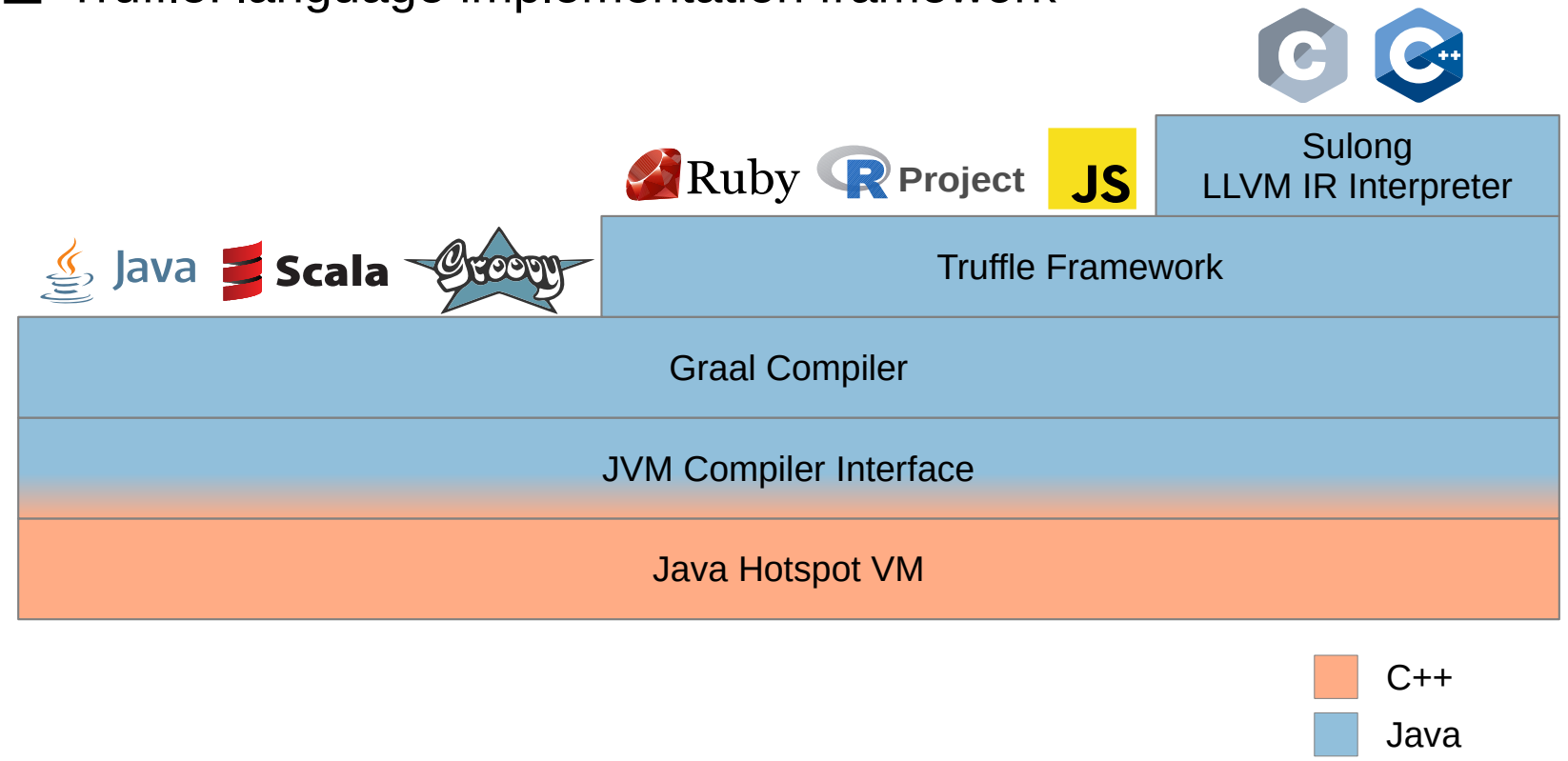
Benchmarking Partial Evaluation in Truffle

Florian Latifi
Virtual Machine Meetup
14th September 2018

JOHANNES KEPLER
UNIVERSITY LINZ

Graal and Truffle [1]

- Graal: dynamic compiler written in Java
- Truffle: language implementation framework



Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  if (2 == 0) {
    1
  } else {
    x * pow(x, 2 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  if (false) {
    1
  } else {
    x * pow(x, 2 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * pow(x, 2 - 1)
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * pow(x, 1)
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x *
  if (1 == 0) {
    1
  } else {
    x * pow(x, 1 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x *
  if (false) {
    1
  } else {
    x * pow(x, 1 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x * pow(x, 1 - 1)
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x * pow(x, 0)
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x *
  if (0 == 0) {
    1
  } else {
    x * pow(x, 0 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x *
  if (true) {
    1
  } else {
    x * pow(x, 0 - 1)
  }
}
```

Partial Evaluation

- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x * 1
}
```

Partial Evaluation

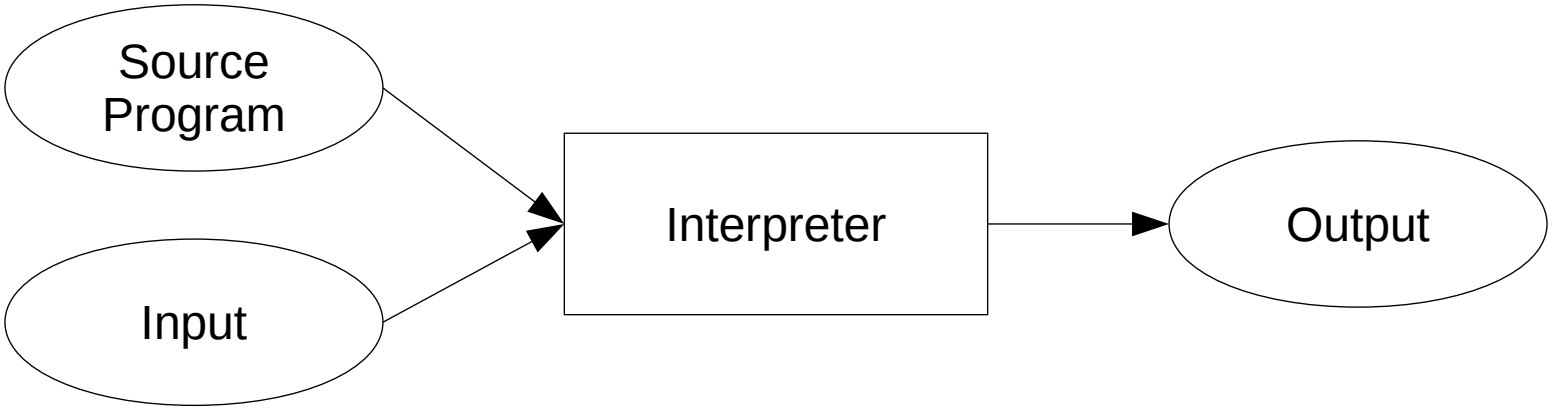
- Optimization by specialization
- Example:
  - specializing `pow(x, n)`
  - by assuming `n=2`

```
fn pow(x, n) {
  if (n == 0) {
    1
  } else {
    x * pow(x, n - 1)
  }
}
```

```
fn pow_2(x) {
  x * x
}
```

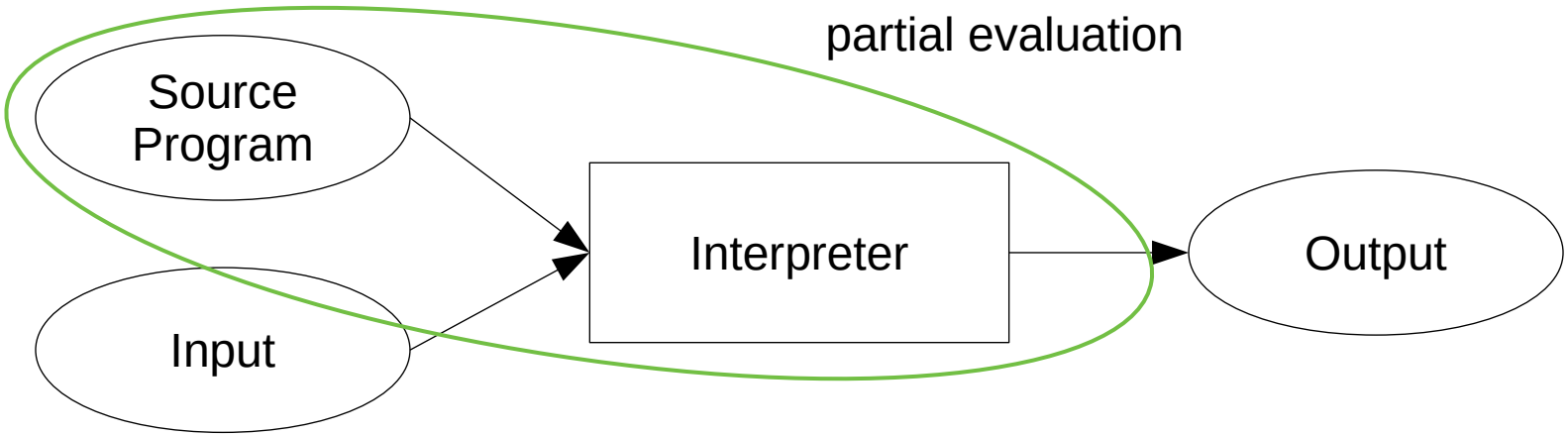
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



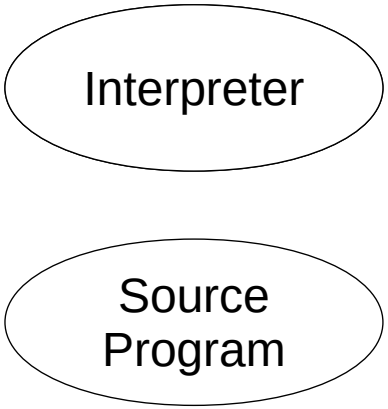
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



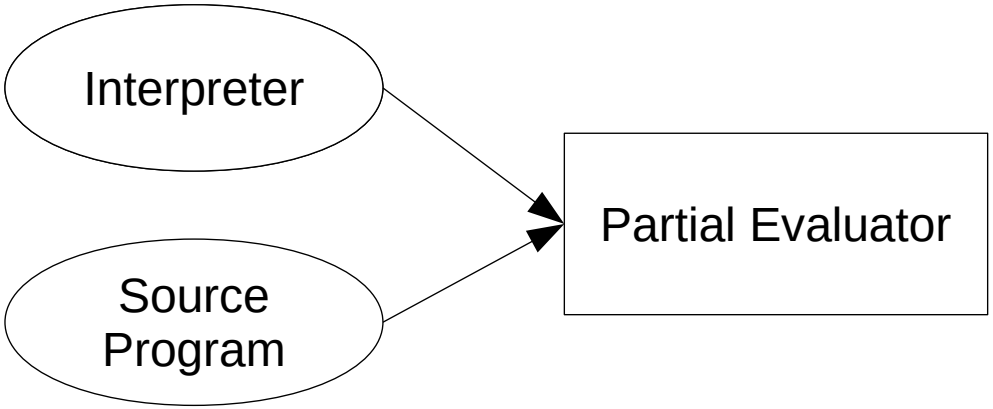
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



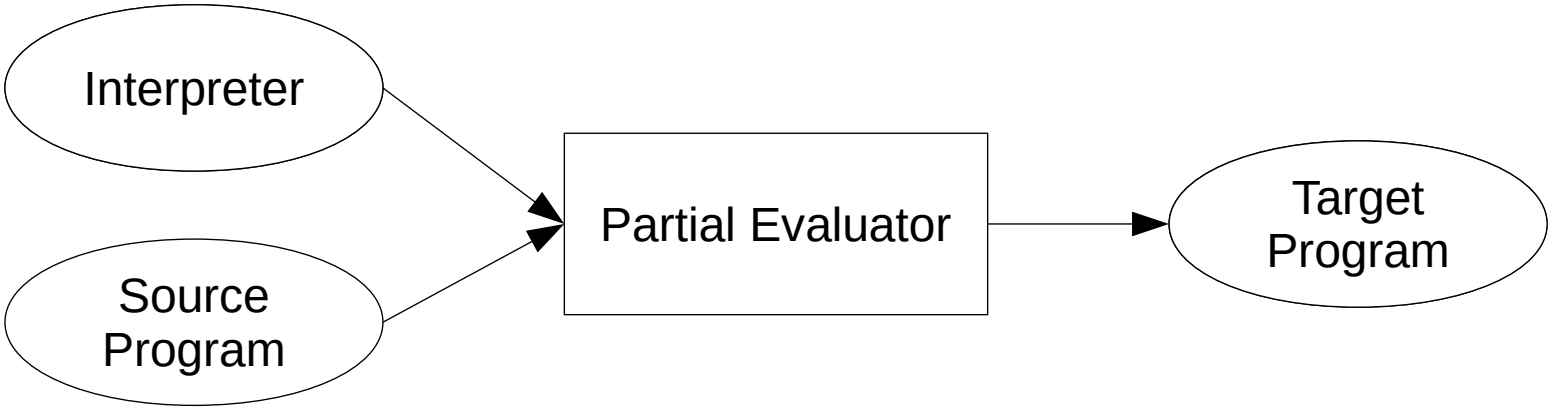
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



