Optimizing JavaScript

Filip Pizlo
Apple

- Untyped
- Objects are hashtables
- Functions are objects

```
var scale = 1.2;

function foo(o) {
    return scale * Math.sqrt(o.x * o.x + o.y * o.y);
}

for (var i = 0; i < 100; ++i)
    print(foo({x:1.5, y:2.5}));
```

History

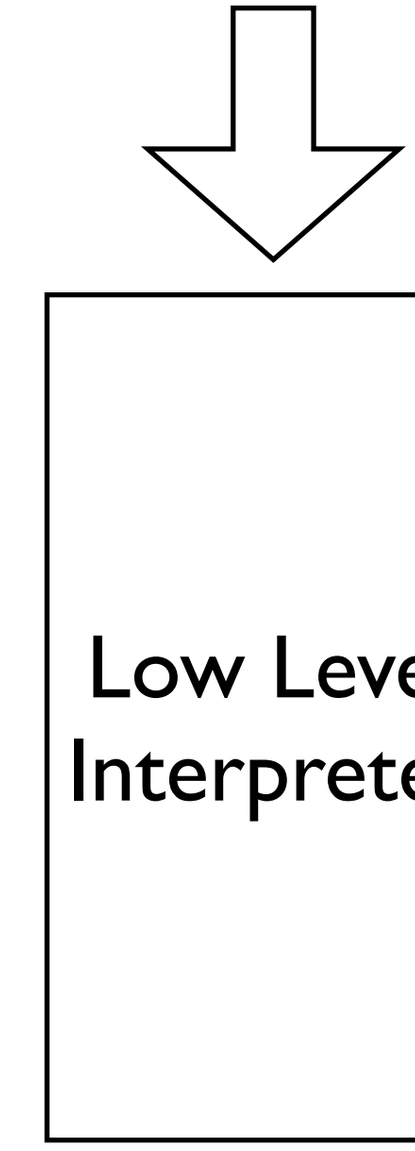
- Smalltalk
  - Deutsch and Schiffman POPL'84
- Self
  - Smith and Ungar OOPSLA'87
  - Holze, Chambers, Ungar ECOOP'91
- widely used in JavaScript
- many, many more recent papers


N
NE
NW
E
W
S
SW
SE

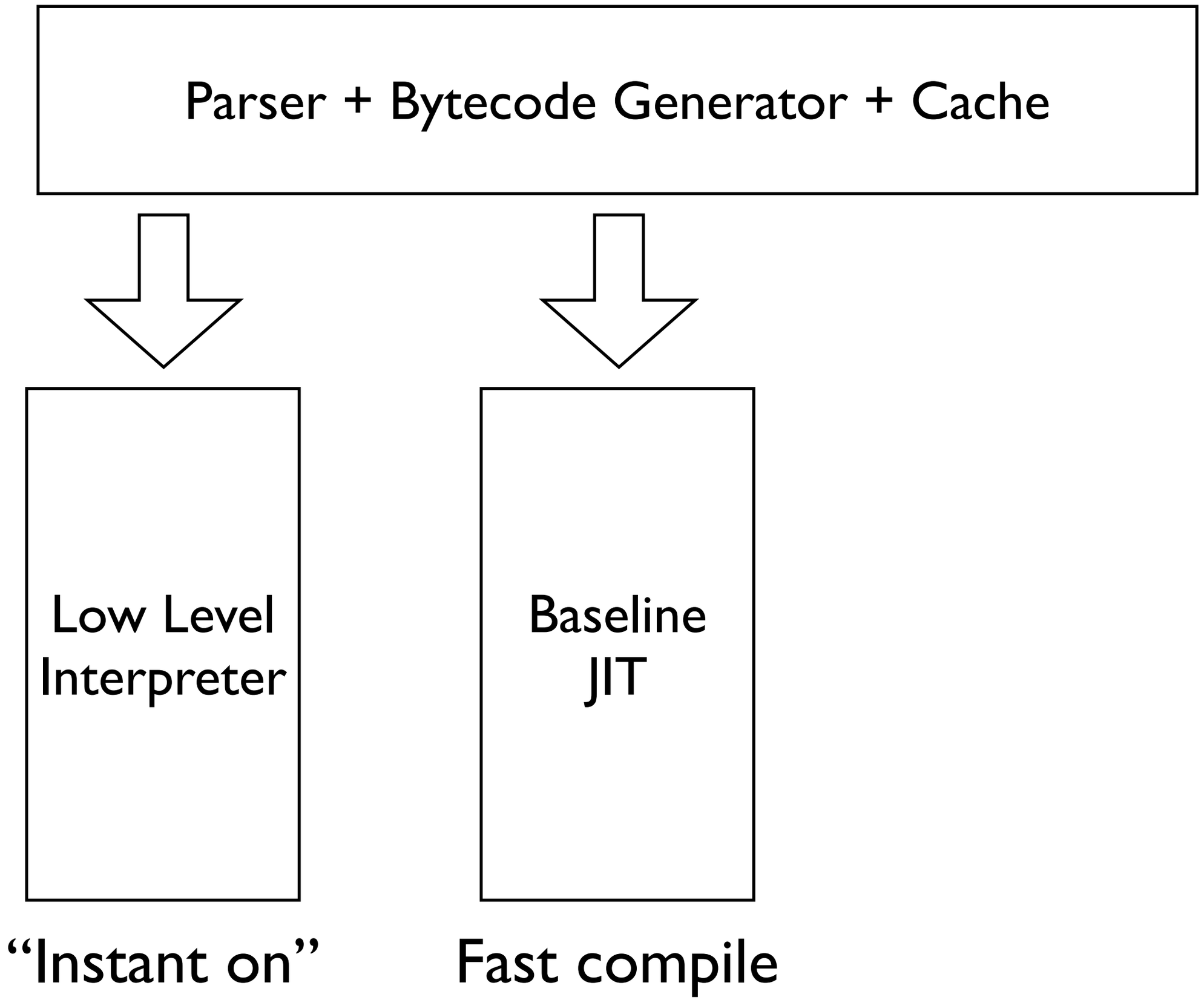
- WebKit open source project
- JavaScriptCore virtual machine
- www.webkit.org

