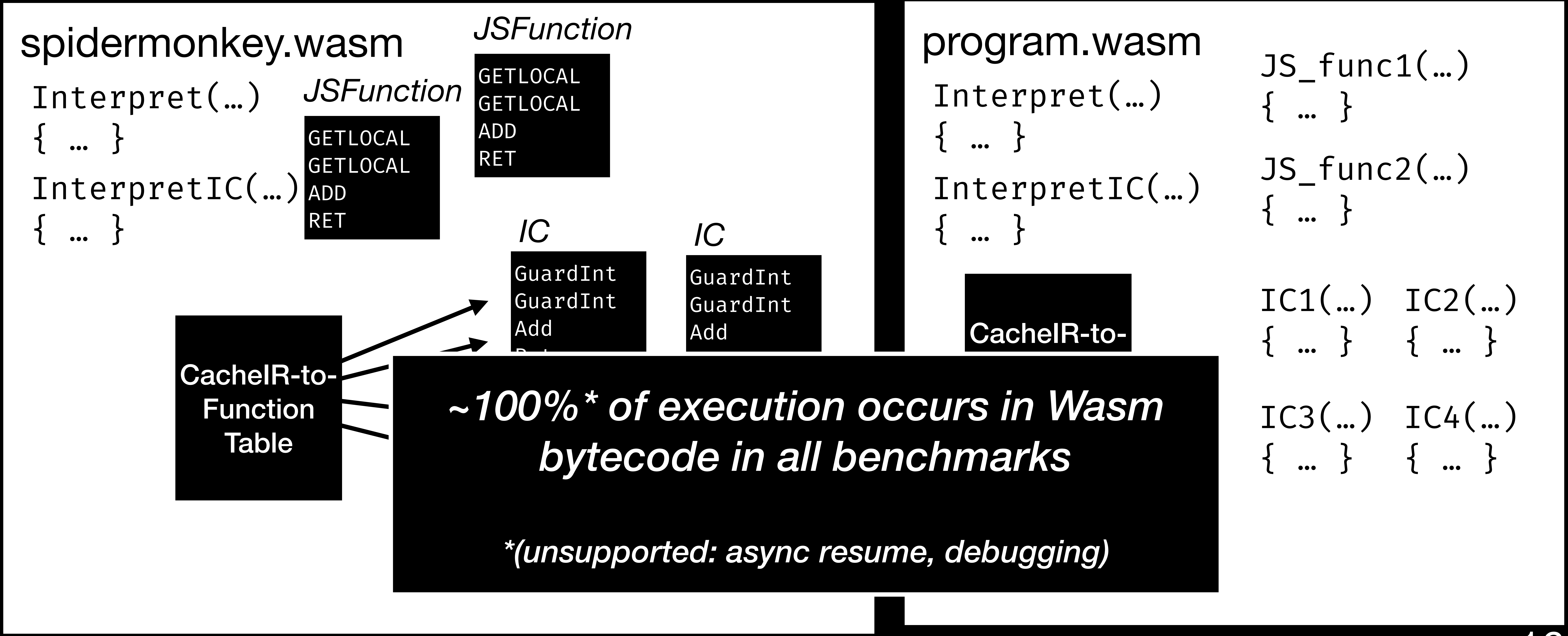


Pre-collected IC corpus (cover 100% of unit tests)