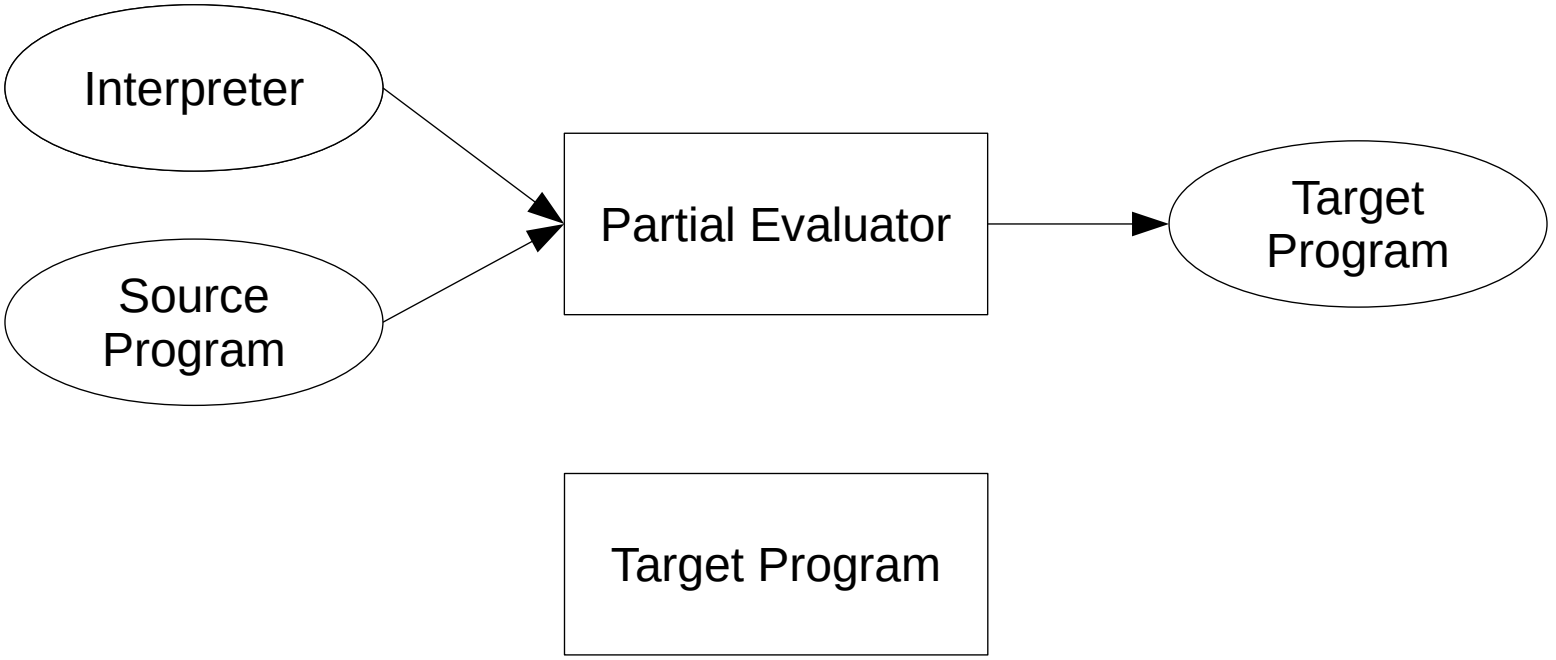
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



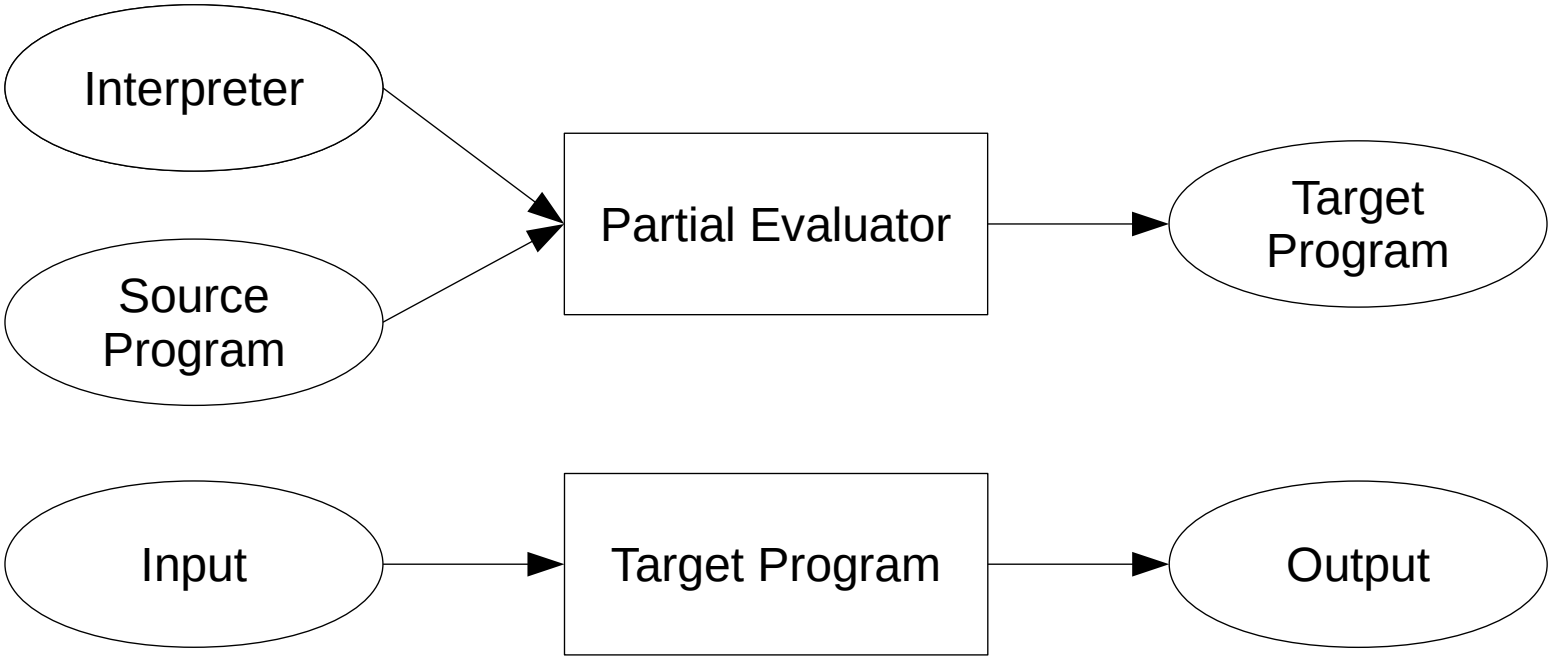
Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



Partial Evaluation in Truffle [1]

- First Futamura projection [2]:
  - specializing an interpreter for a given source program, yielding an executable