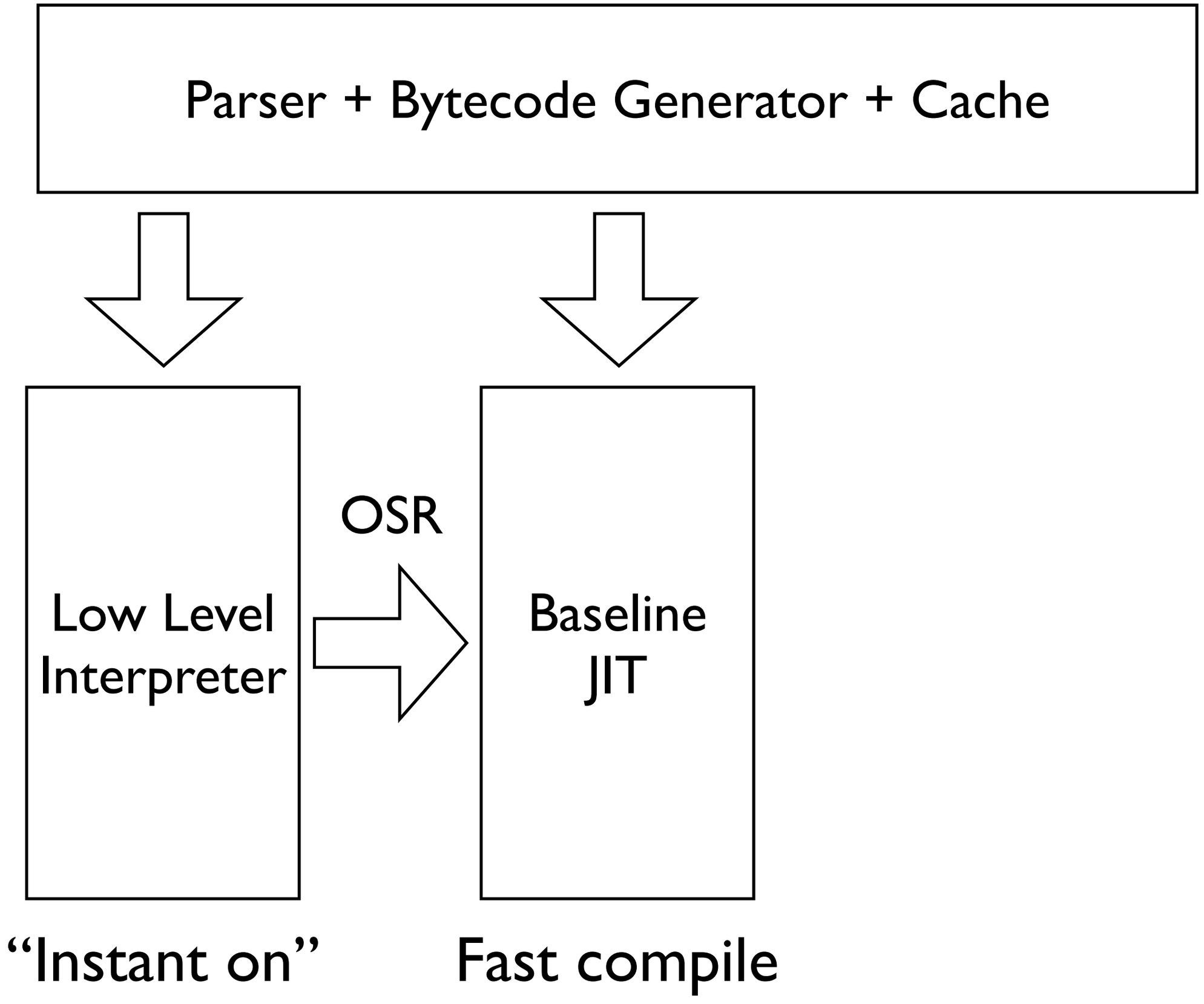
Parser + Bytecode Generator + Cache

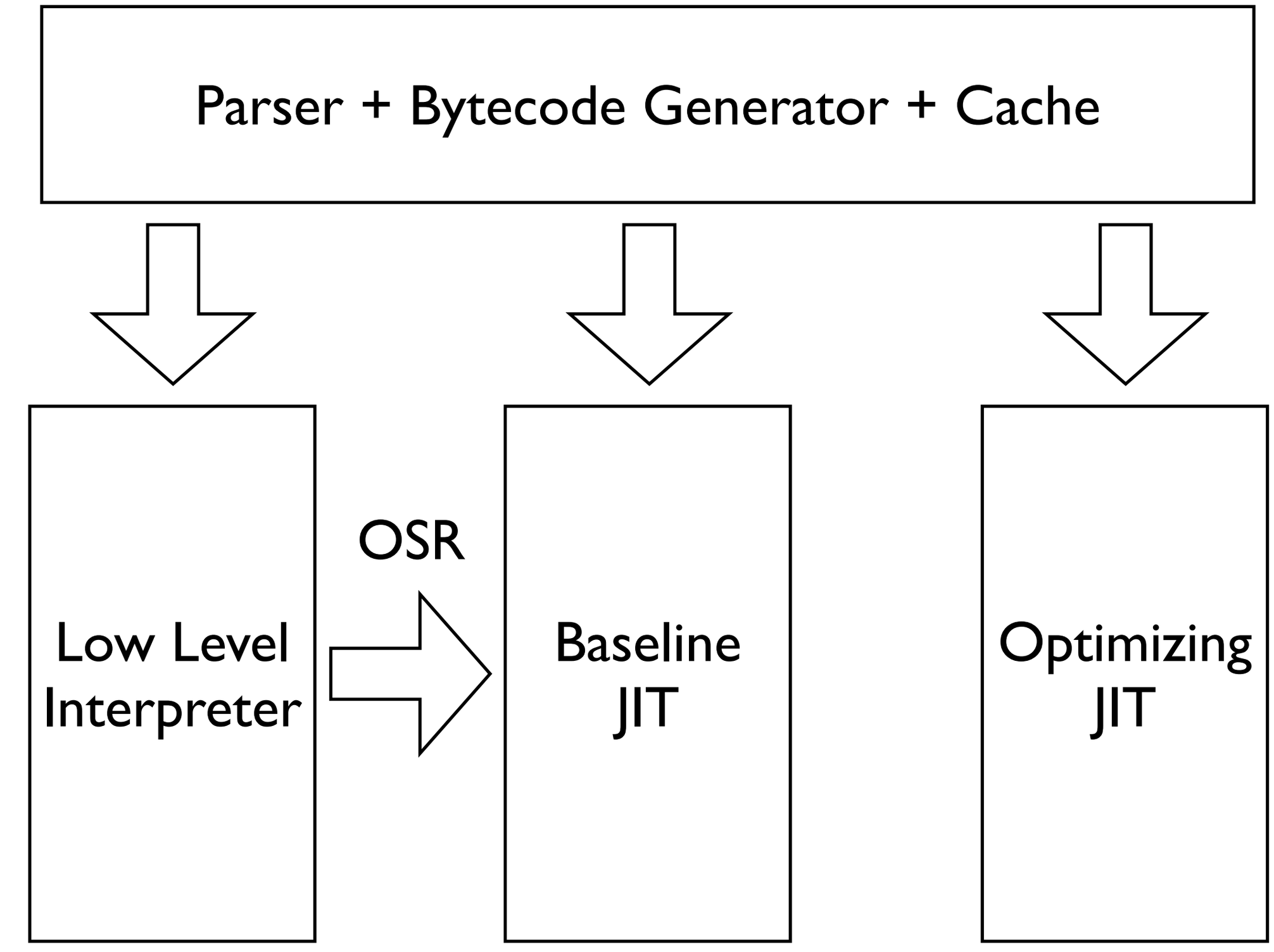
Parser + Bytecode Generator + Cache

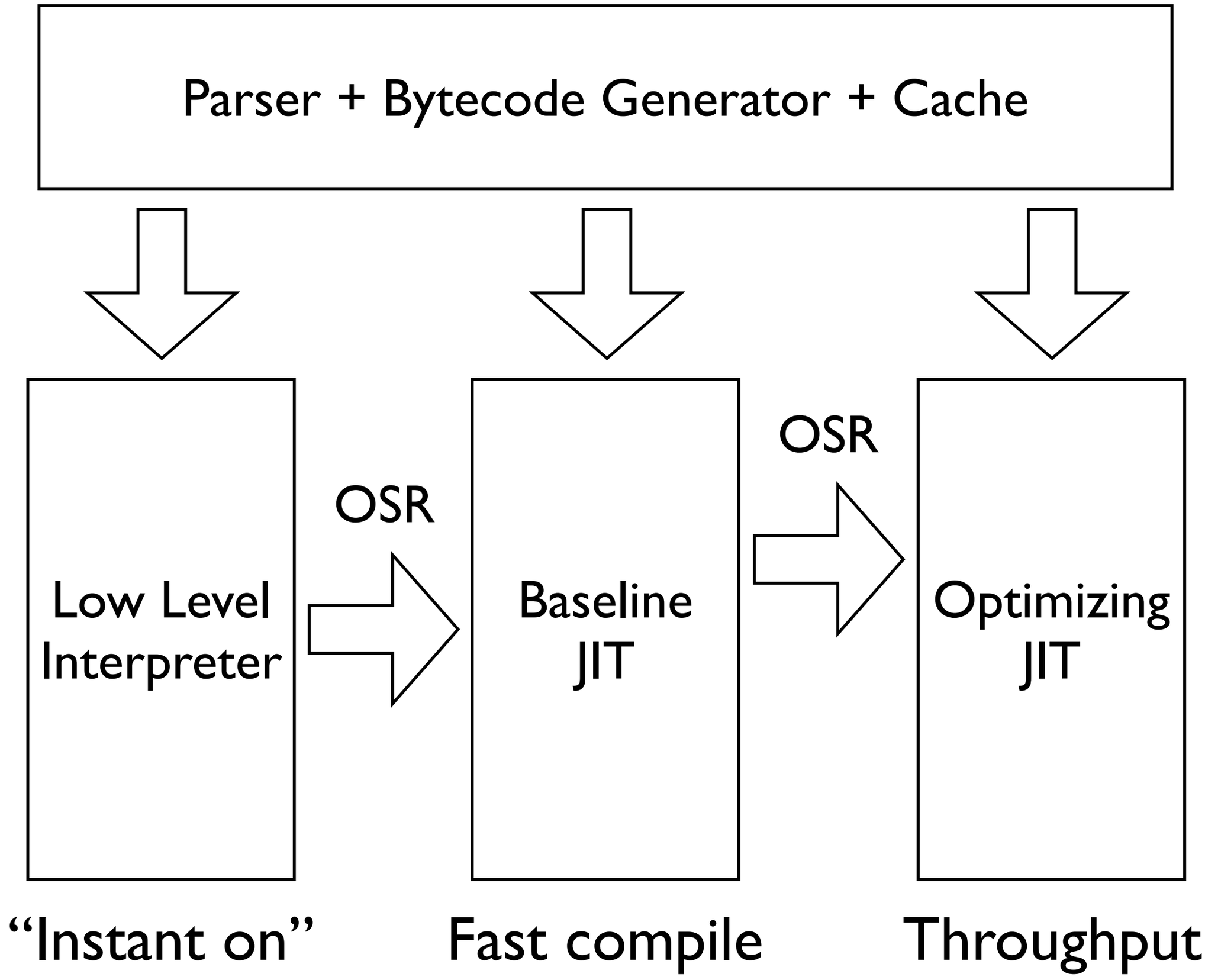


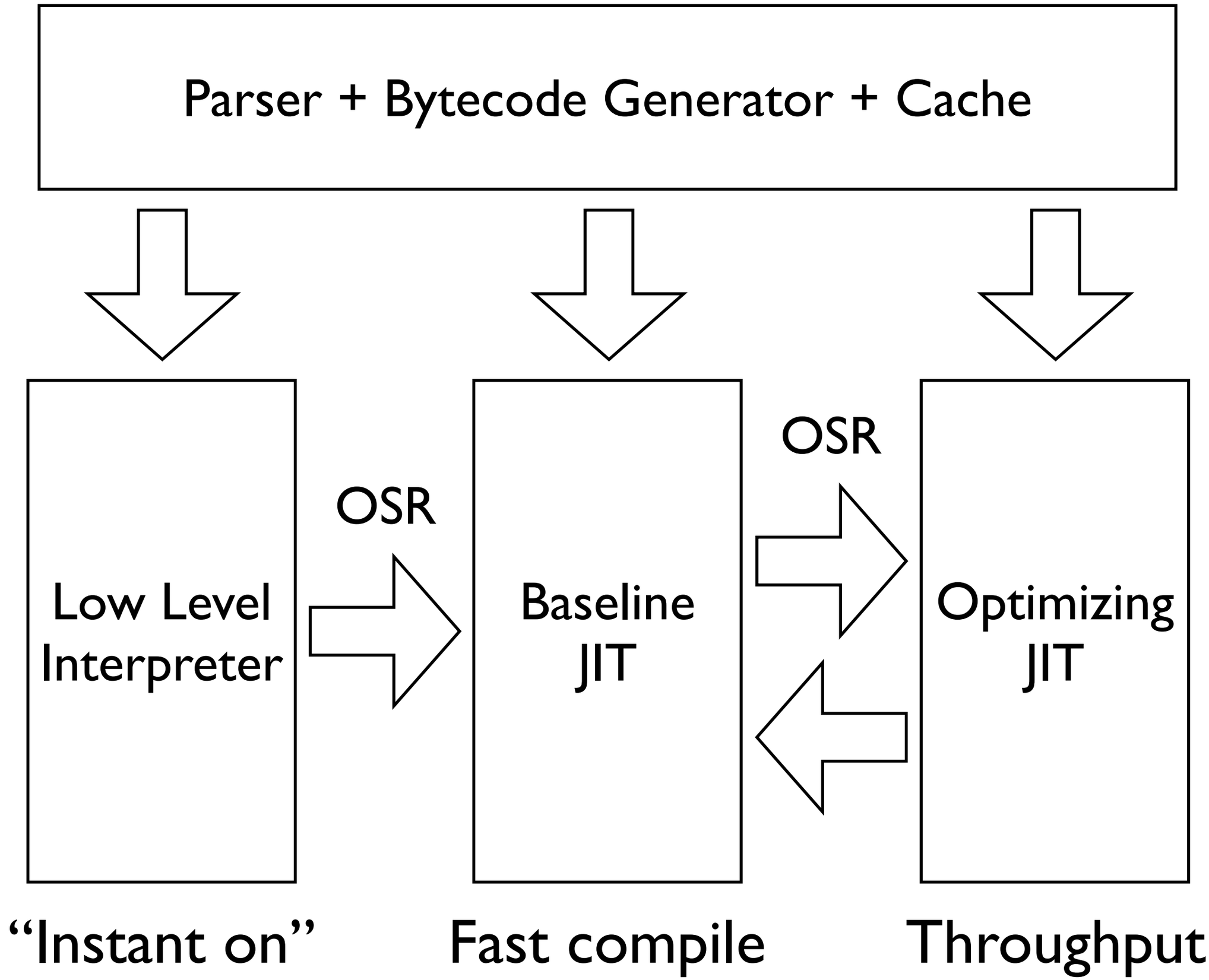
“Instant on”

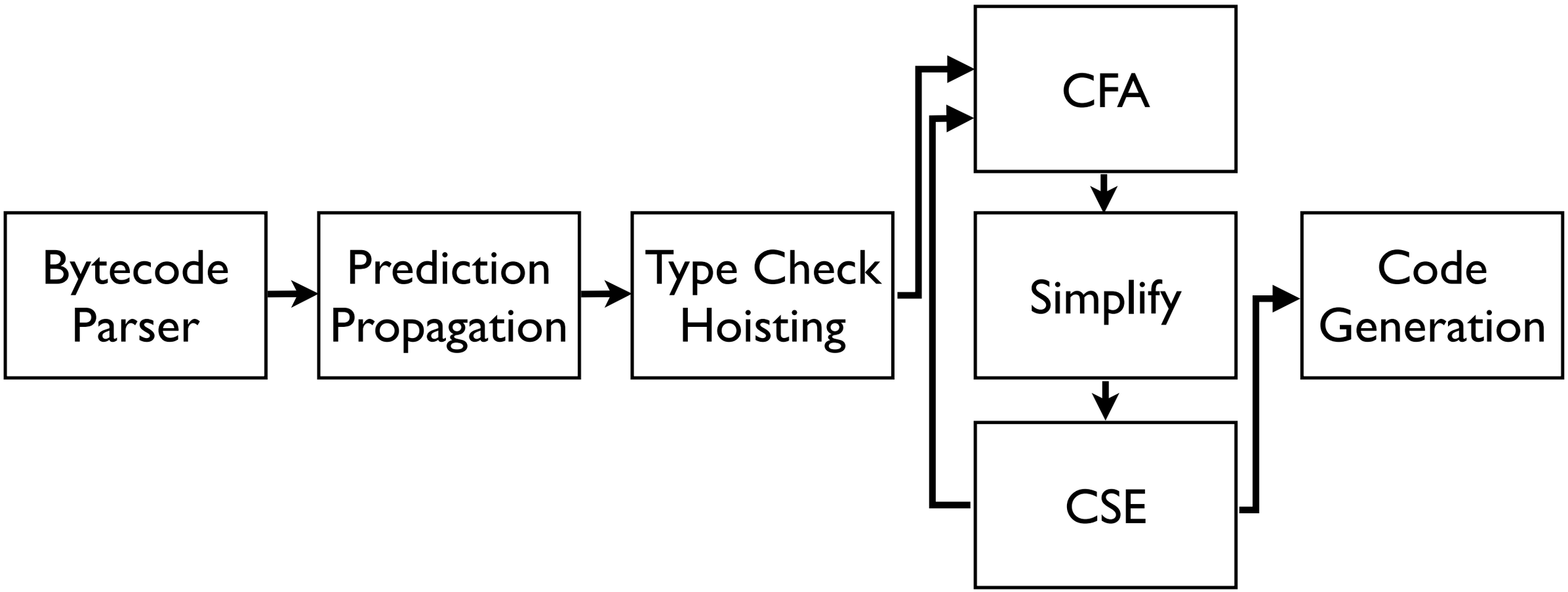

Parser + Bytecode Generator + Cache
Low Level Interpreter
Baseline JIT
“Instant on”
Fast compile


Parser + Bytecode Generator + Cache
Low Level Interpreter
OSR
Baseline JIT
“Instant on”
Fast compile


Parser + Bytecode Generator + Cache
Low Level Interpreter
OSR
Baseline JIT
Optimizing JIT
“Instant on”
Fast compile
Throughput


Parser + Bytecode Generator + Cache
Low Level Interpreter
OSR
Baseline JIT
OSR
Optimizing JIT
"Instant on"
Fast compile
Throughput


Parser + Bytecode Generator + Cache
Low Level Interpreter
OSR
Baseline JIT
OSR
Optimizing JIT
"Instant on"
Fast compile
Throughput


Bytecode Parser
Prediction Propagation
Type Check Hoisting
CFA
Simplify
CSE
Code Generation

- Martin Richards' PL benchmark

- Martin Richards' PL benchmark
  - C & Java: 1.2ms

- Martin Richards' PL benchmark
  - C & Java: 1.2ms
  - Simple JS interpreter: 129ms