AOT Baseline Compilation



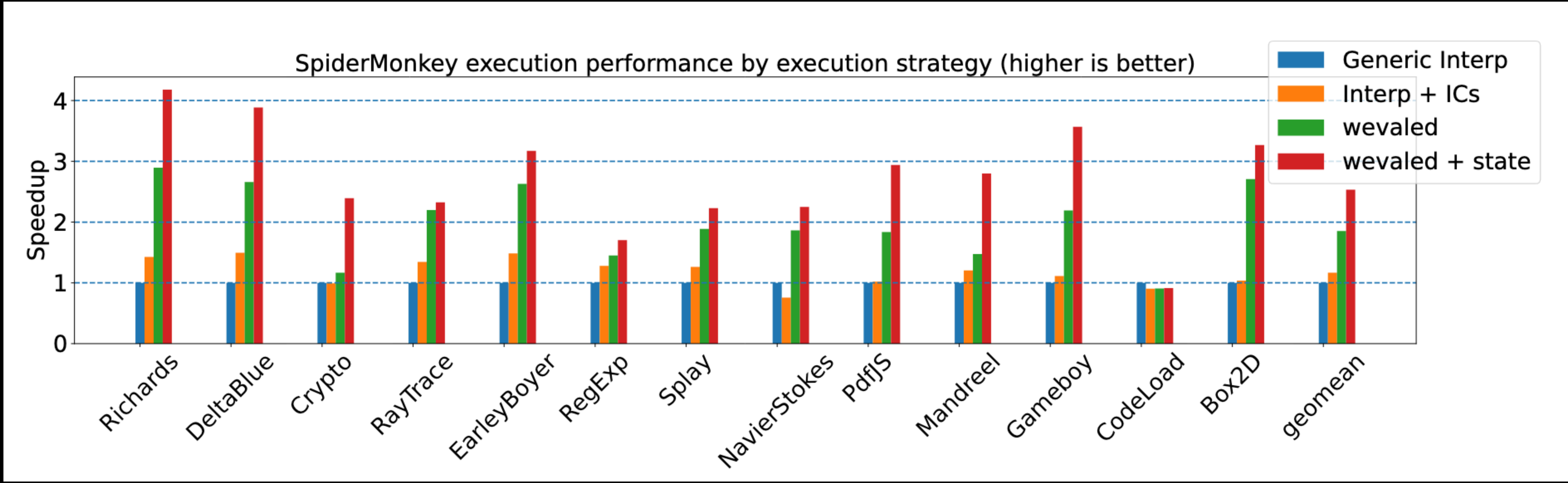
AOT Baseline Compilation



~100%* of execution occurs in Wasm bytecode in all benchmarks

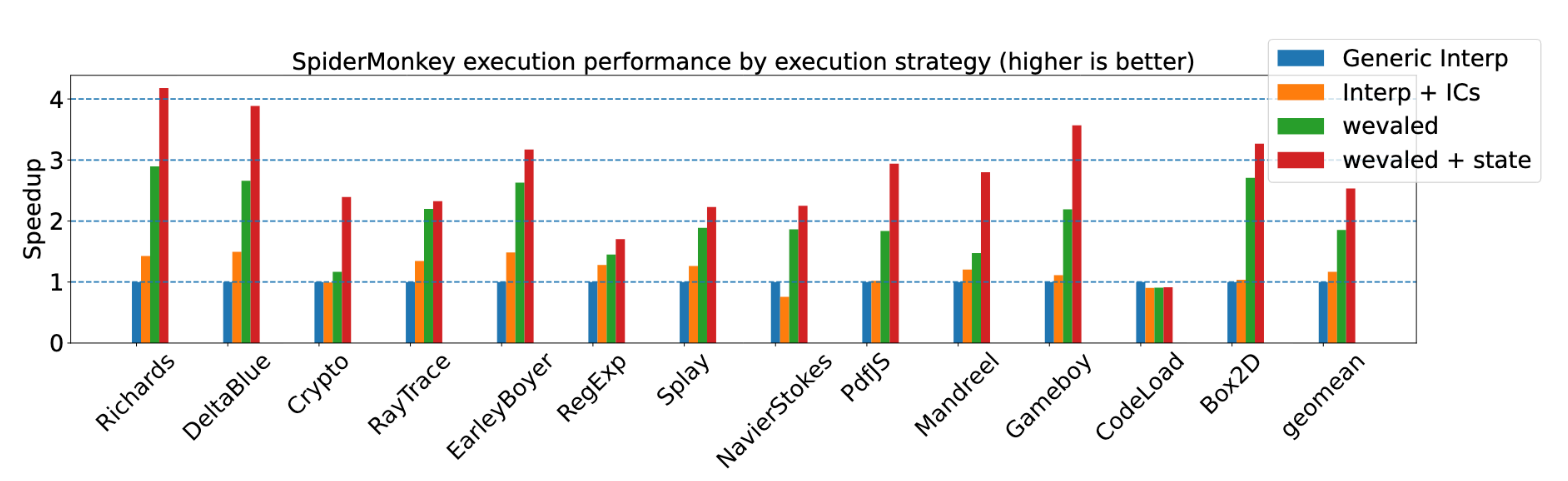
**(unsupported: async resume, debugging)*