Partial Evaluation in Truffle

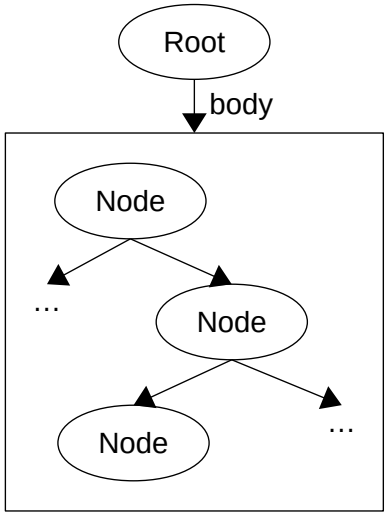
- Example: `foo(n)` and `bar(n)`

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

Partial Evaluation in Truffle

- Example: `foo(n)` and `bar(n)`

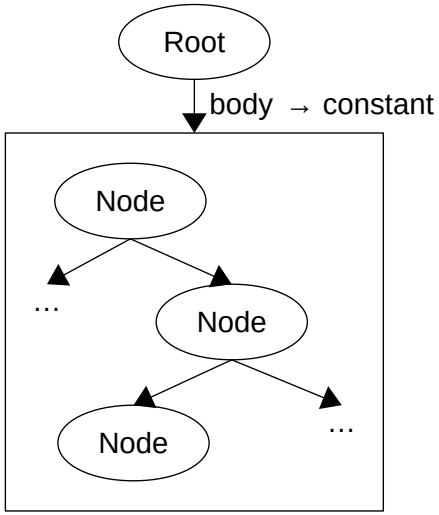
```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```



Partial Evaluation in Truffle

- Example: `foo(n)` and `bar(n)`

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

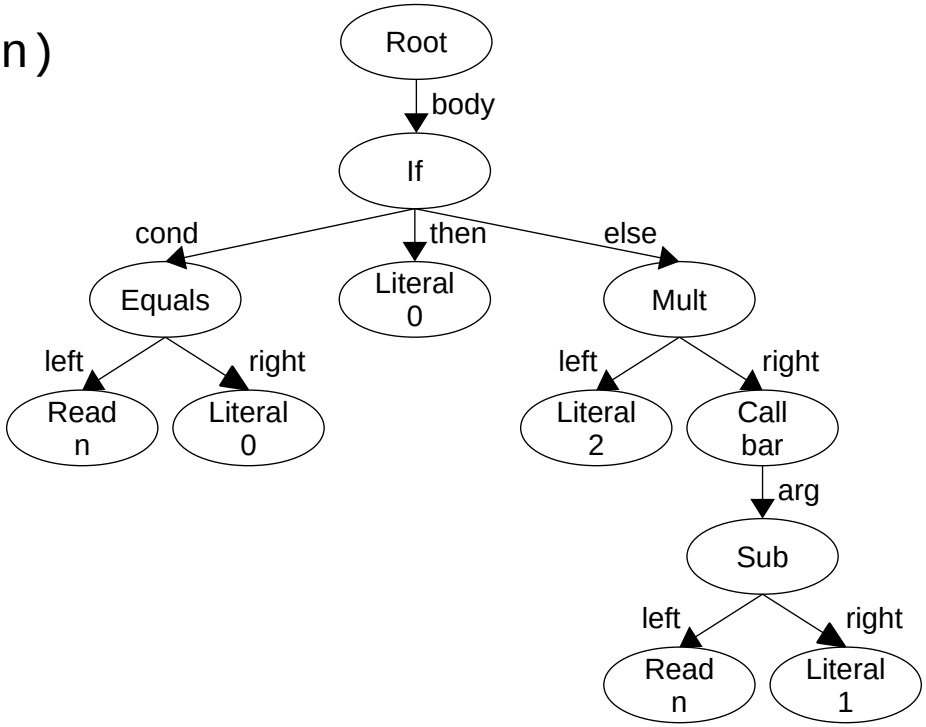


- First Futamura projection:
  - by assuming that AST is constant

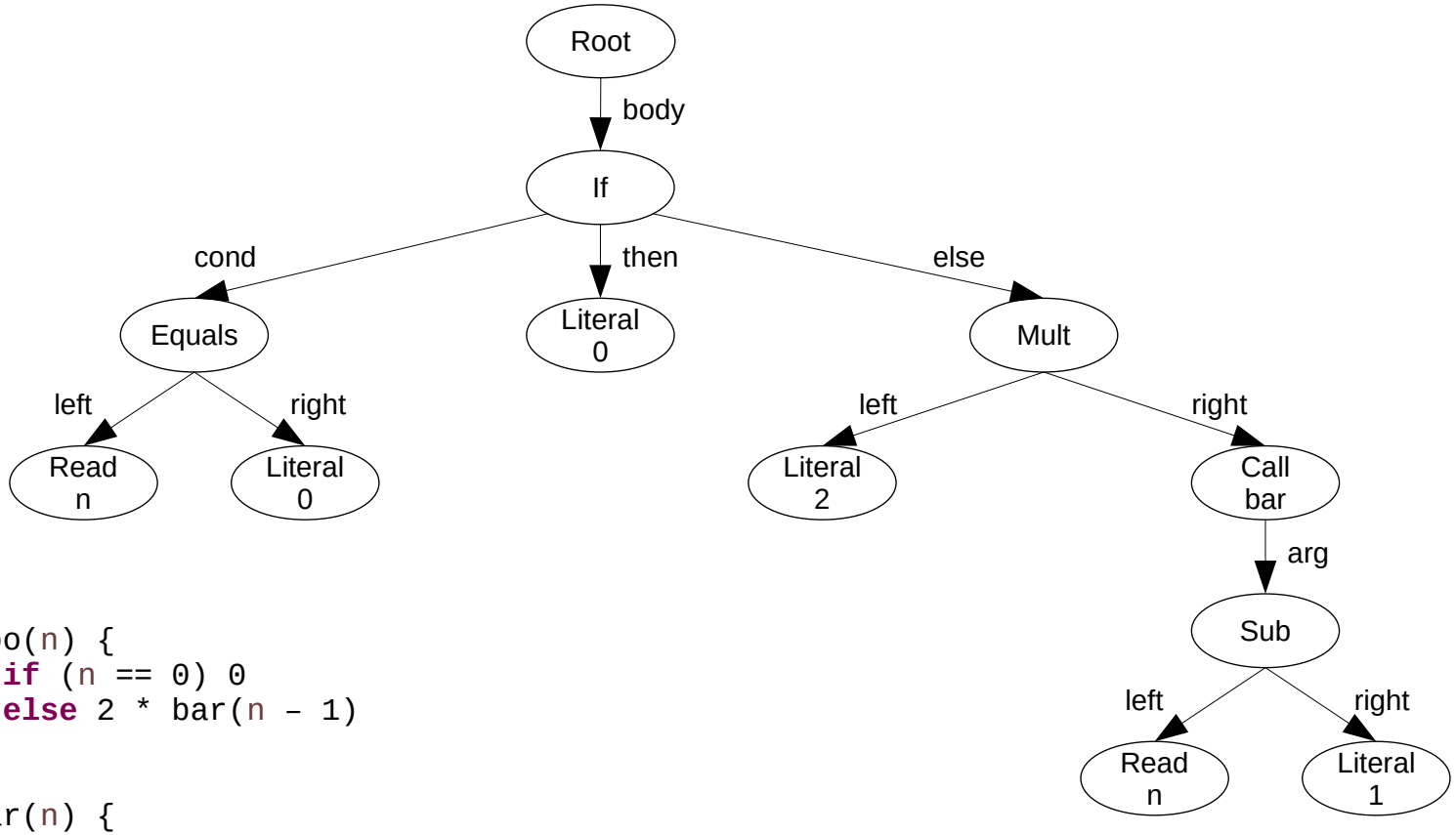
Partial Evaluation in Truffle

- Example: `foo(n)` and `bar(n)`

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

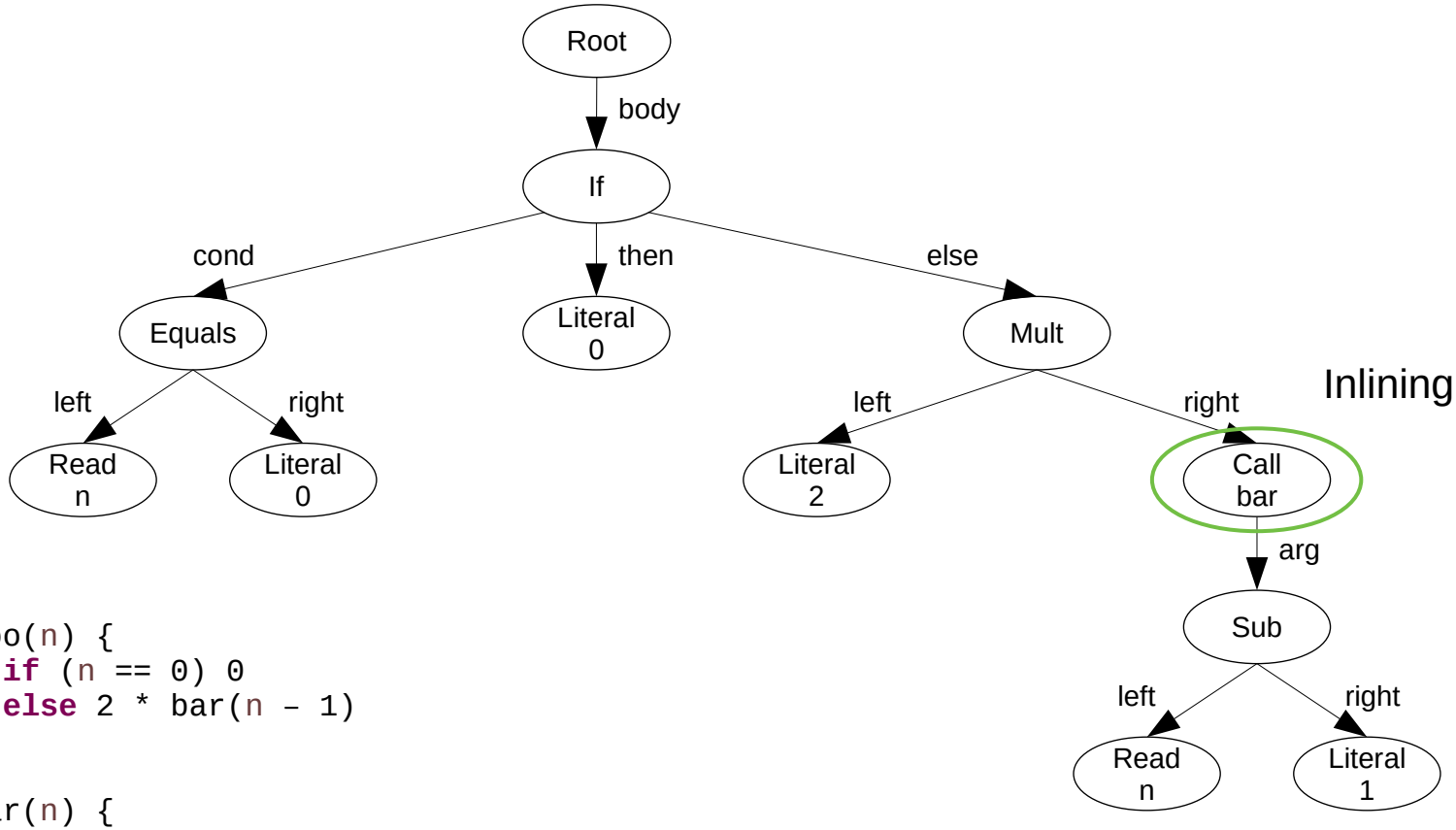


- First Futamura projection:
  - by assuming that AST is constant




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