- Martin Richards' PL benchmark
  - C & Java: 1.2ms
  - Simple JS interpreter: 129ms
  - Low Level Interpreter: 58ms

- Martin Richards' PL benchmark
  - C & Java: 1.2ms
  - Simple JS interpreter: 129ms
  - Low Level Interpreter: 58ms
  - Baseline JIT: 8.4ms

- Martin Richards' PL benchmark
  - C & Java: 1.2ms
  - Simple JS interpreter: 129ms
  - Low Level Interpreter: 58ms
  - Baseline JIT: 8.4ms
  - Optimizing JIT: 2.1ms

1. Profile

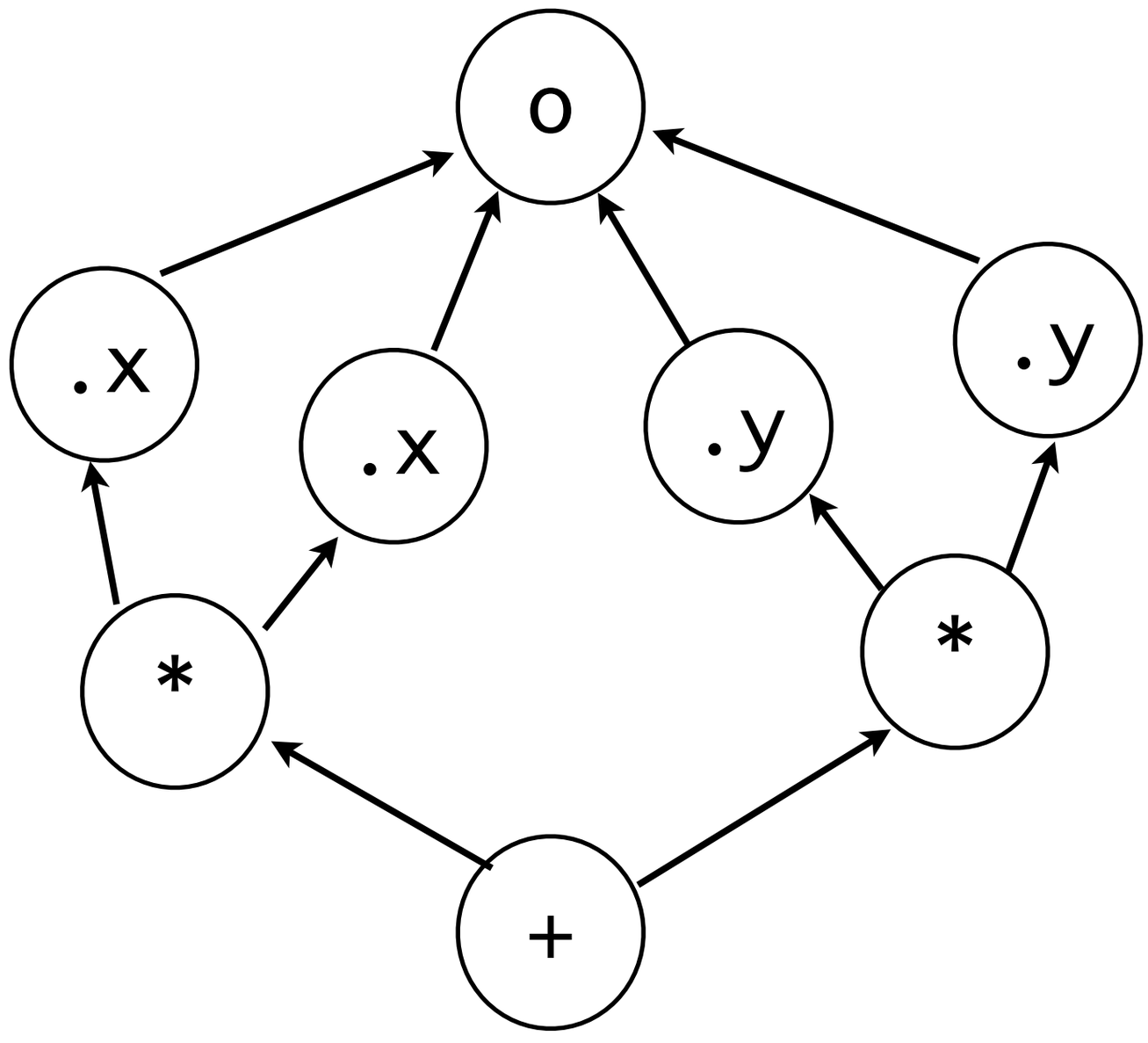
2. Predict

3. Prove

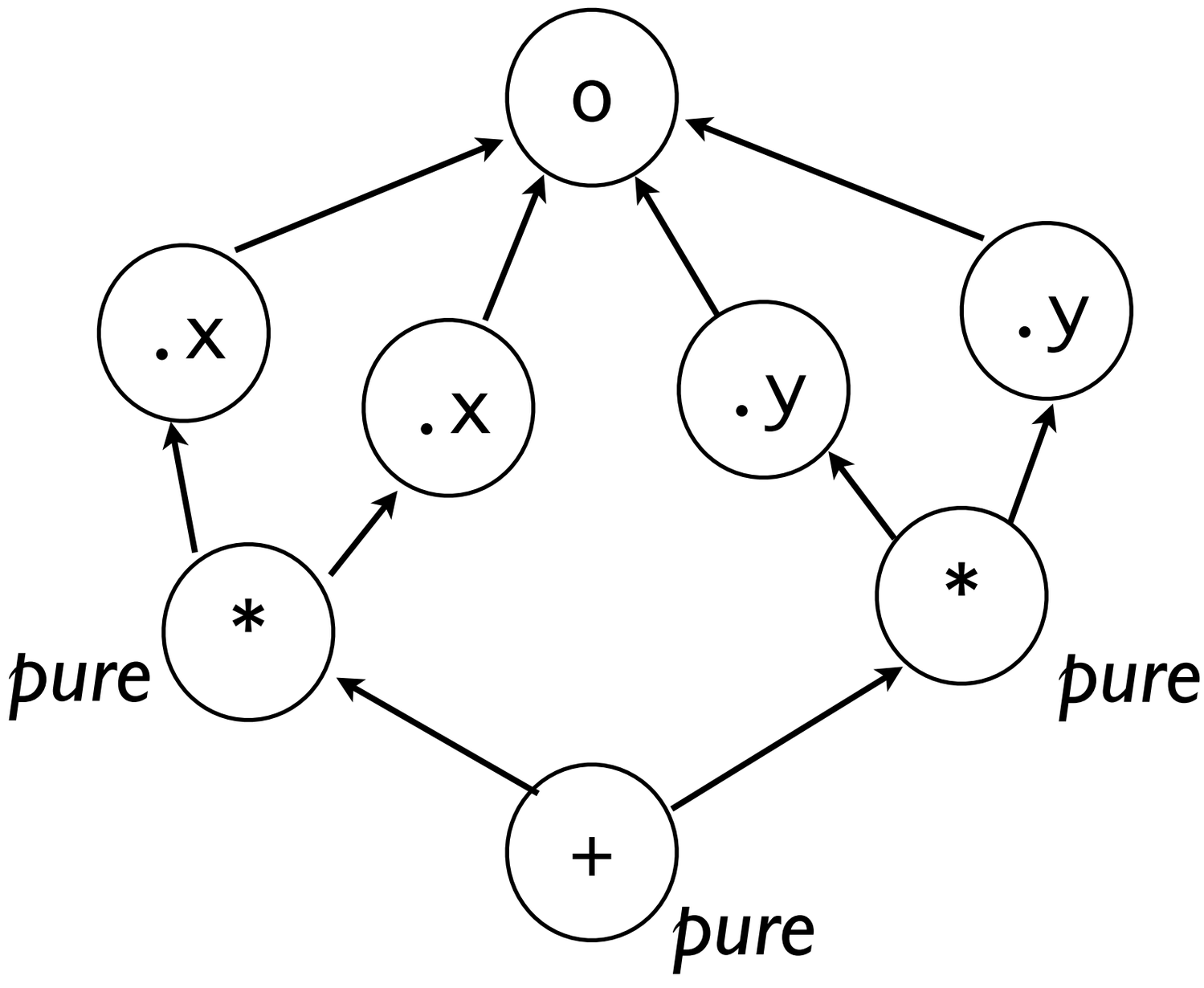
```
var scale = 1.2;

function foo(o) {
    return scale * Math.sqrt(o.x * o.x + o.y * o.y);
}

for (var i = 0; i < 100; ++i)
    print(foo({x:1.5, y:2.5}));
```

```
o.x * o.x + o.y * o.y
```

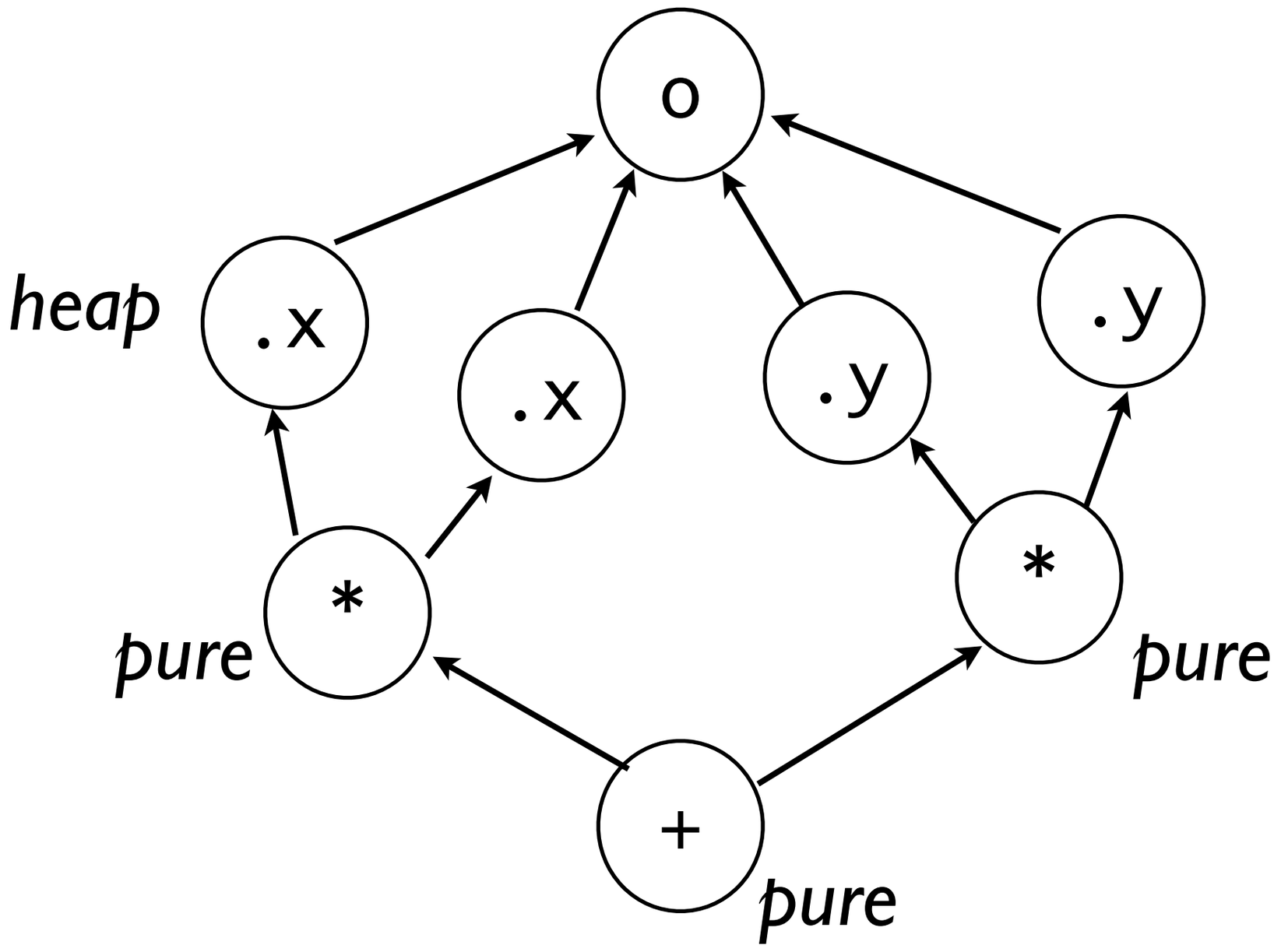
```
o.x * o.x + o.y * o.y
```



```
o.x * o.x + o.y * o.y
```



```
o.x * o.x + o.y * o.y
```