Results



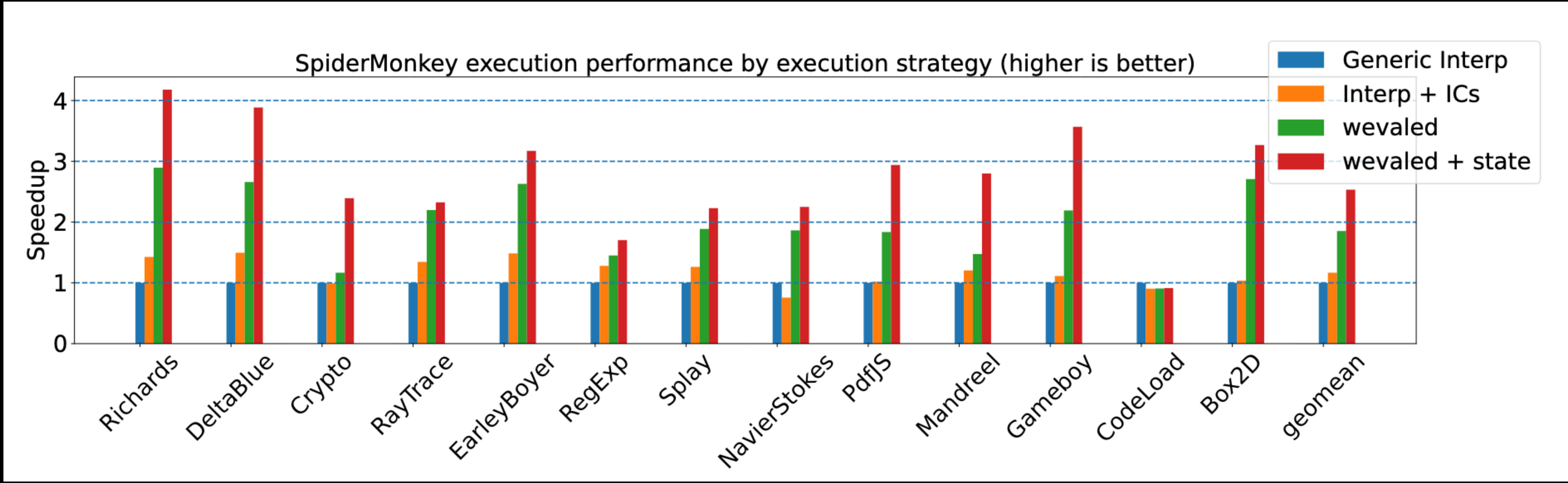
- 2.17x geomean (2.93x max) speedup over interpreter with ICs
- 2.53x geomean (4.18x max) speedup over generic interpreter (original)
- Remaining gap vs. native interp-to-baseline-compile due to Wasm call overhead

Results



- This is in production (`--enable-aot`), with speedups seen in real use-cases!

Results



- This is in production (`--enable-aot`), with speedups seen in real use-cases!
- In the paper: results on Lua interpreter and toy interpreter to show generality

Thanks! Questions?

Partial Evaluation, Whole-Program Compilation

Chris Fallin (F5), Maxwell Bernstein (Recurse Center)