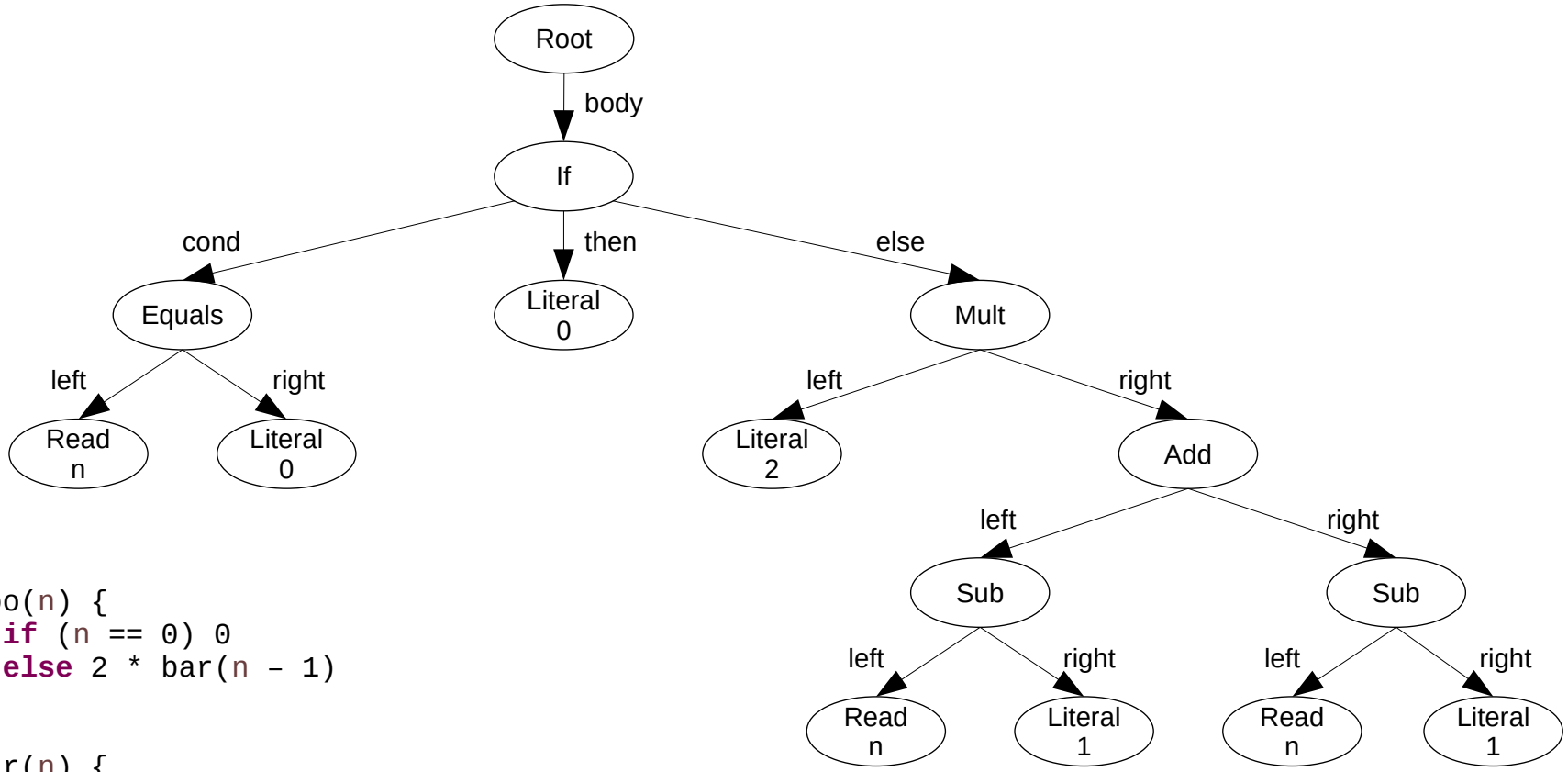
- Counters:
  - Bytecode parsings: 0
  - Call inlinings: 0
  - Simplifications: 0



Inlining on AST level

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

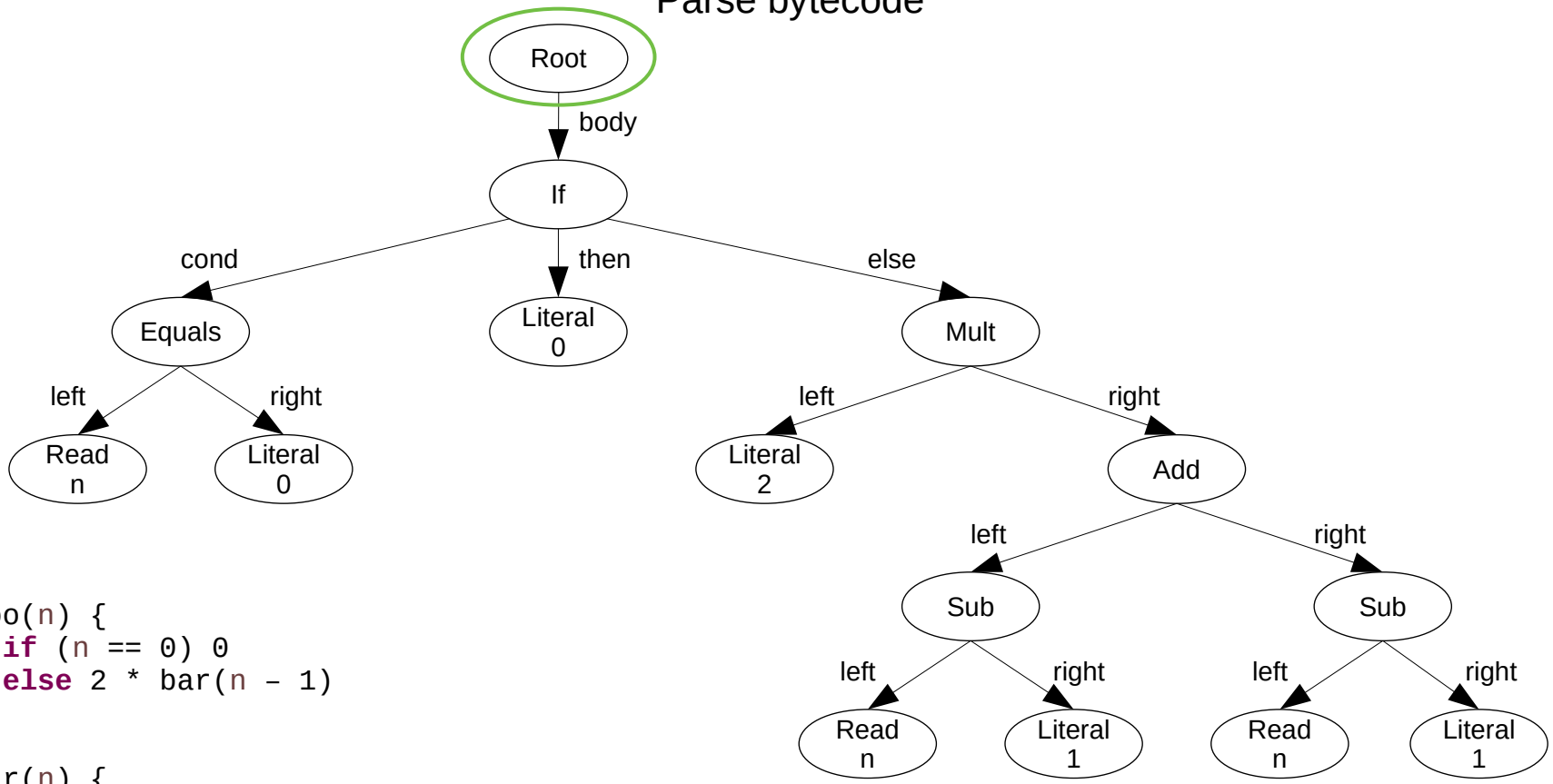
- Counters:
  - Bytecode parsings: 0
  - Call inlinings: 0
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

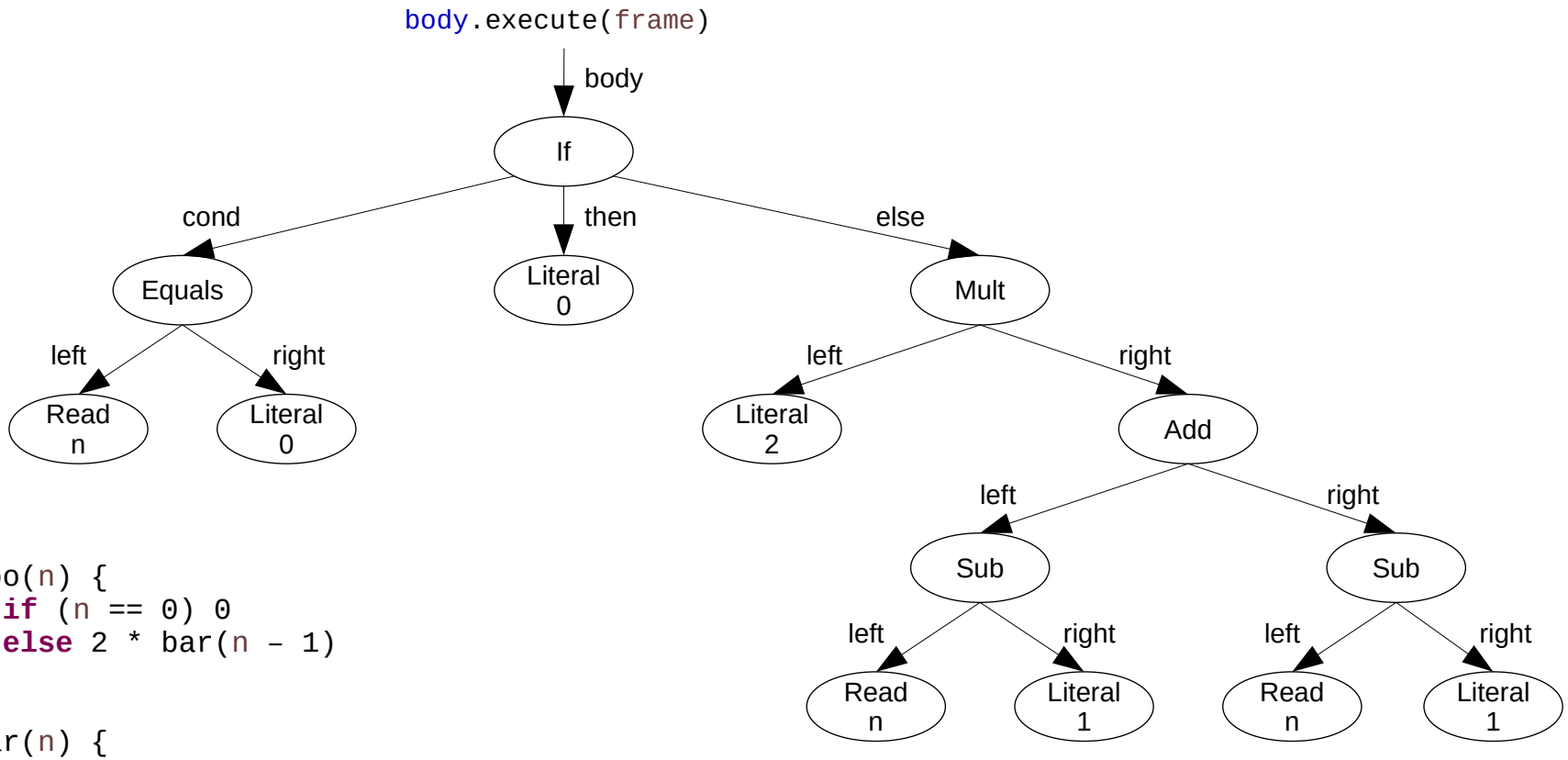
- Counters:
  - Bytecode parsings: 0
  - Call inlinings: 0
  - Simplifications: 0

Parse bytecode




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

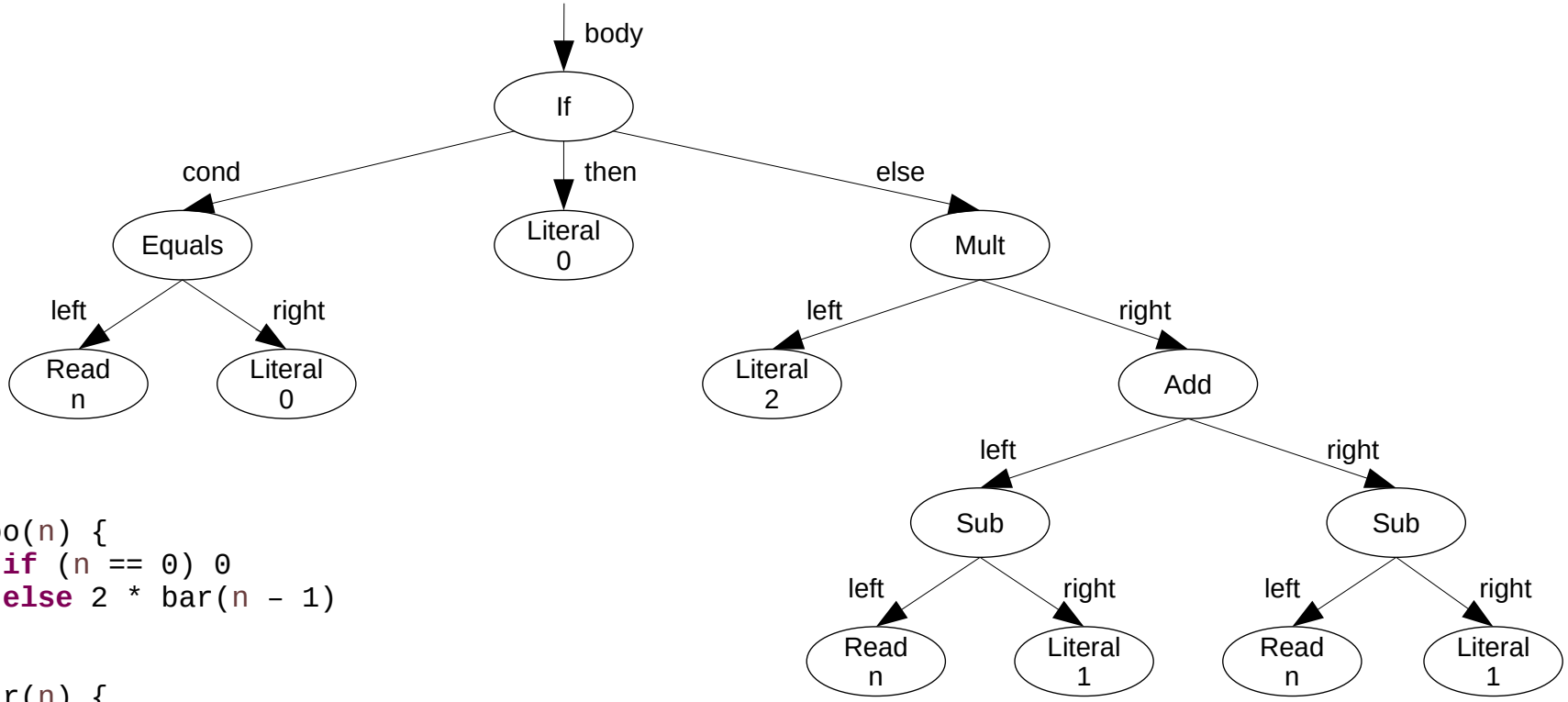
- Counters:
  - Bytecode parsings: 0
  - Call inlinings: 0
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

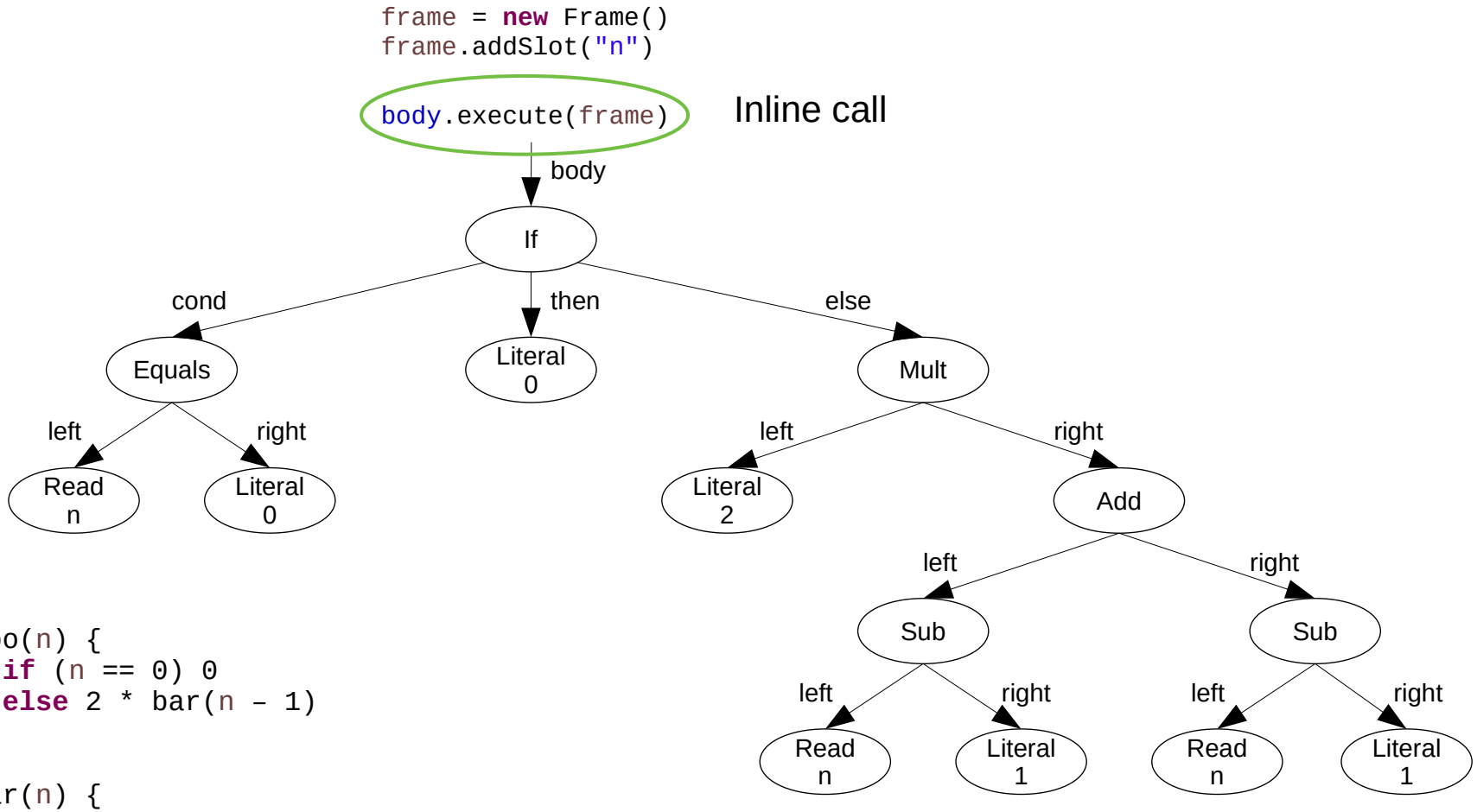
- Counters:
  - Bytecode parsings: 1
  - Call inlinings: 0
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

body.execute(frame)
```