Profile

- Heap
- Arguments
- Call returns

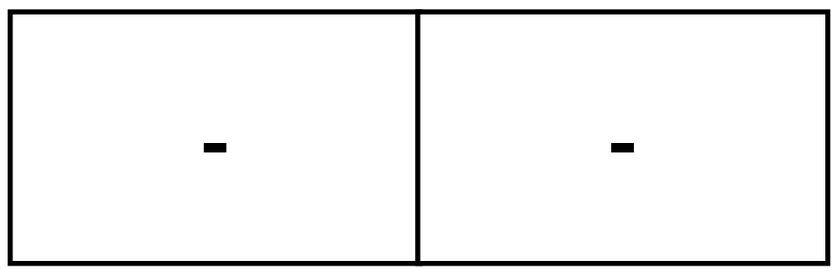
JITPropertyAccess.cpp

```
void JIT::emit_op_get_by_id(Instruction* currentInstruction)
{
    unsigned resultVReg = currentInstruction[1].u.operand;
    unsigned baseVReg = currentInstruction[2].u.operand;
    Identifier* ident = &(m_codeBlock->
        identifier(currentInstruction[3].u.operand));

    emitGetVirtualRegister(baseVReg, regT0);
    compileGetByIdHotPath(baseVReg, ident);
    emitValueProfilingSite();
    emitPutVirtualRegister(resultVReg);
}
```

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile



- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

0	-

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

5	-

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

0.5	-

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

7	-

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

7	Int32

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

4.5	Int32

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

9.5	Int32

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

10.1	Int32

- Unpredictable values are profiled.
- Every ~1000 executions of a function, a bounding type is computed for each profile.

```
var x = o.f;
```

ValueProfile

10.1	Int32 ∪ Double

Predict

- Heap: type that bounds all values seen
- Pure: abstract interpretation

DFGPredictionPropagationPhase.cpp
(roughly)

```
case ArithMul: {
    SpeculatedType left = node->child1()->prediction();
    SpeculatedType right = node->child2()->prediction();

if (left && right) {
    if (isInt32(left) && isInt32(right))
        changed |= mergePrediction(SpecInt32);
    else
        changed |= mergePrediction(SpecDouble);
}
```

Prove

ArithMul will *spec-fail* if its operands are not numbers.

- Code size reduction
- Type propagation

We know that an ArithMul that is predicted double will always produce a double.

```
.
.
.
c: ArithMul(@a, @b)
.
.
.
```

We know that an ArithMul that is predicted double will always produce a double.

```
.
.
.
c: ArithMul(@a, @b)
.
.
.
```

know nothing about a, b

We know that an ArithMul that is predicted double will always produce a double.

```
.
.
.
c: ArithMul(@a, @b)
.
.
.
```

know nothing about a, b

know that a, b, c must be double

```
[  61] mul       r5, r5, r6
          0x10b05169c: mov %rax, %rdx
          0x10b05169f: mov 0x28(%r13), %rax
          0x10b0516a3: cmp %r14, %rax
          0x10b0516a6: jb 0x10b051b1b
          0x10b0516ac: cmp %r14, %rdx
          0x10b0516af: jb 0x10b051b47
          0x10b0516b5: mov %rax, %rcx
          0x10b0516b8: imul %edx, %ecx
          0x10b0516bb: jo 0x10b051ada
          0x10b0516c1: test %ecx, %ecx
          0x10b0516c3: jnz 0x10b0516ee
          0x10b0516c9: cmp $0x0, %eax
          0x10b0516cc: jl 0x10b0516db
          0x10b0516d2: cmp $0x0, %edx
          0x10b0516d5: jge 0x10b0516ee
          0x10b0516db: mov $0x10af99bfc, %r11
          0x10b0516e5: add $0x1, (%r11)
          0x10b0516e9: jmp 0x10b051ada
          0x10b0516ee: mov %rcx, %rax
          0x10b0516f1: or %r14, %rax
          0x10b0516f4: mov %rax, 0x28(%r13)
```

```
      28:             <!1:3>   ArithMul(d@23<Double>, d@23<Double>,
Number|MustGen|CanExit, bc#61)
              0x10b051dff: cmp %r14, %rcx
              0x10b051e02: jae 0x10b051e21
              0x10b051e08: test %rcx, %r14
              0x10b051e0b: jz 0x10b051f5c
              0x10b051e11: mov %rcx, %rax
              0x10b051e14: add %r14, %rax
              0x10b051e17: movd %rax, %xmm0
              0x10b051e1c: jmp 0x10b051e25
              0x10b051e21: cvtsi2sd %ecx, %xmm0
              0x10b051e25: movsd %xmm0, %xmm2
              0x10b051e29: mulsd %xmm0, %xmm2
```

spec fail

OSR exit

OSR exit

op_add

Bytecode

OSR exit

op_add

```
mov 0x0(%r13), %rax
mov -0x40(%r13), %rdx
cmp %r14, %rax
jb <slow path>
cmp %r14, %rdx
jb <slow path>
add %edx, %eax
jo <slow path>
or %r14, %rax
mov %rax, 0x8(%r13)
```

Bytecode

Baseline

OSR exit

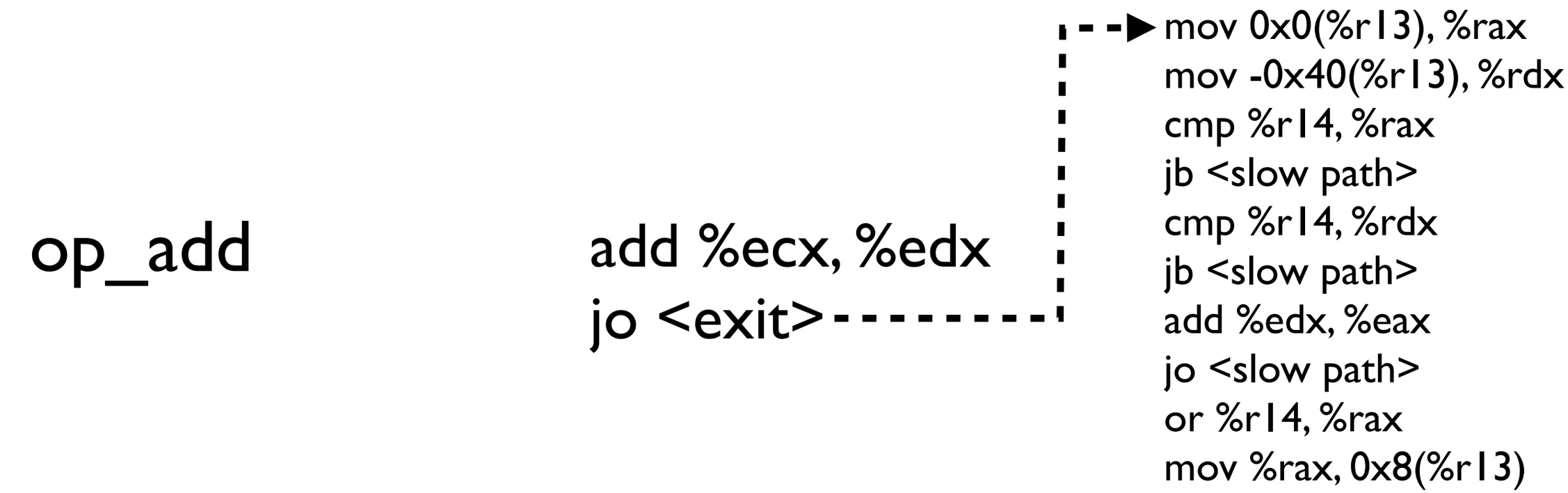
op_add

add %ecx, %edx
jo <exit>

```
mov 0x0(%r13), %rax
mov -0x40(%r13), %rdx
cmp %r14, %rax
jb <slow path>
cmp %r14, %rdx
jb <slow path>
add %edx, %eax
jo <slow path>
or %r14, %rax
mov %rax, 0x8(%r13)
```

Bytecode | Optimized | Baseline

OSR exit


op_add
add %ecx, %edx
jo <exit>
mov 0x0(%r13), %rax
mov -0x40(%r13), %rdx
cmp %r14, %rax
jb <slow path>
cmp %r14, %rdx
jb <slow path>
add %edx, %eax
jo <slow path>
or %r14, %rax
mov %rax, 0x8(%r13)
Bytecode
Optimized
Baseline

OSR exit

```
add %ecx, %edx
jo <exit>
```

Optimized

```
mov 0x0(%r13), %rax
mov -0x40(%r13), %rdx
cmp %r14, %rax
jb <slow path>
cmp %r14, %rdx
jb <slow path>
add %edx, %eax
jo <slow path>
or %r14, %rax
mov %rax, 0x8(%r13)
```

Baseline

OSR exit

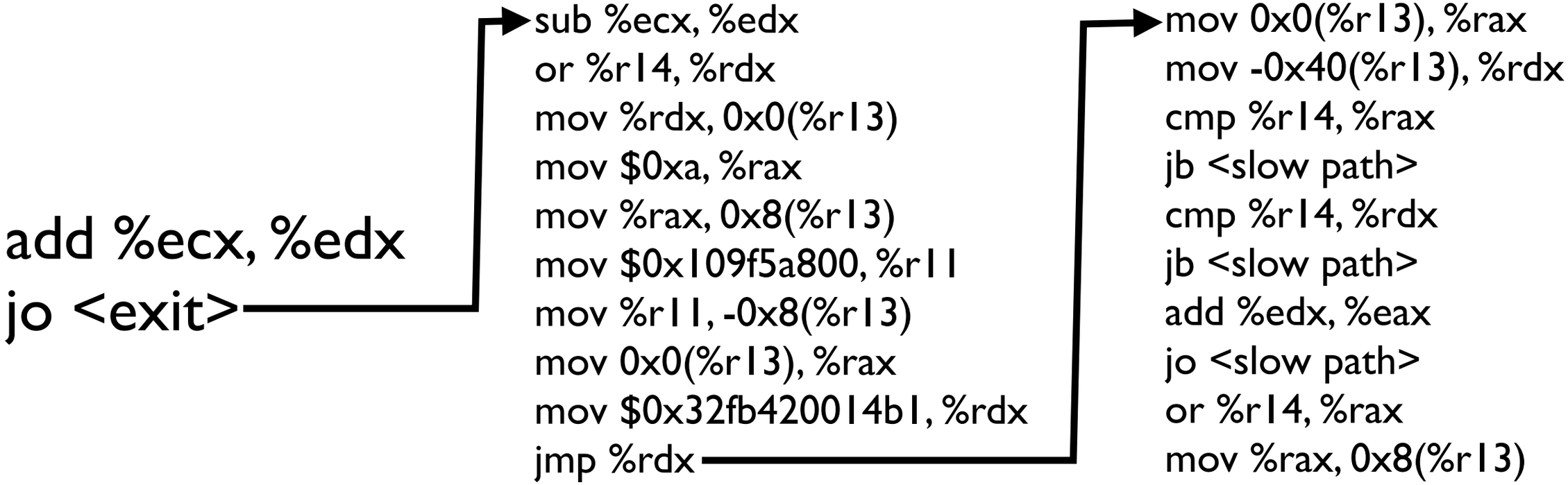
```
add %ecx, %edx
jo <exit>
```

Optimized

```
mov 0x0(%r13), %rax
mov -0x40(%r13), %rdx
cmp %r14, %rax
jb <slow path>
cmp %r14, %rdx
jb <slow path>
add %edx, %eax
jo <slow path>
or %r14, %rax
mov %rax, 0x8(%r13)
```

Baseline

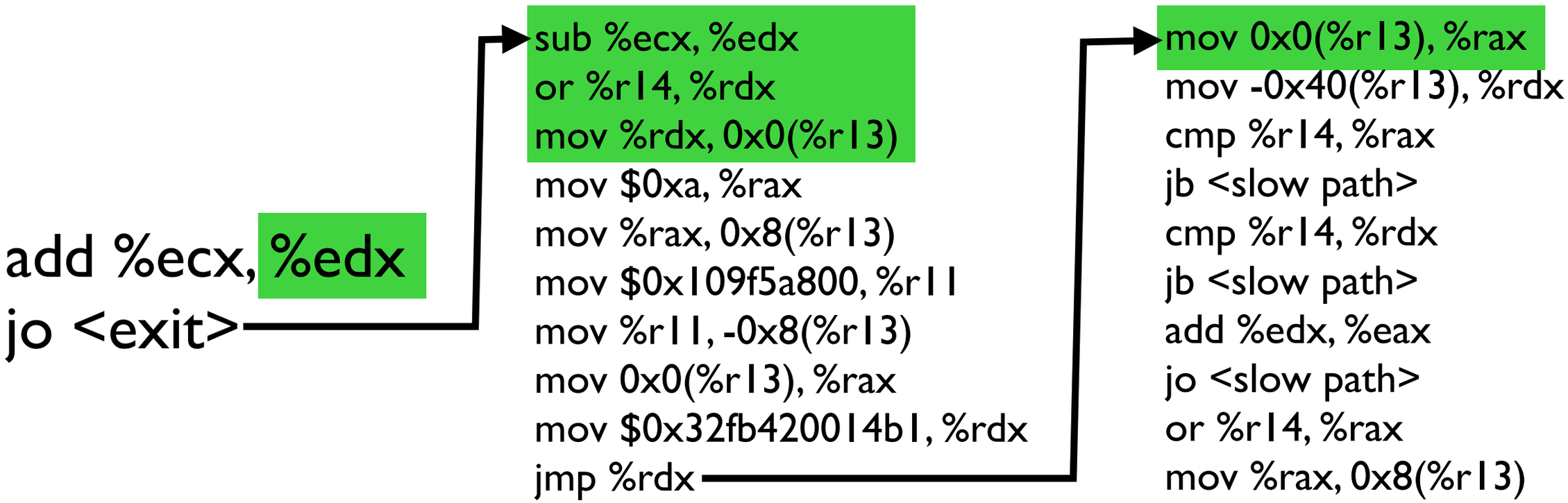
OSR exit



Optimized

Baseline

OSR exit

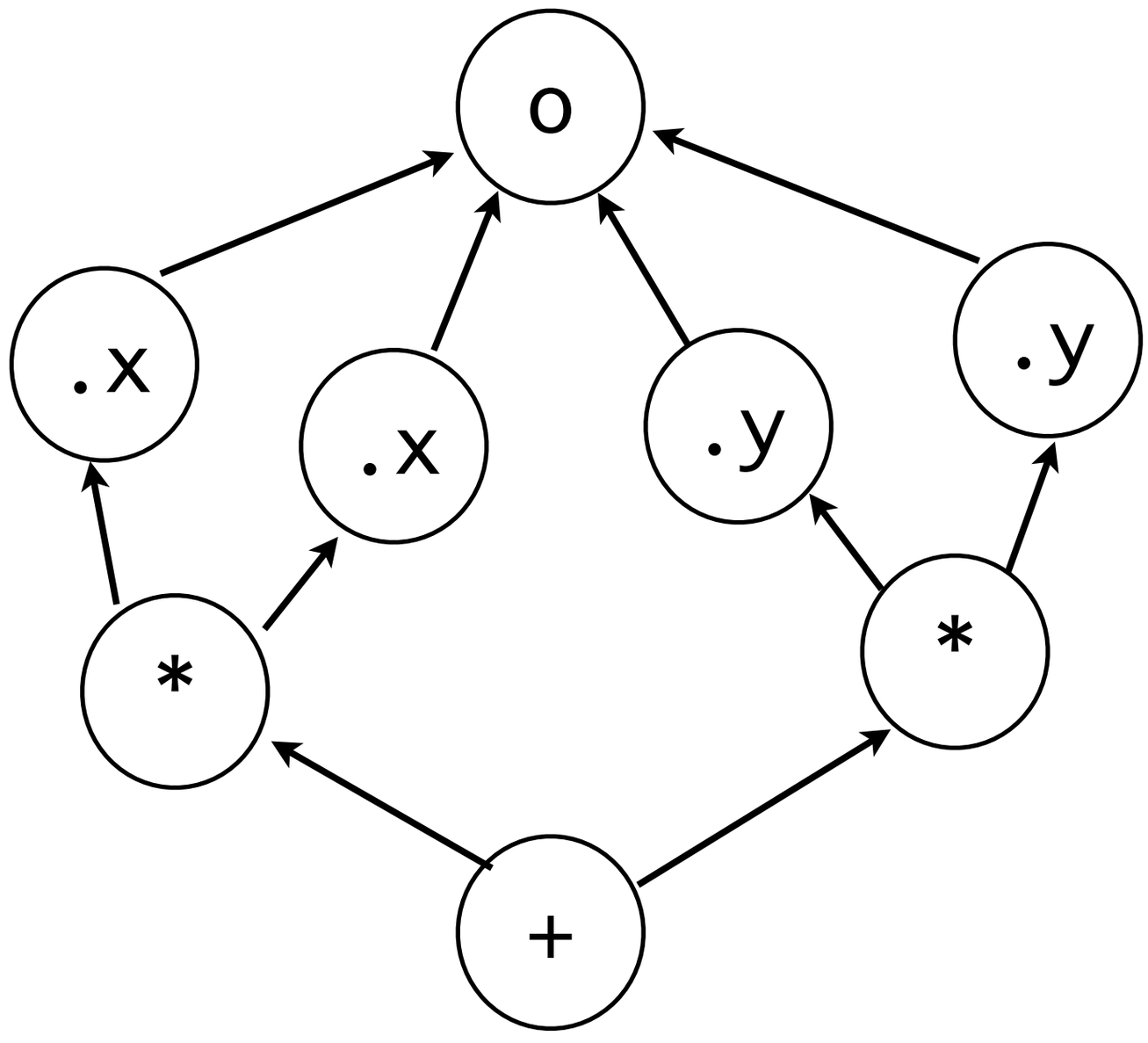


Optimized

Baseline

- Protect the main path
- Record why we exited
- Recompile with exponential backoff

```
o.x * o.x + o.y * o.y
```



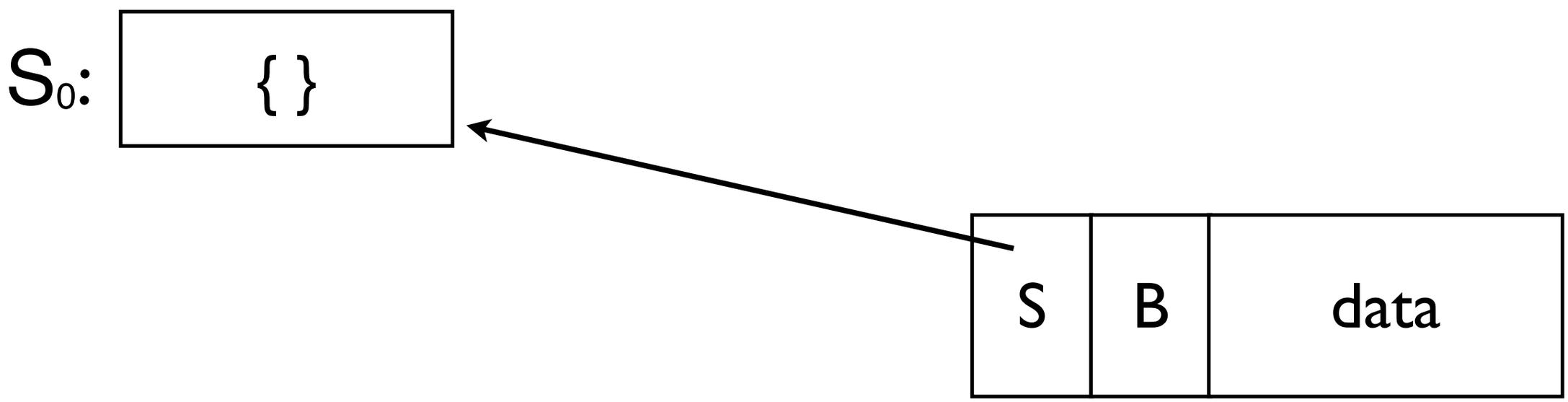
```
[  66] add       r2, r4, r5
          0x10b0516f8: mov %rax, %rdx
          0x10b0516fb: mov 0x20(%r13), %rax
          0x10b0516ff: cmp %r14, %rax
          0x10b051702: jb 0x10b051bc0
          0x10b051708: cmp %r14, %rdx
          0x10b05170b: jb 0x10b051bda
          0x10b051711: add %edx, %eax
          0x10b051713: jo 0x10b051b7f
          0x10b051719: or %r14, %rax
          0x10b05171c: mov %rax, 0x10(%r13)
```

```
30:                <!1:3>   ArithAdd(d@20<Double>, d@28<Double>,
NumberlMustGen, bc#66)
                0x10b051e2d: addsd %xmm2, %xmm1
```

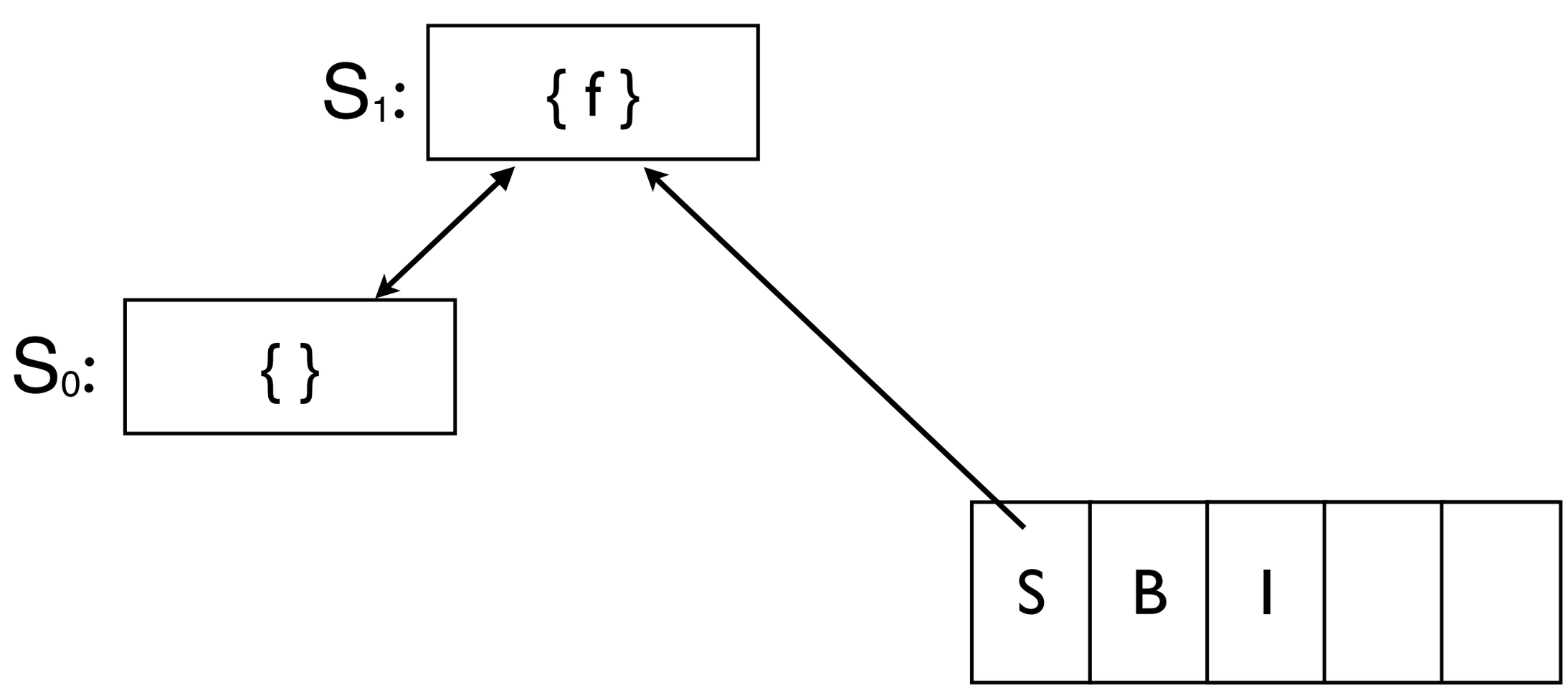
Objects

```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```

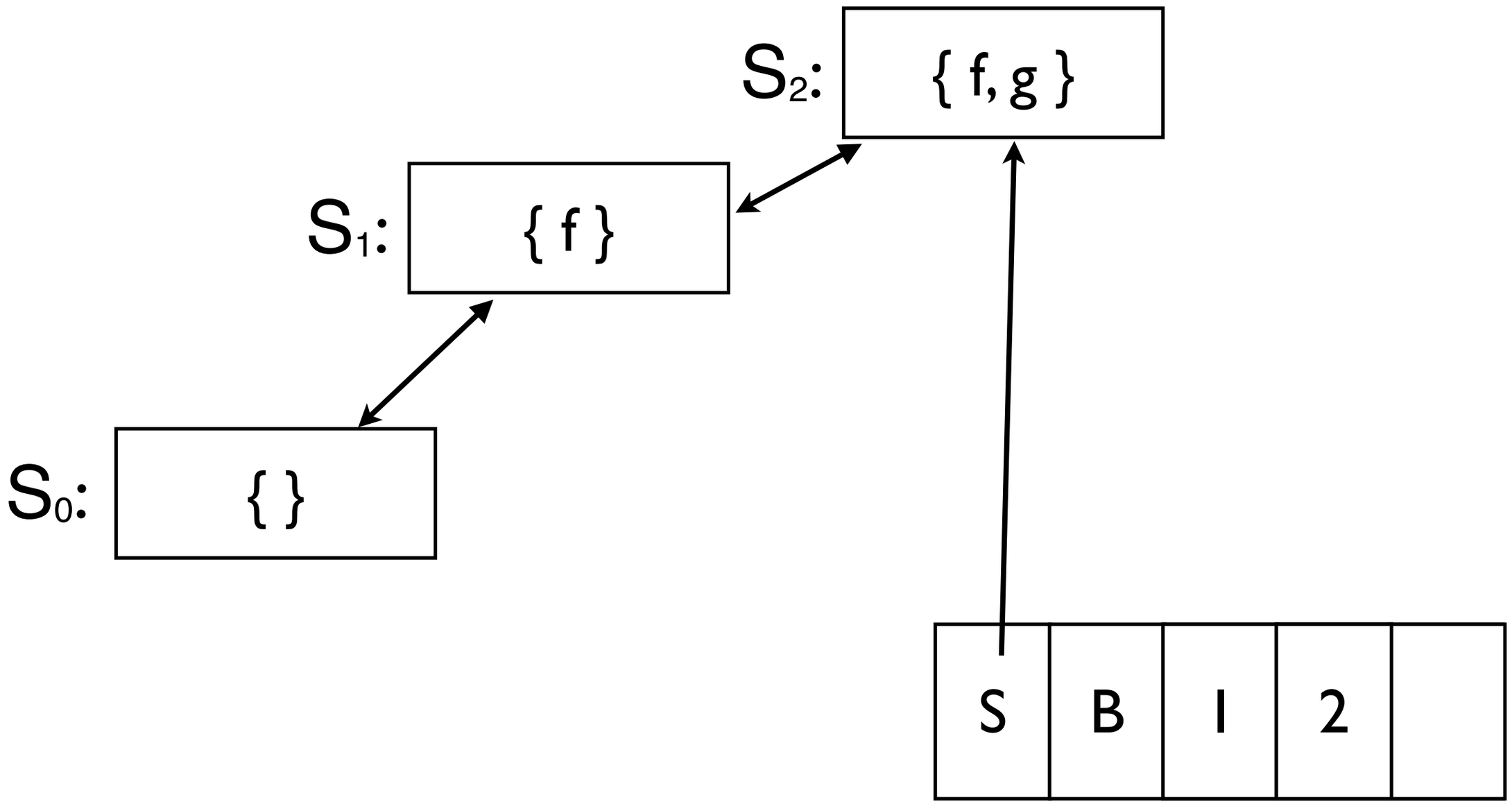
```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