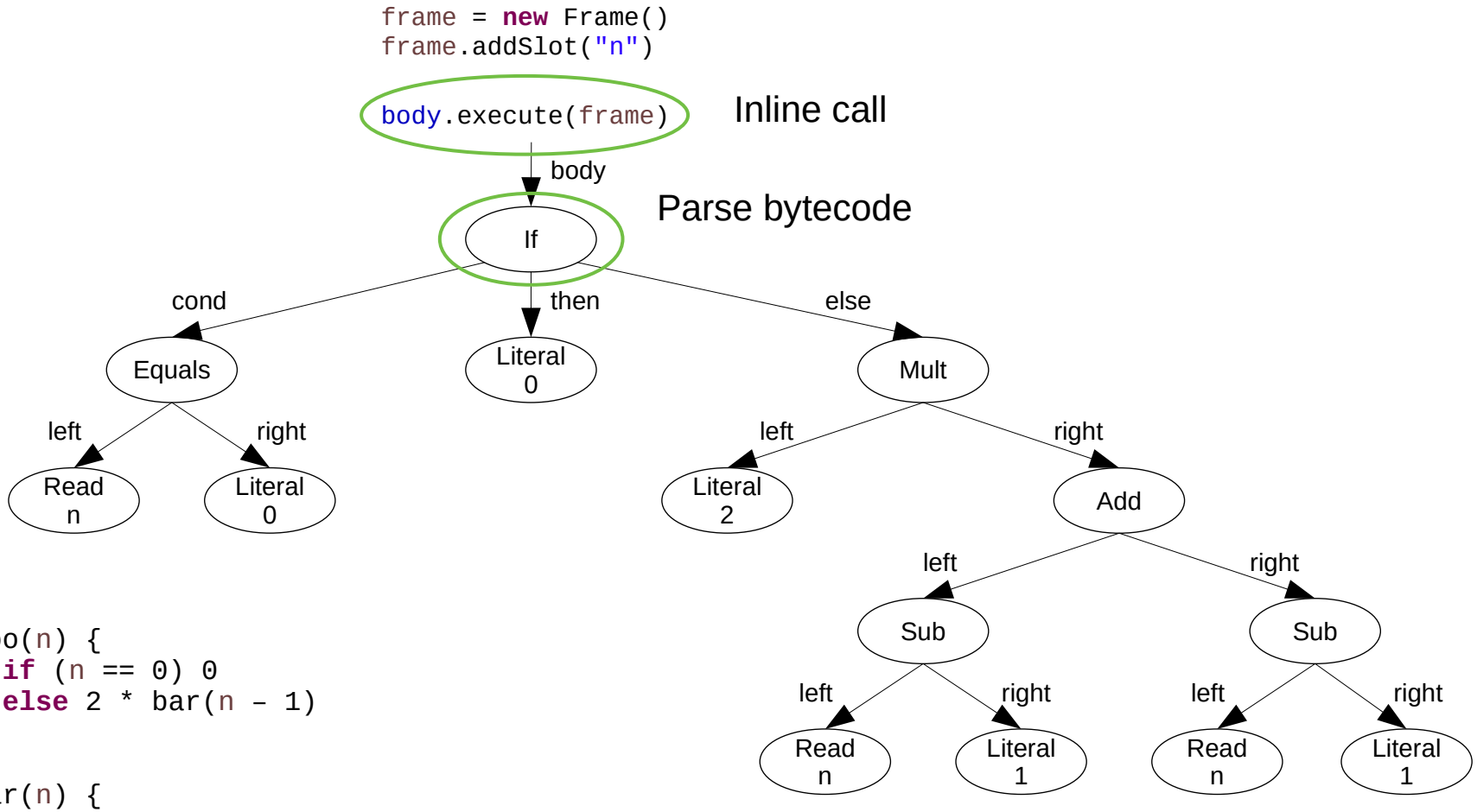

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 1
  - Call inlinings: 0
  - Simplifications: 0



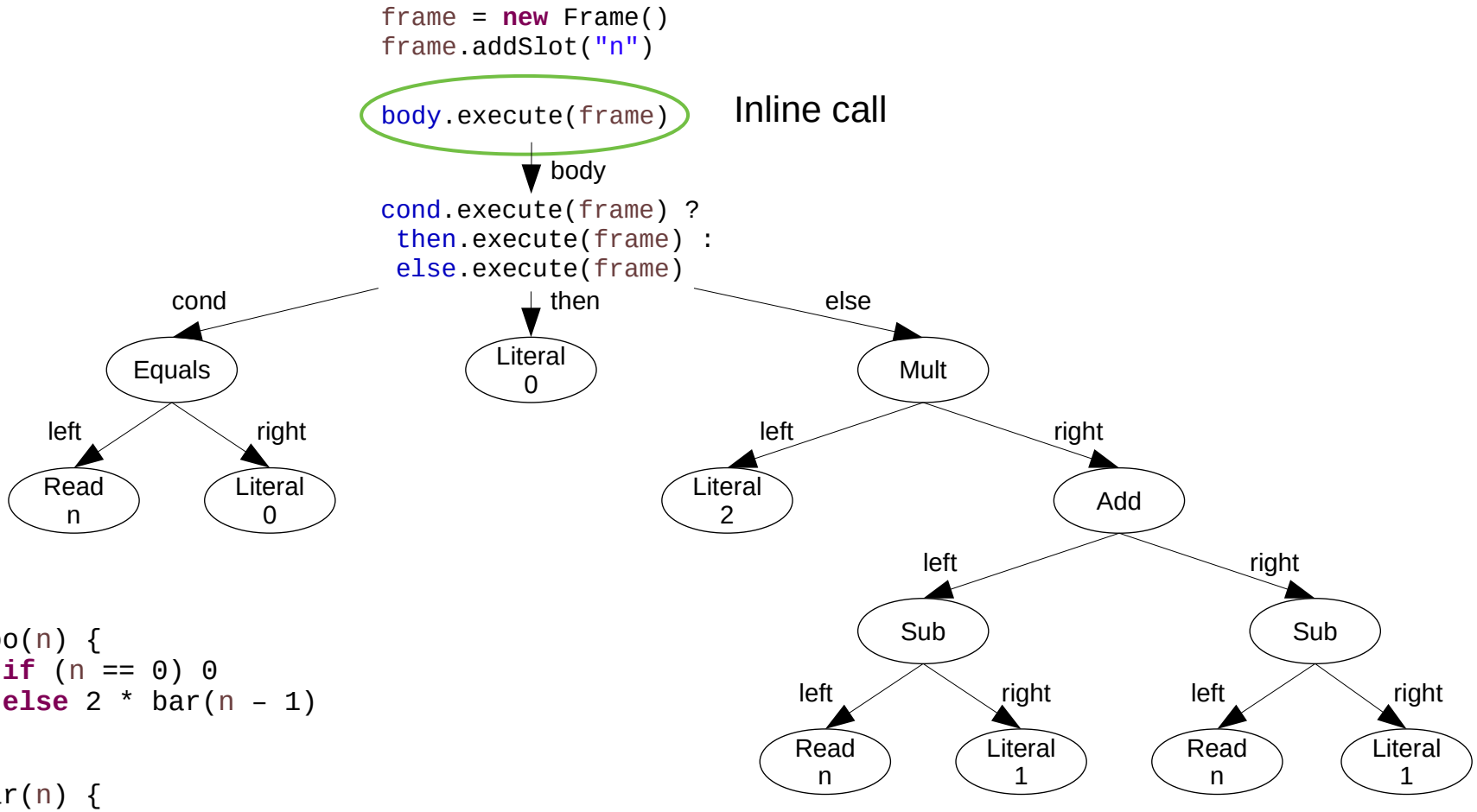

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 1
  - Call inlinings: 0
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

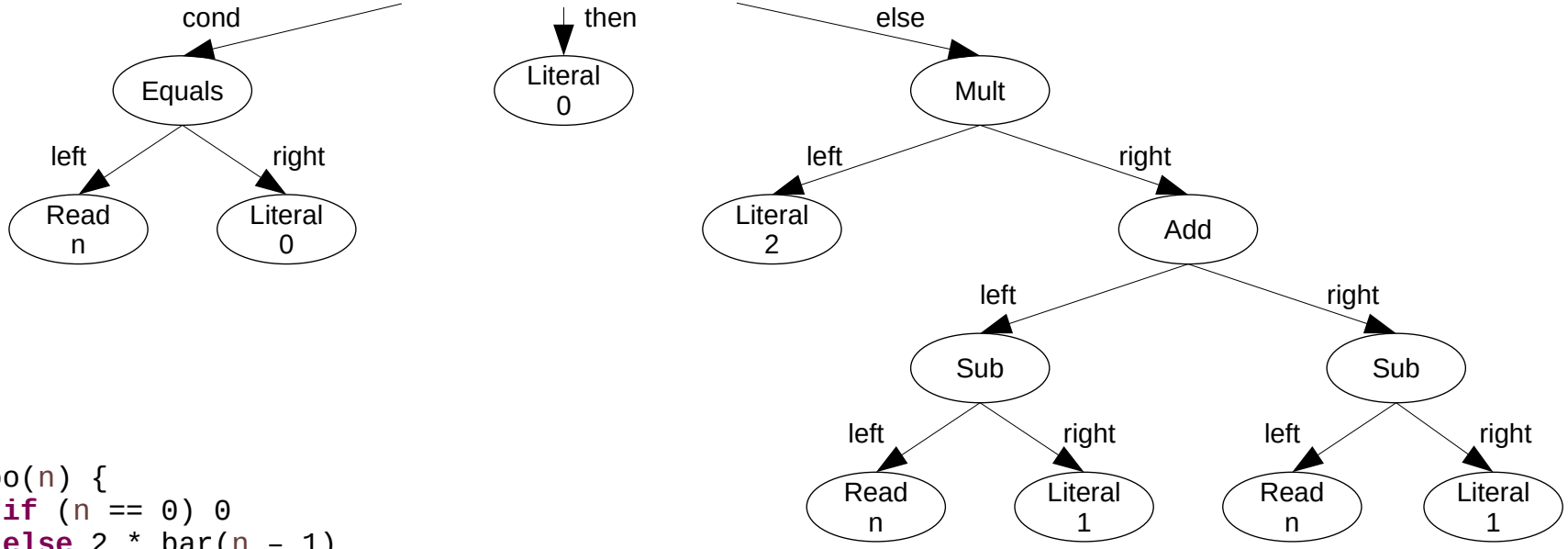
- Counters:
  - Bytecode parsings: 1
  - Call inlinings: 0
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 2
  - Call inlinings: 0
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

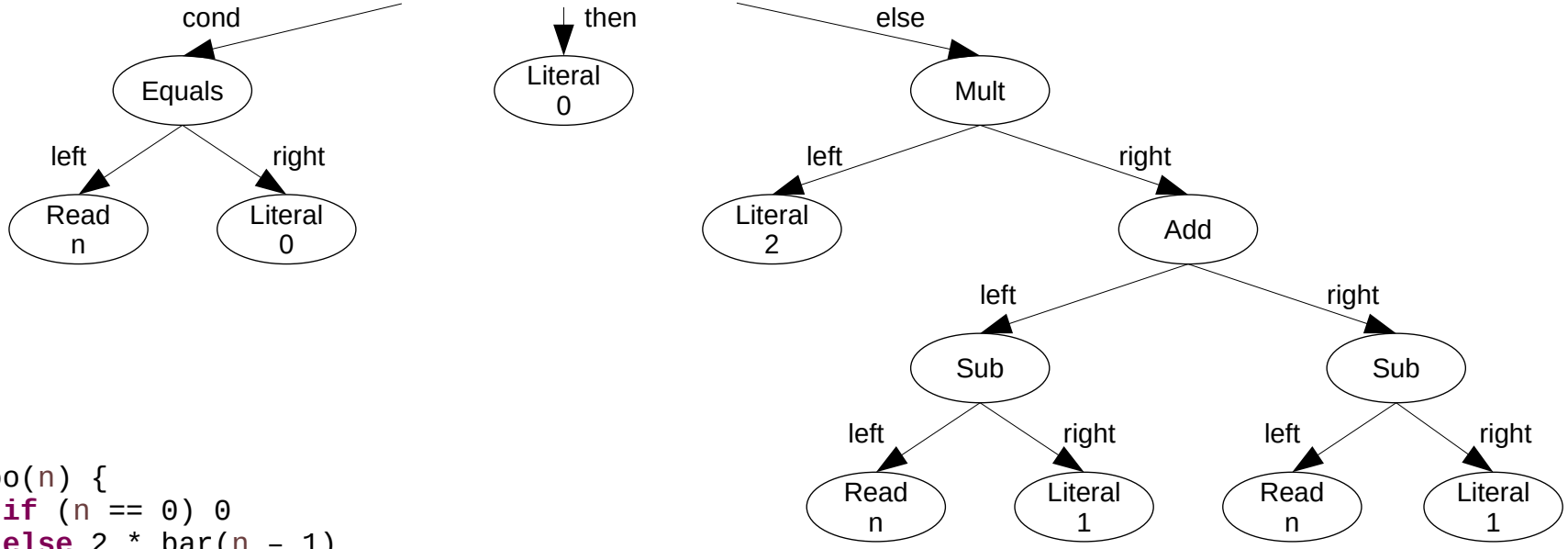



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 2
  - Call inlinings: 1
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