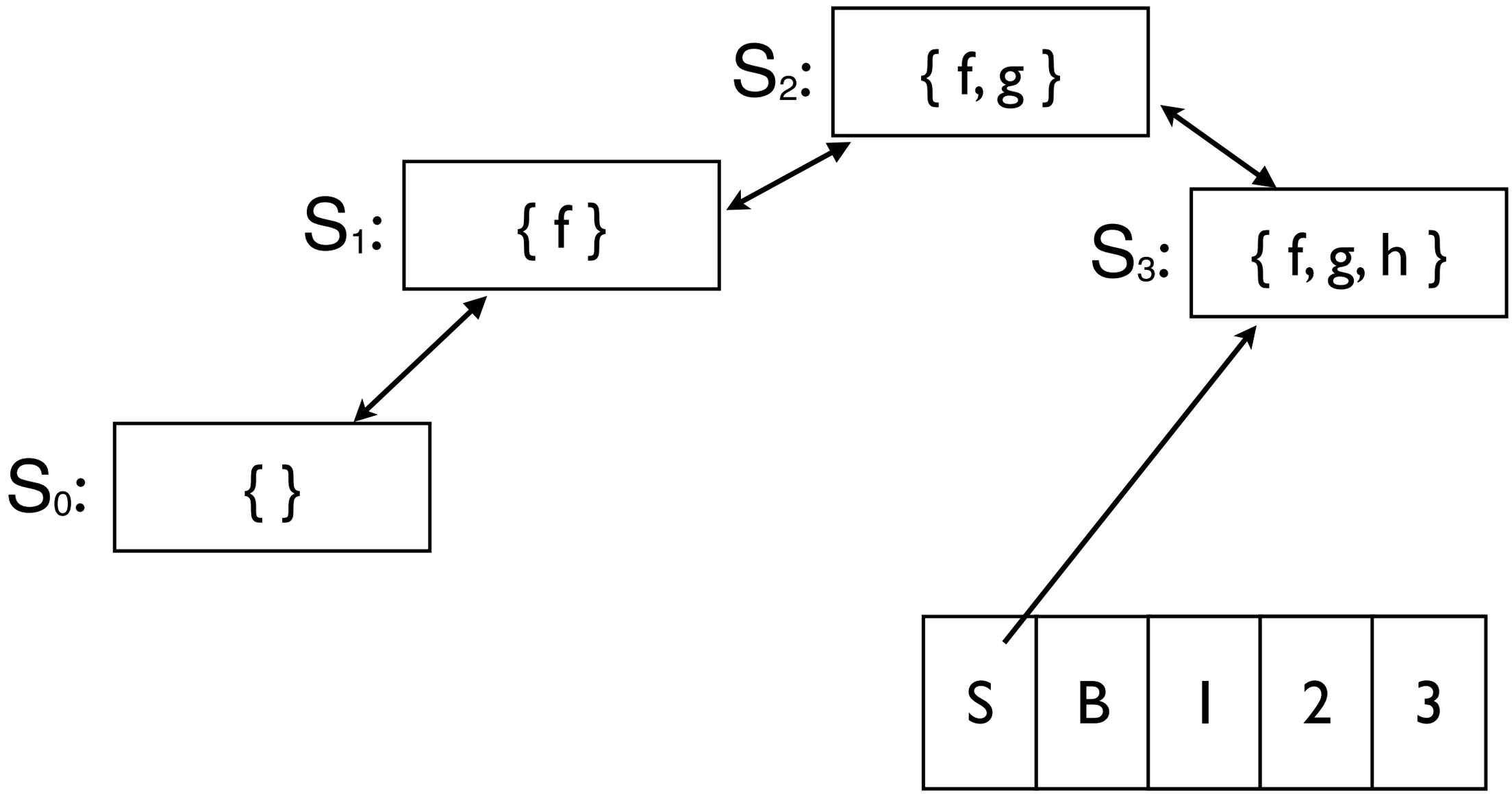
```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



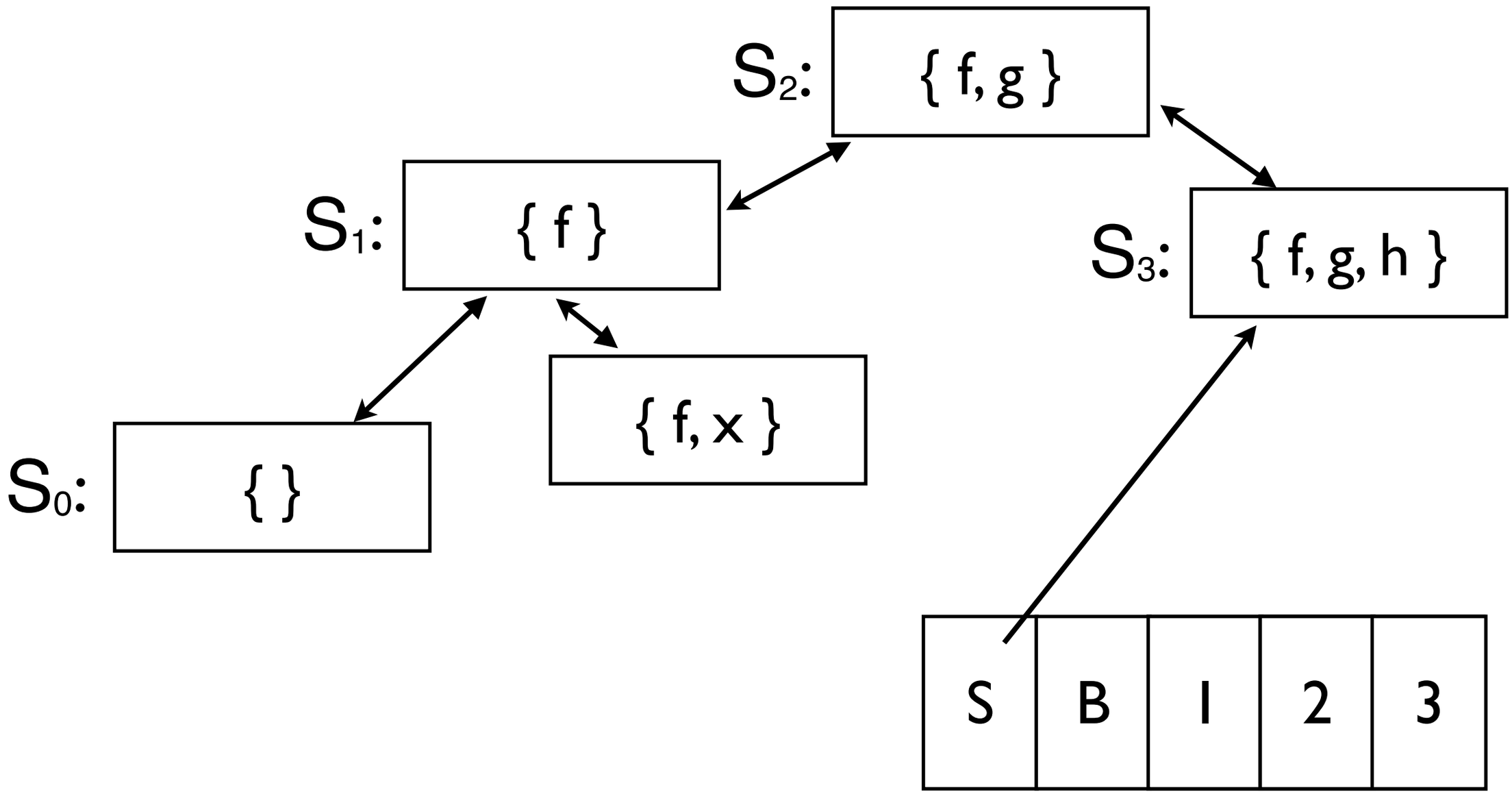
```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



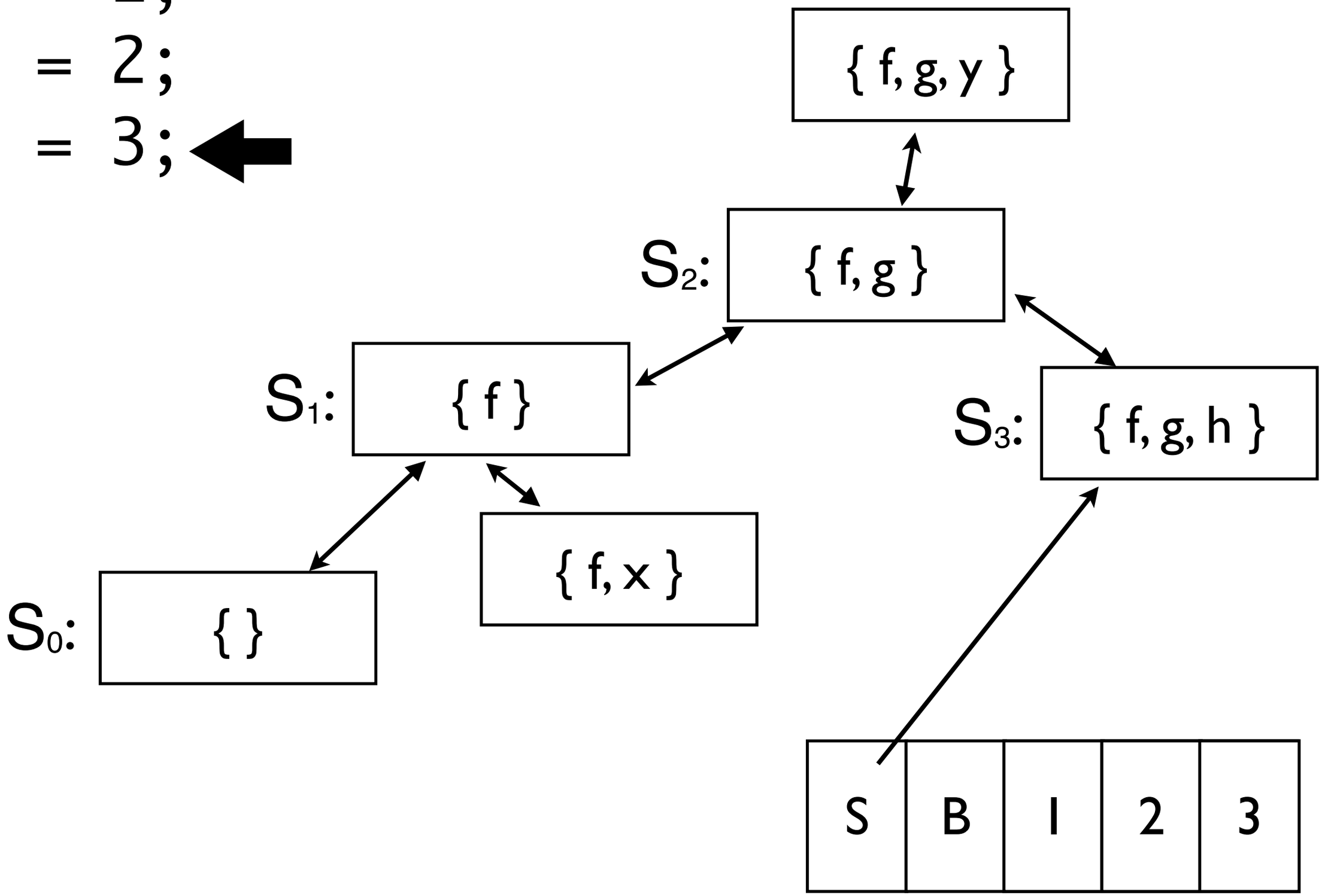
```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



```
var o = new Object();
o.f = 1;
o.g = 2;
o.h = 3;
```



```
var x = o.f;
```

```
cmpq S3, (%rax)
jne _slowPath
movq 16(%rax), %rax
```

```
o.f = x;
```

```
cmpq S0, (%rax)
jne _slowPath
movq S1, (%rax)
movq %rdx, 16(%rax)
```

```
var scale = 1.2;

function foo(o) {
    return scale * Math.sqrt(o.x * o.x + o.y * o.y);
}

for (var i = 0; i < 100; ++i)
    print(foo({x:1.5, y:2.5}));
```

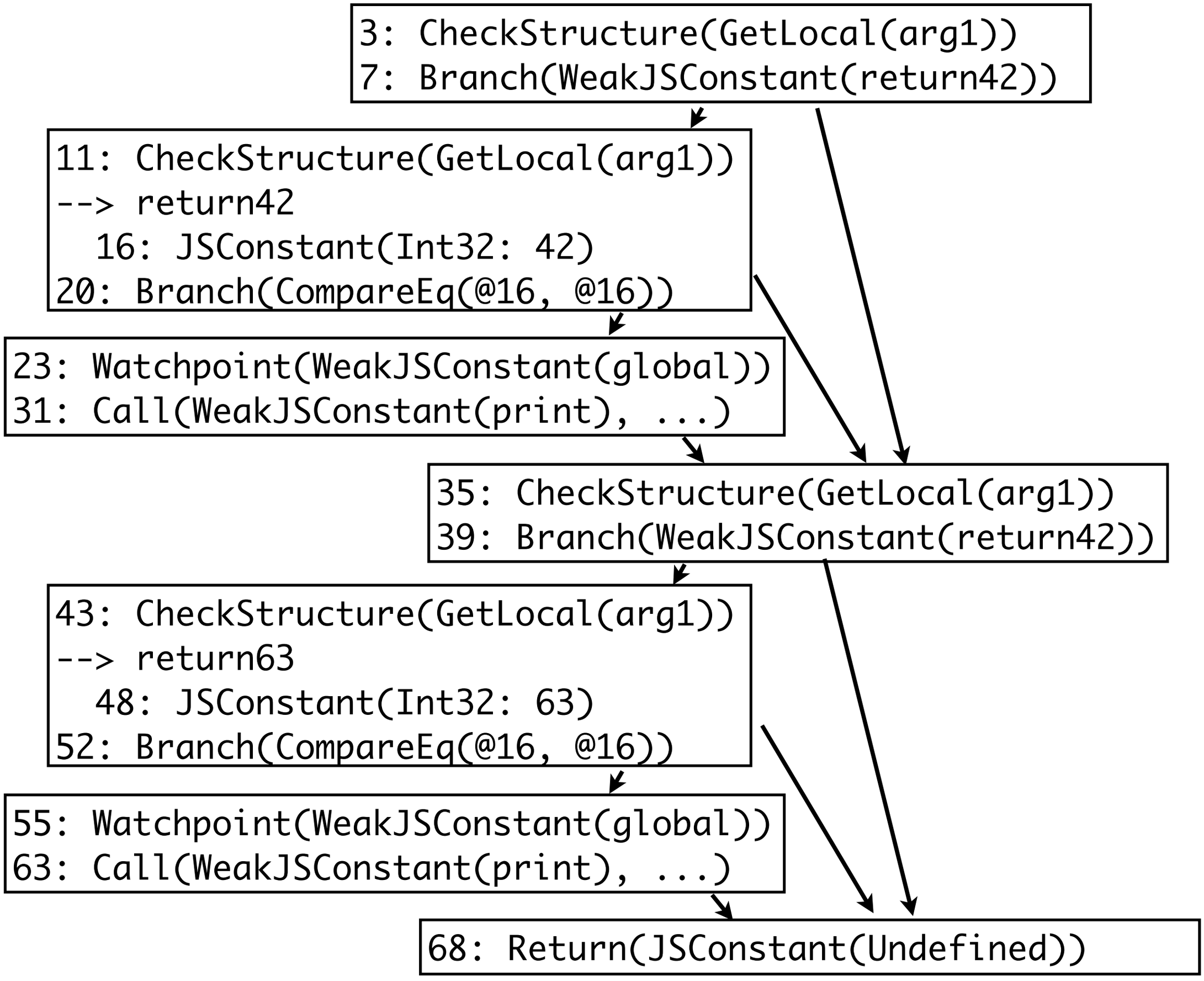
```
CheckStructure(@41<Final>, struct(0x108aec560))
            0x2ffd0ae01c78: mov $0x108aec560, %r11
            0x2ffd0ae01c82: cmp %r11, (%rax)
            0x2ffd0ae01c85: jnz 0x2ffd0ae01dee

15: GetByOffset(@41<Final>, JS, id3{x}, 2)
               0x2ffd0ae01cc9: mov 0x10(%rax), %rbx

20: ArithMul(d@15<Double>, d@15<Double>)
```

- Untyped languages are cool
- We optimized one of them
- Now it runs faster

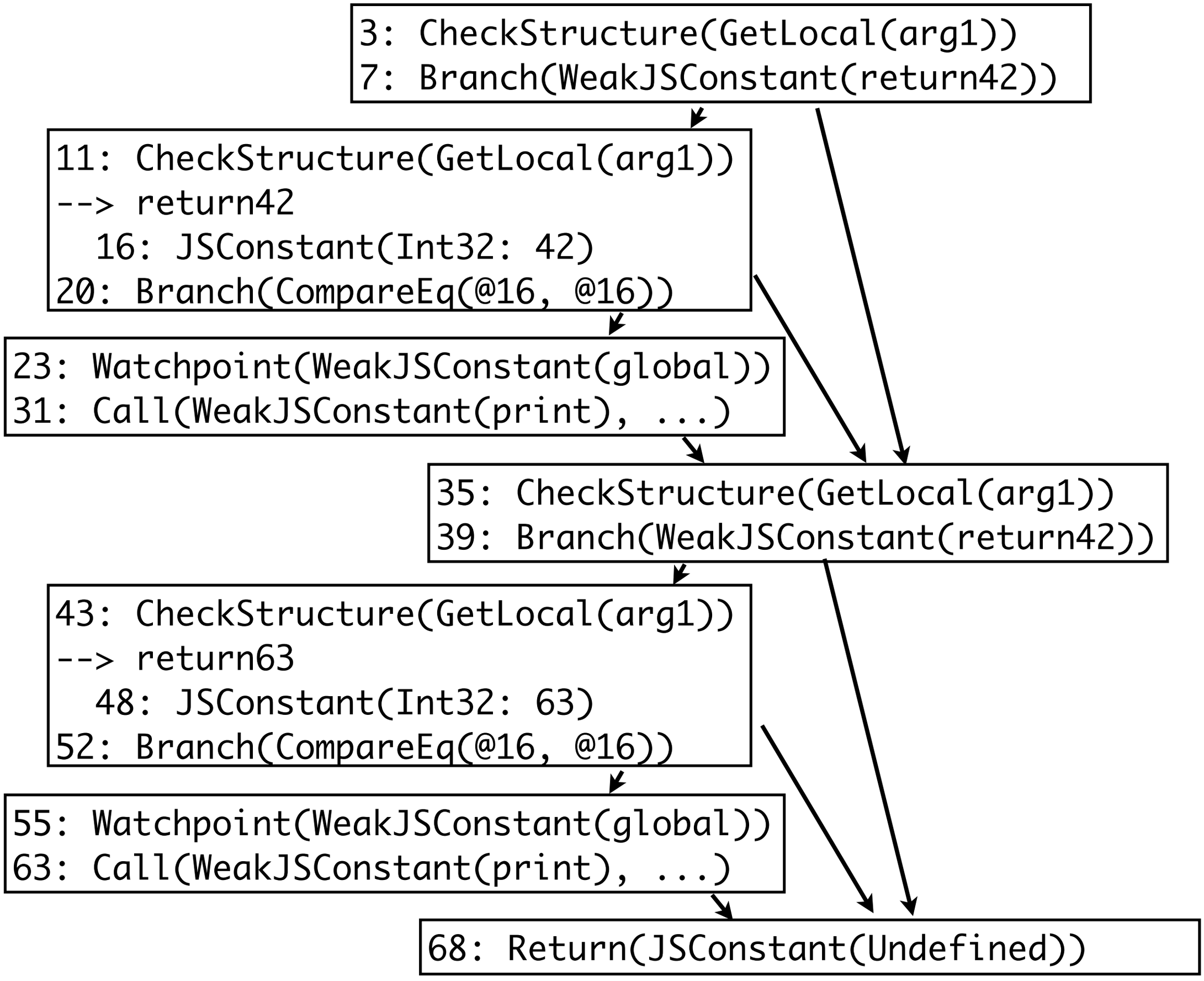
```
function foo(o) {
    if (o.f && o.f() == 42)
        print("hello");
    if (o.g && o.g() == 63)
        print("bye");
}
```

```
function return42() { return 42; }
function return63() { return 63; }

for (var i = 0; i < 2000; ++i)
    foo({f:return42, g:return63});
```

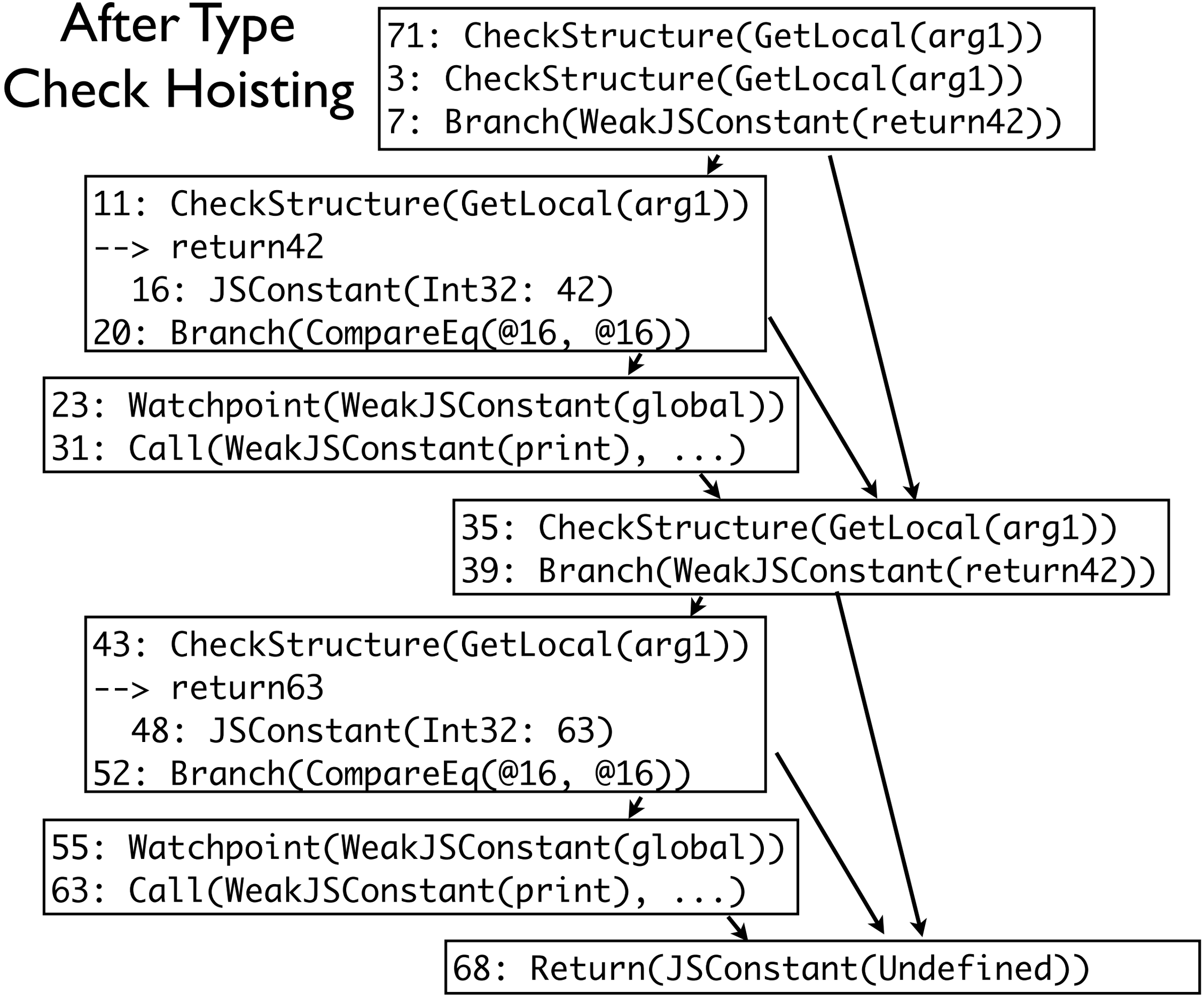
After Bytecode Parsing



After Prediction Propagation



After Type Check Hoisting



After CFA & folding