Inline call



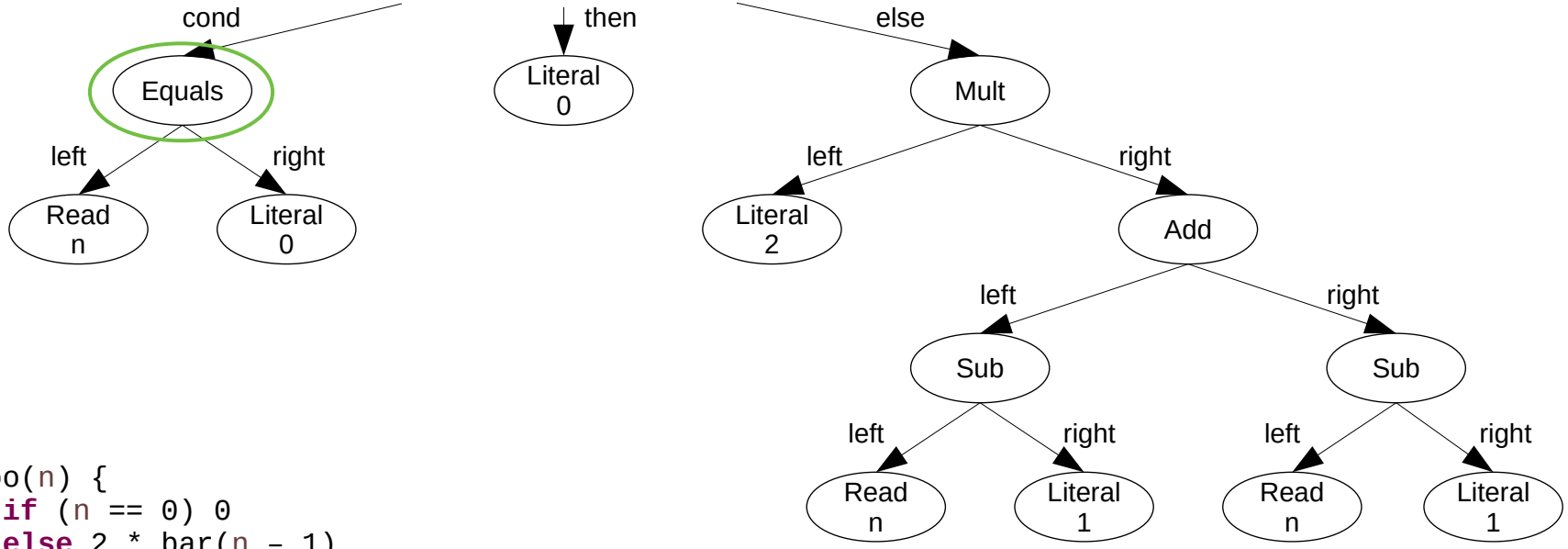

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 2
  - Call inlinings: 1
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

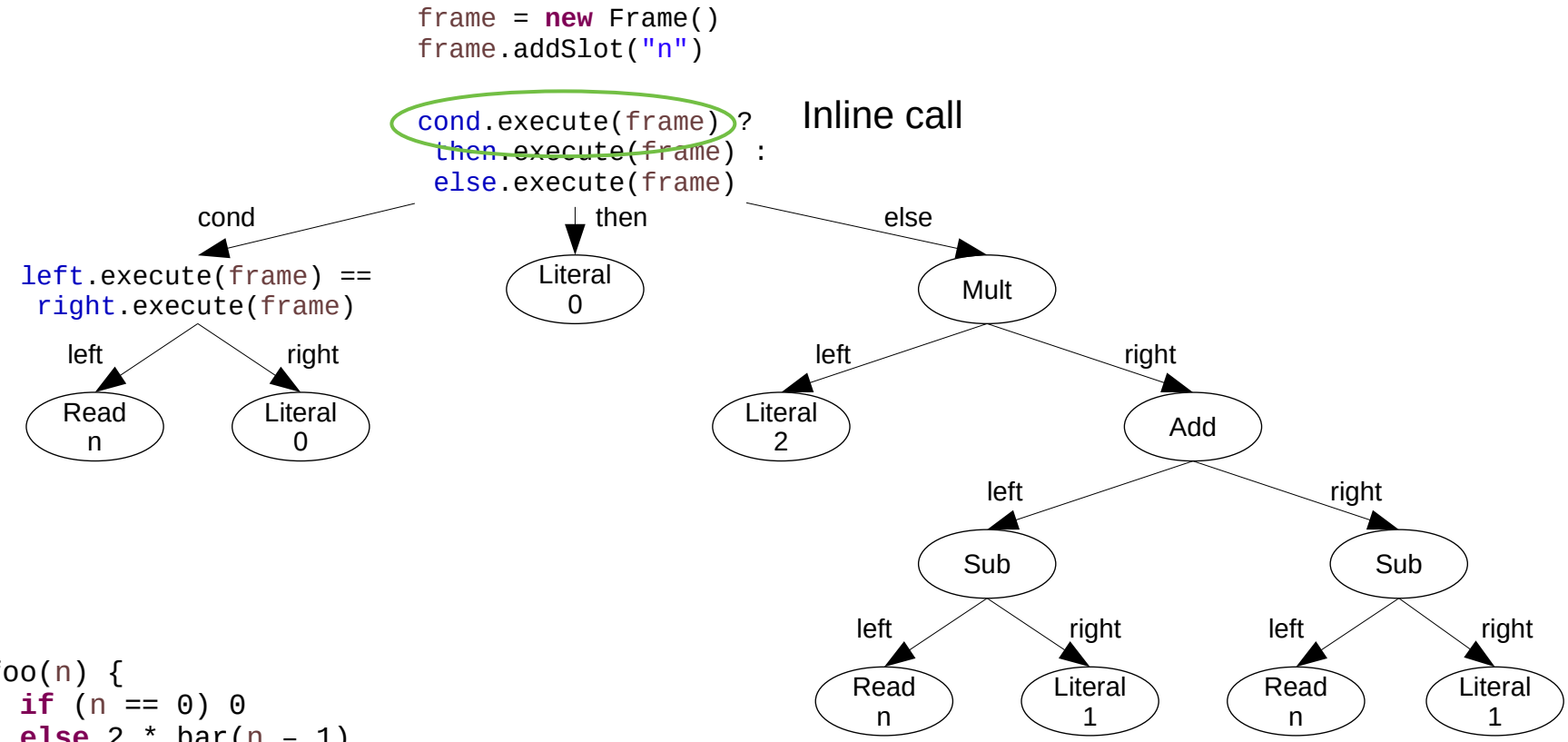
Inline call

Parse bytecode




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

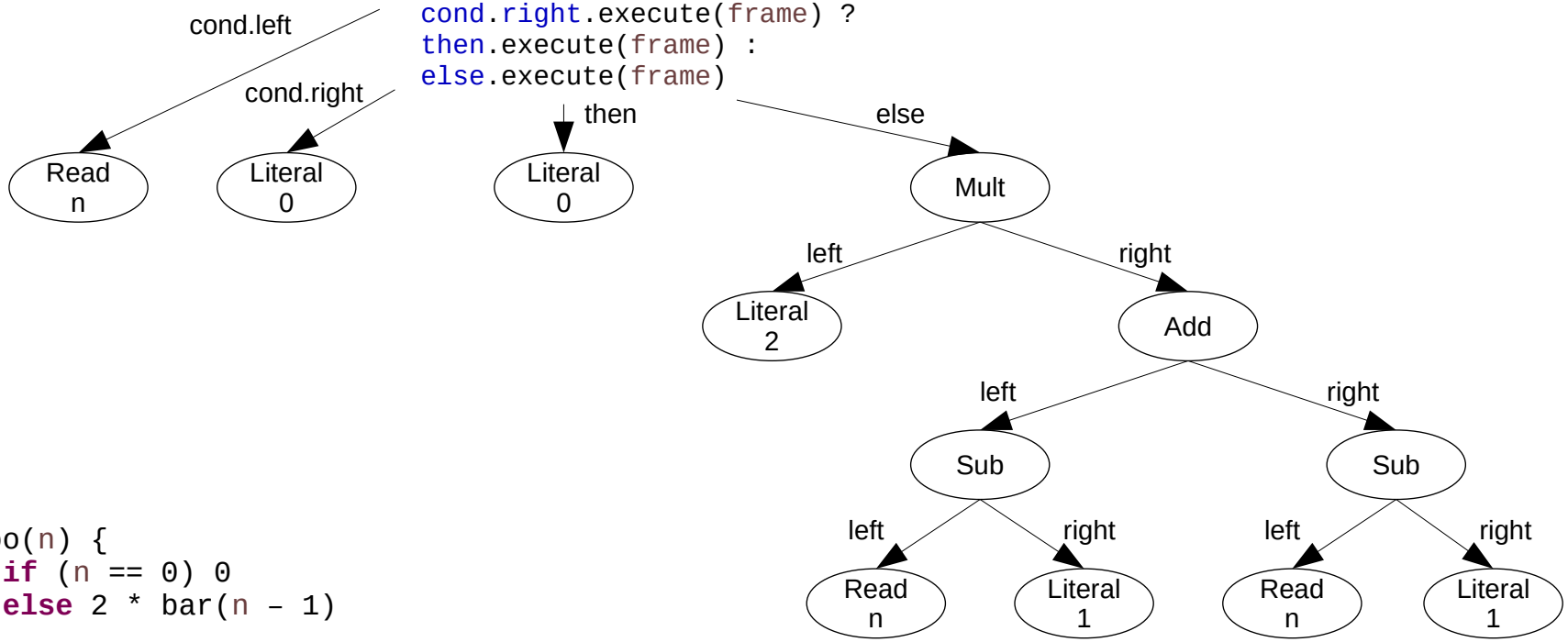
- Counters:
  - Bytecode parsings: 2
  - Call inlinings: 1
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 3
  - Call inlinings: 1
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.left.execute(frame) ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

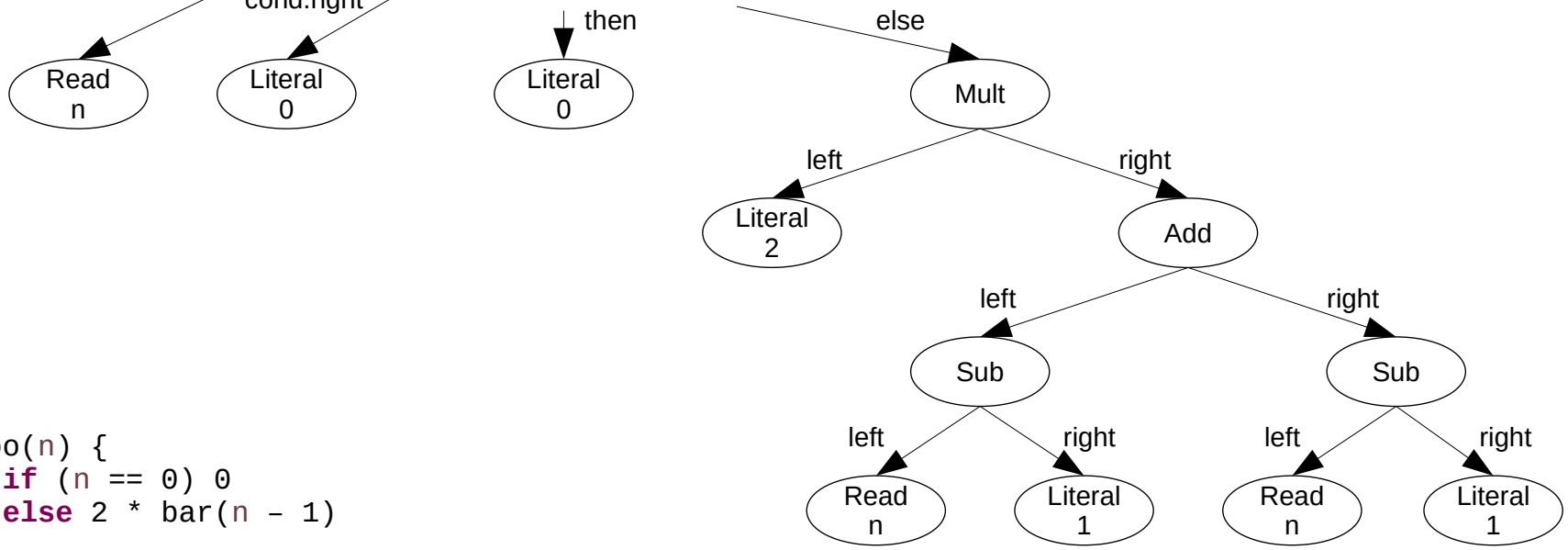



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 3
  - Call inlinings: 2
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.left.execute(frame) ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

Inline call

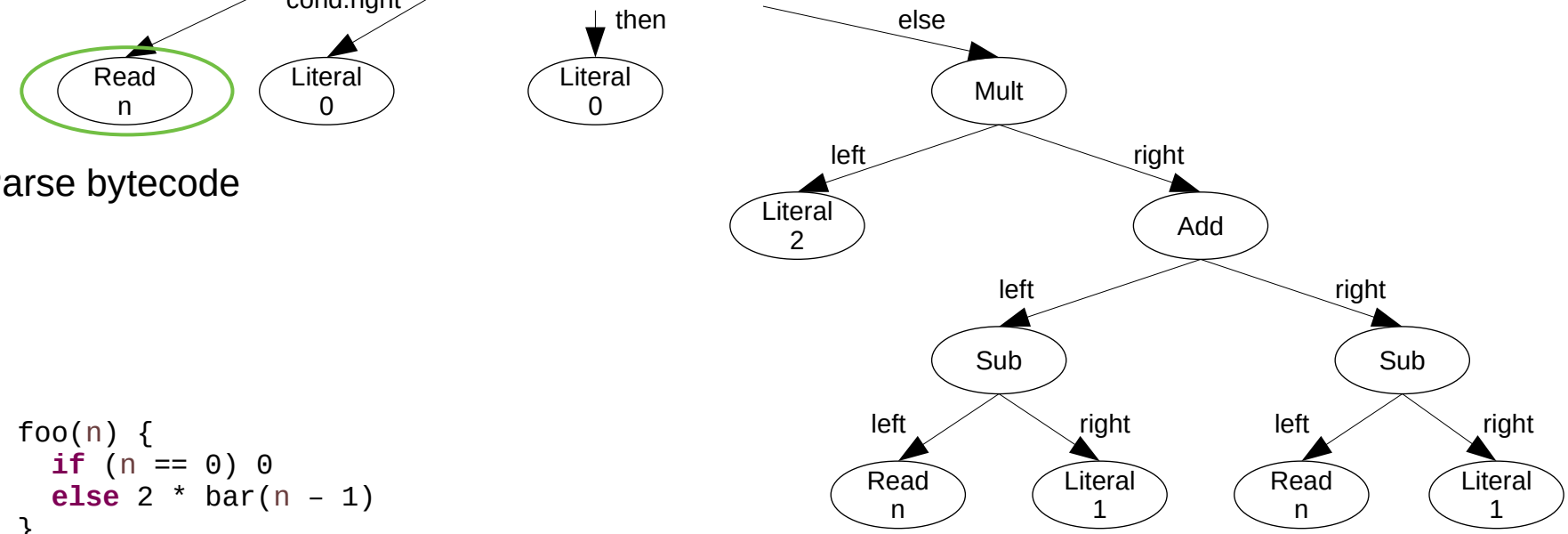



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 3
  - Call inlinings: 2
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

cond.left.execute(frame) ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

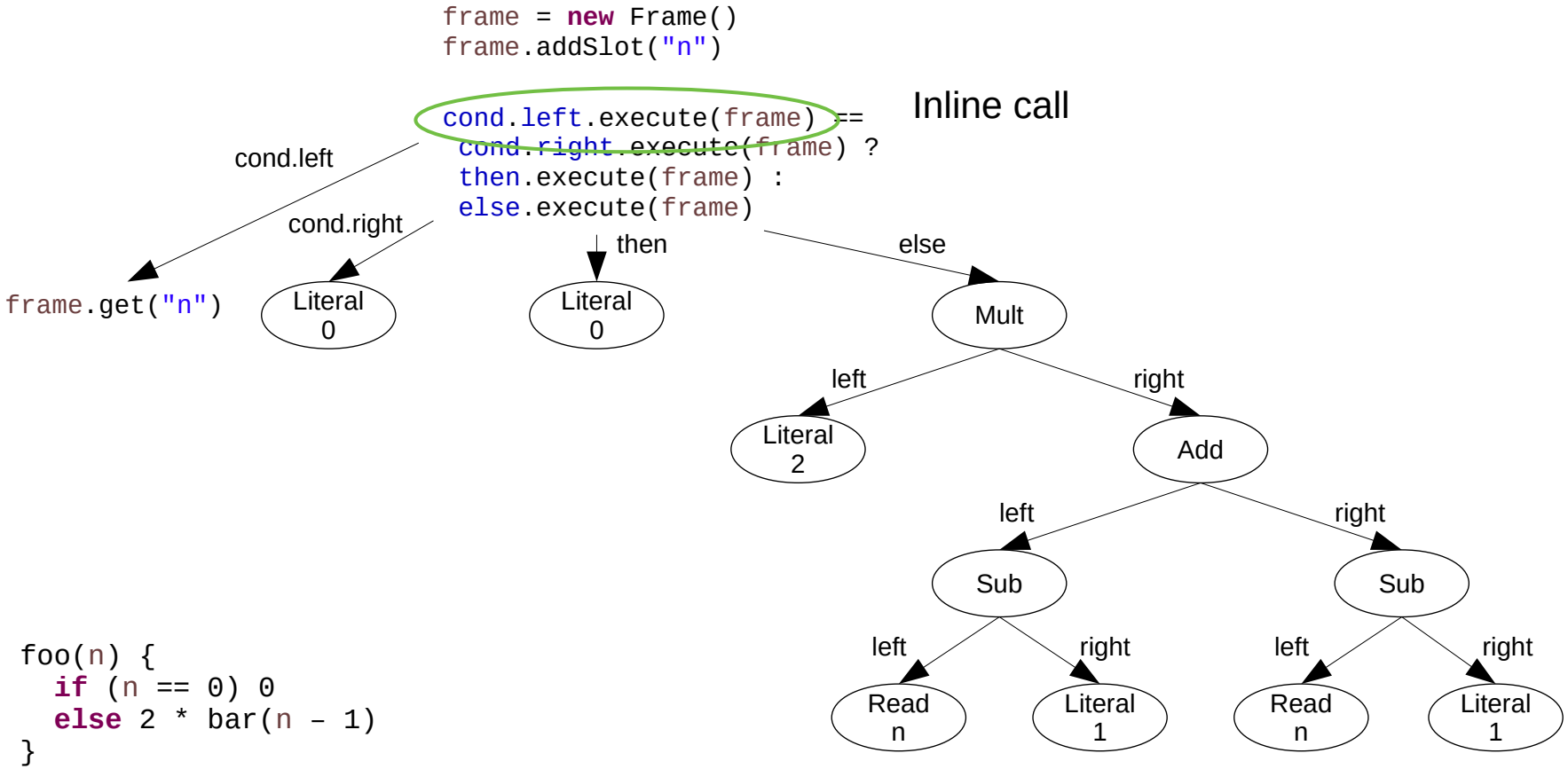
Inline call



Parse bytecode

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

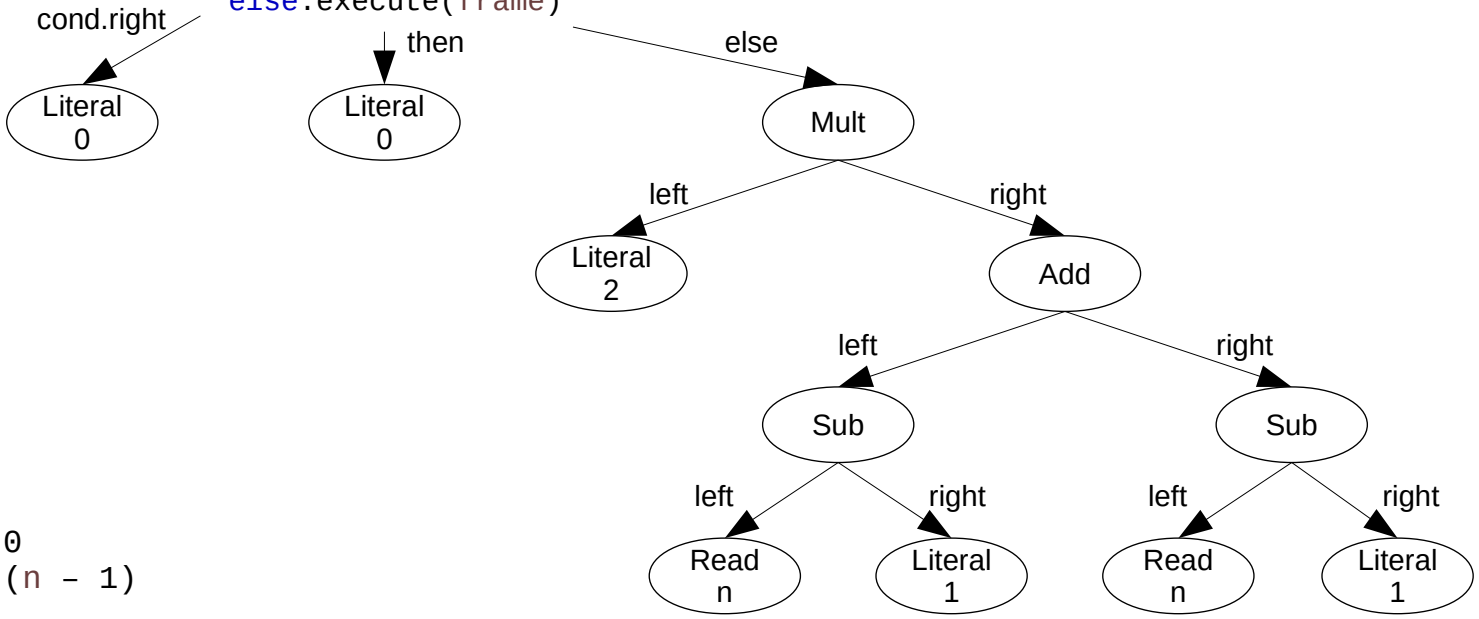
- Counters:
  - Bytecode parsings: 3
  - Call inlinings: 2
  - Simplifications: 0




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 4
  - Call inlinings: 2
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

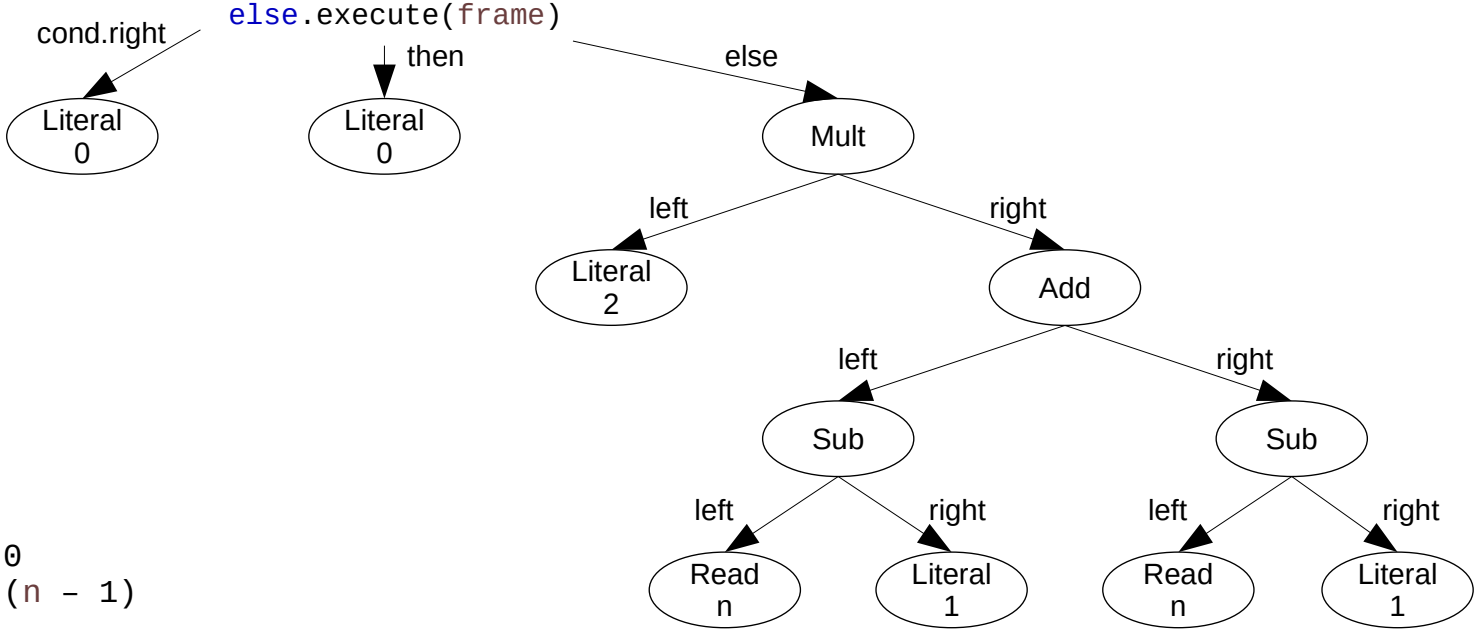



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 4
  - Call inlinings: 3
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

Inline call

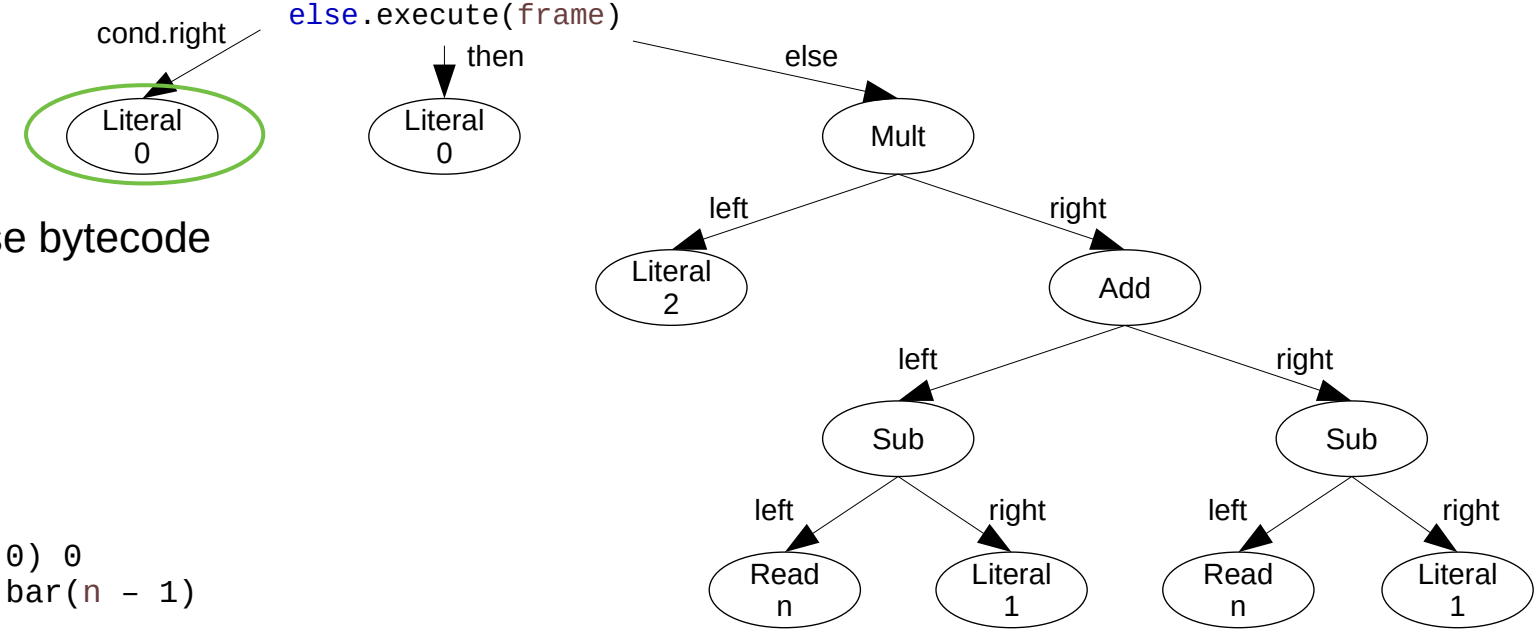



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 4
  - Call inlinings: 3
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