```
71: CheckStructure(GetLocal(arg1))
3: CheckStructure(GetLocal(arg1))
7: Branch(WeakJSConstant(return42))
```

```
11: CheckStructure(GetLocal(arg1))
--> return42
  16: JSConstant(Int32: 42)
20: Branch(CompareEq(@16, @16))
```

```
23: Watchpoint(WeakJSConstant(global))
31: Call(WeakJSConstant(print), ...)
```

```
35: CheckStructure(GetLocal(arg1))
39: Branch(WeakJSConstant(return42))
```

```
43: CheckStructure(GetLocal(arg1))
--> return63
  48: JSConstant(Int32: 63)
52: Branch(CompareEq(@16, @16))
```

```
55: Watchpoint(WeakJSConstant(global))
63: Call(WeakJSConstant(print), ...)
```

```
68: Return(JSConstant(Undefined))
```

After CFA & folding

```
71: CheckStructure(GetLocal(arg1))
~~3: CheckStructure(GetLocal(arg1))~~
7: Branch(WeakJSConstant(return42))
```

```
~~11: CheckStructure(GetLocal(arg1))~~
--> return42
  16: JSConstant(Int32: 42)
20: Branch(CompareEq(@16, @16))
```

```
23: Watchpoint(WeakJSConstant(global))
31: Call(WeakJSConstant(print), ...)
```

```
35: CheckStructure(GetLocal(arg1))
39: Branch(WeakJSConstant(return42))
```

```
~~43: CheckStructure(GetLocal(arg1))~~
--> return63
  48: JSConstant(Int32: 63)
52: Branch(CompareEq(@16, @16))
```

```
55: Watchpoint(WeakJSConstant(global))
63: Call(WeakJSConstant(print), ...)
```

```
68: Return(JSConstant(Undefined))
```

After CFA & folding

```
71: CheckStructure(GetLocal(arg1))
~~3: CheckStructure(GetLocal(arg1))~~
7: Branch(WeakJSConstant(return42))
```

```
~~11: CheckStructure(GetLocal(arg1))~~
--> return42
  16: JSConstant(Int32: 42)
20: Branch(CompareEq(@16, @16))
```

```
23: Watchpoint(WeakJSConstant(global))
31: Call(WeakJSConstant(print), ...)
```

```
35: Watchpoint(GetLocal(arg1))
39: Branch(WeakJSConstant(return42))
```

```
~~43: CheckStructure(GetLocal(arg1))~~
--> return63
  48: JSConstant(Int32: 63)
52: Branch(CompareEq(@16, @16))
```

```
55: Watchpoint(WeakJSConstant(global))
63: Call(WeakJSConstant(print), ...)
```

```
68: Return(JSConstant(Undefined))
```

After CFA & folding

```
71: CheckStructure(GetLocal(arg1))
~~3: CheckStructure(GetLocal(arg1))~~
7: Branch(WeakJSConstant(return42))
```

```
~~11: CheckStructure(GetLocal(arg1))~~
~~--> return42~~
~~  16: JSConstant(Int32: 42)~~
20: Branch(JSConstant(True))
```

```
23: Watchpoint(WeakJSConstant(global))
31: Call(WeakJSConstant(print), ...)
```

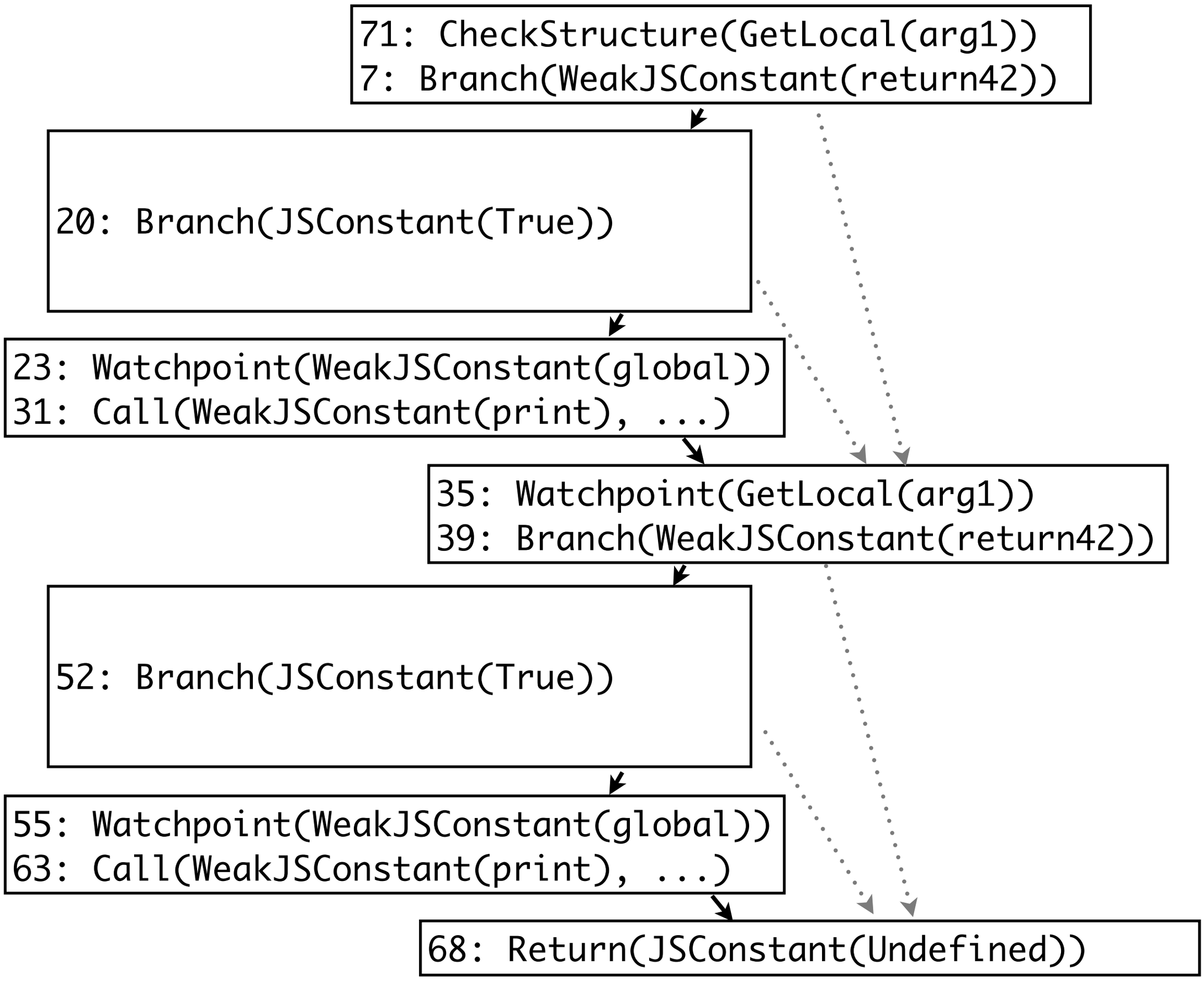
```
35: Watchpoint(GetLocal(arg1))
39: Branch(WeakJSConstant(return42))
```

```
~~43: CheckStructure(GetLocal(arg1))~~
~~--> return63~~
~~  48: JSConstant(Int32: 63)~~
52: Branch(JSConstant(True))
```

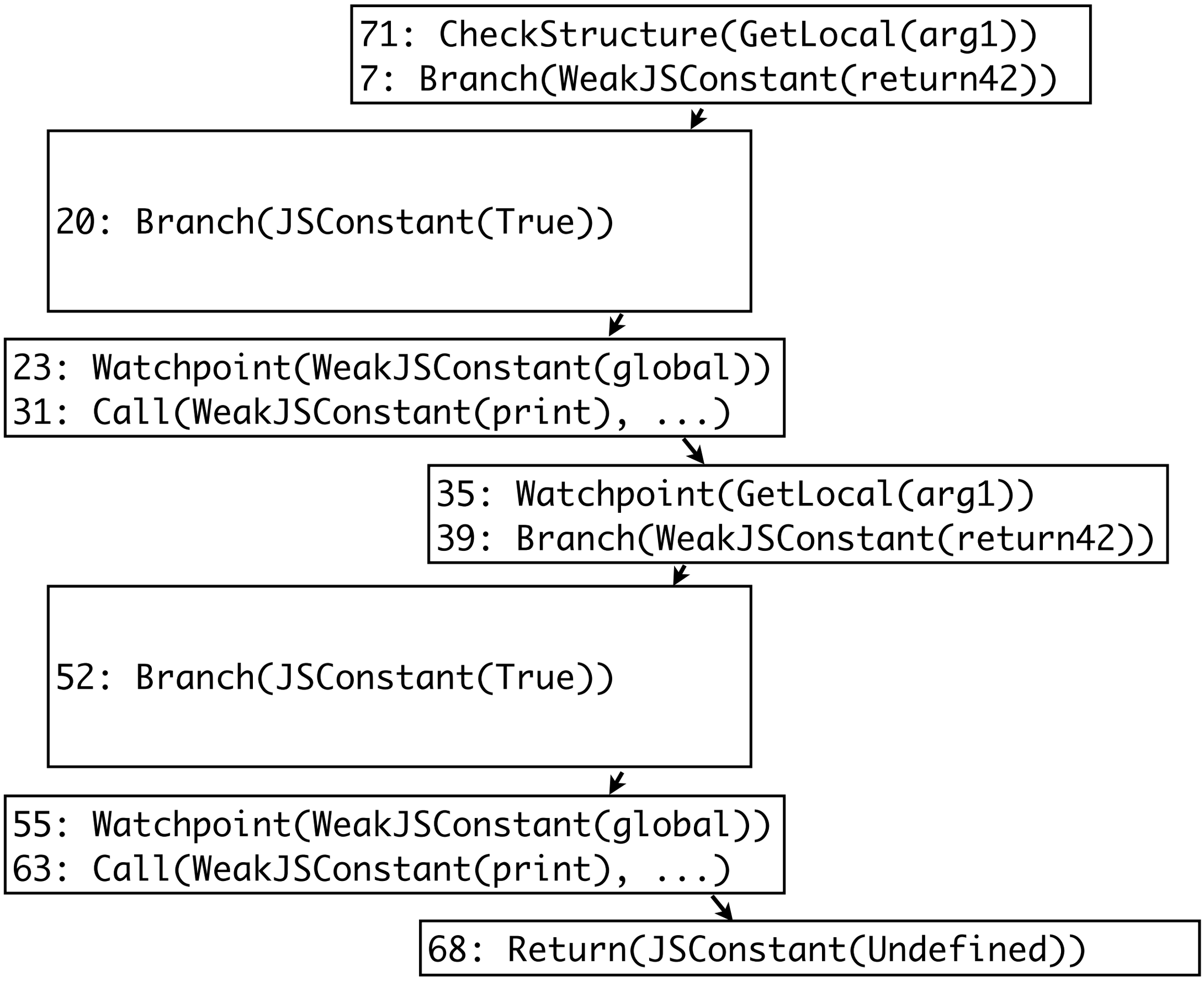
```
55: Watchpoint(WeakJSConstant(global))
63: Call(WeakJSConstant(print), ...)
```

```
68: Return(JSConstant(Undefined))
```

After CFA & folding



After CFG simplify



After CFG simplify

```
71: CheckStructure(GetLocal(arg1))
23: Watchpoint(WeakJSConstant(global))
31: Call(WeakJSConstant(print), ...)
35: Watchpoint(GetLocal(arg1))
55: Watchpoint(WeakJSConstant(global))
63: Call(WeakJSConstant(print), ...)
68: Return(JSConstant(Undefined))
```