Inline call



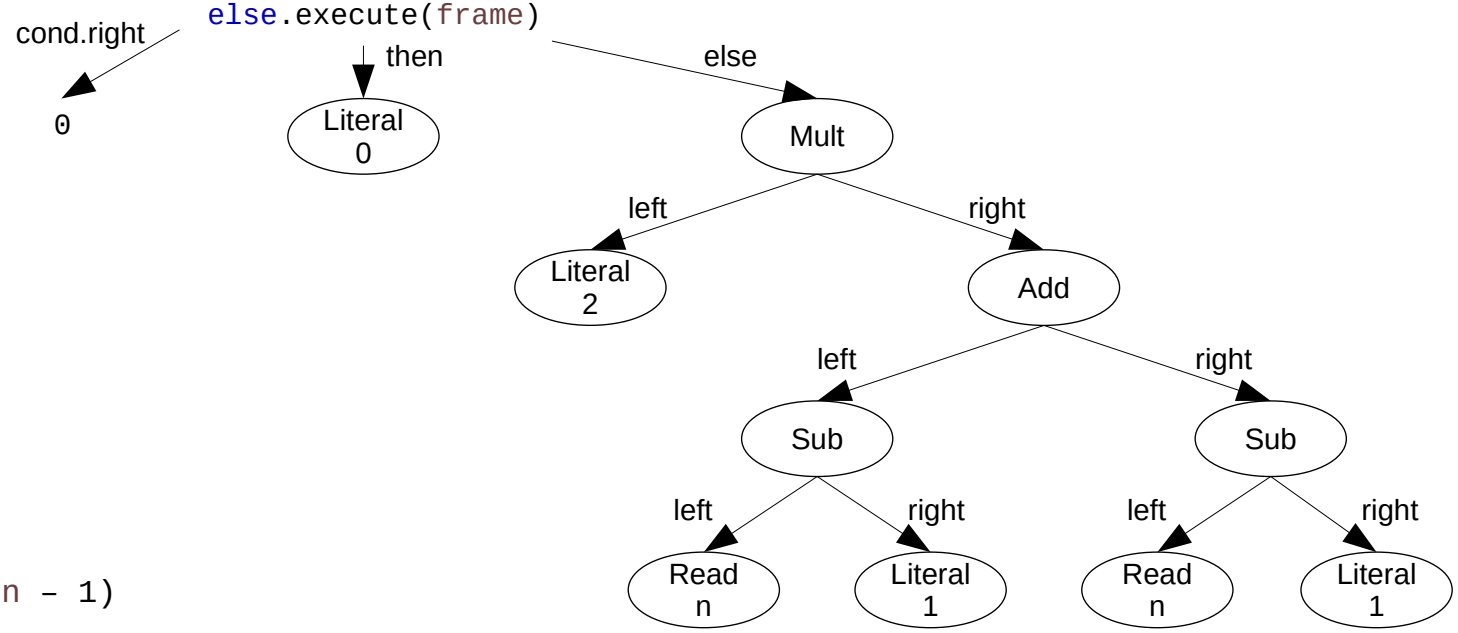
Parse bytecode

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 4
  - Call inlinings: 3
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") ==
 cond.right.execute(frame) ?
 then.execute(frame) :
 else.execute(frame)
```

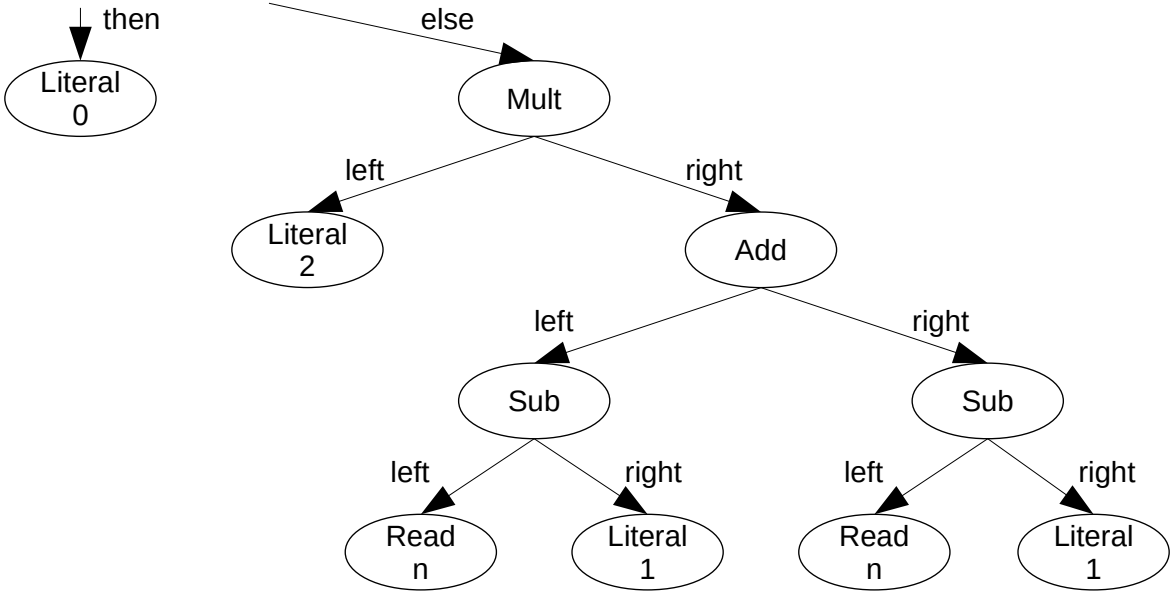
Inline call




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 5
  - Call inlinings: 3
  - Simplifications: 0

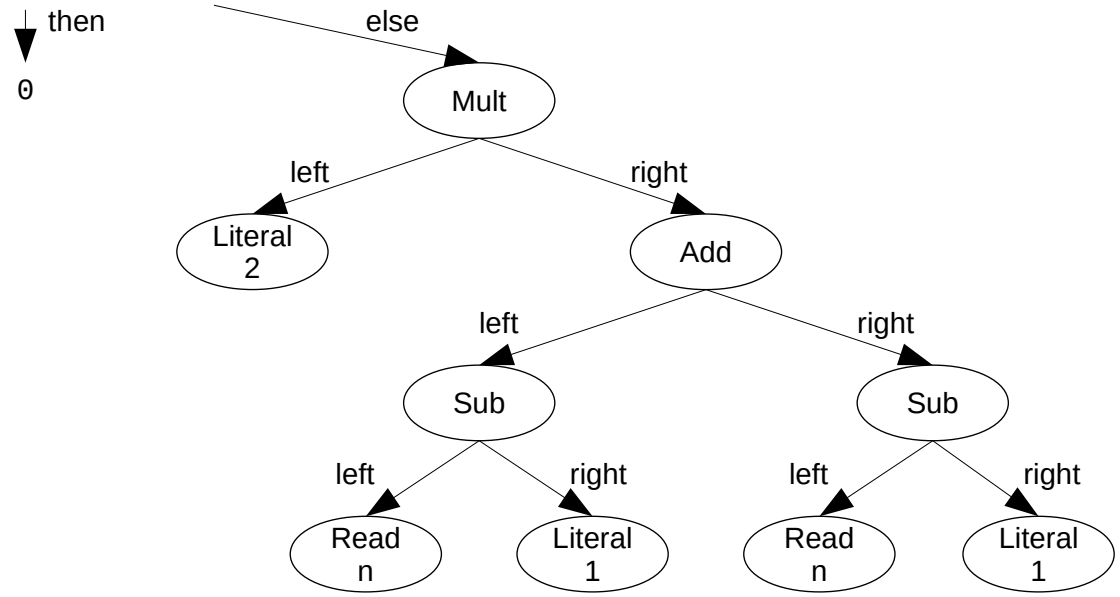
```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 then.execute(frame) :
 else.execute(frame)
```




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 5
  - Call inlinings: 4
  - Simplifications: 0

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 then.execute(frame) :
 else.execute(frame)
```

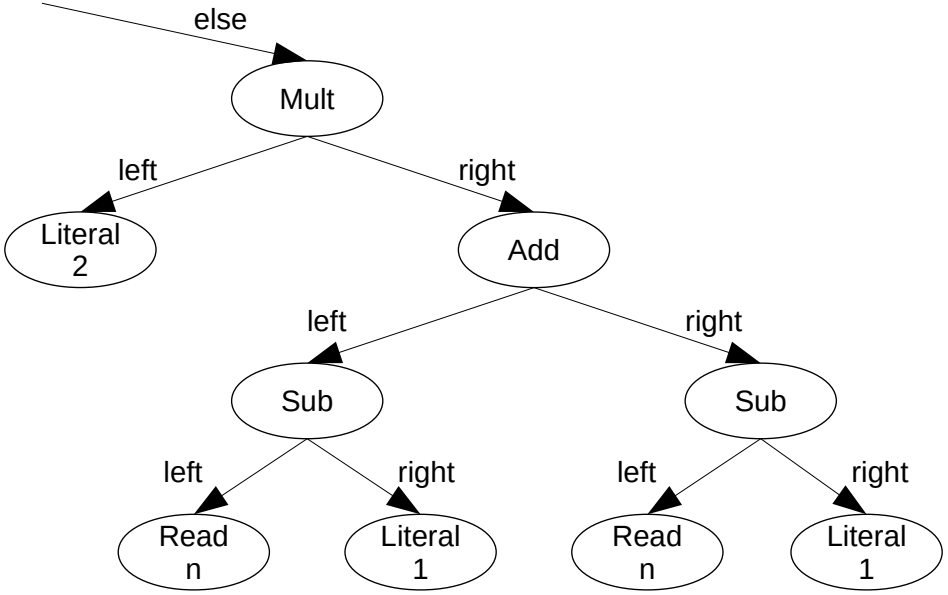



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 6
  - Call inlinings: 4
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 else.execute(frame)
```

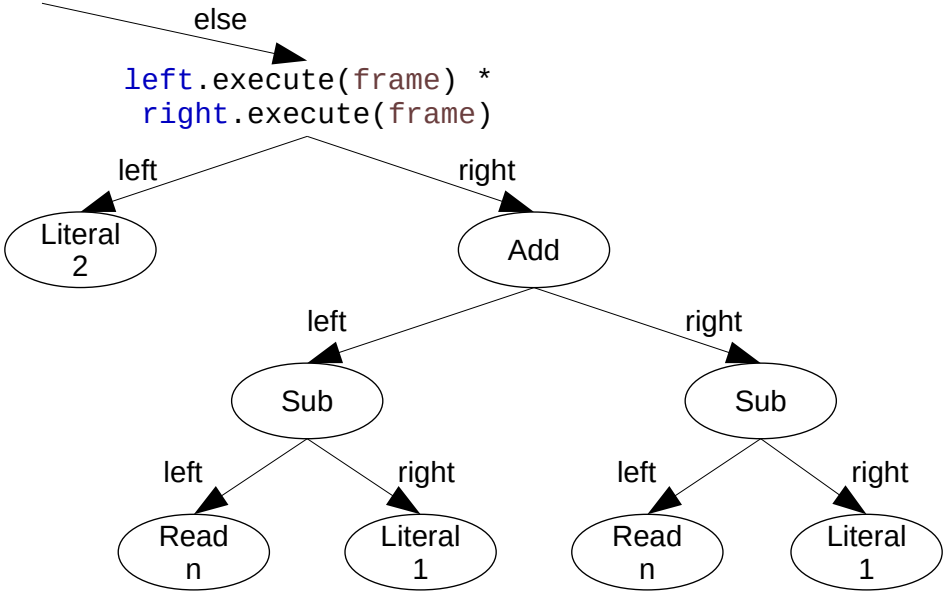



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 6
  - Call inlinings: 5
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 else.execute(frame)
```

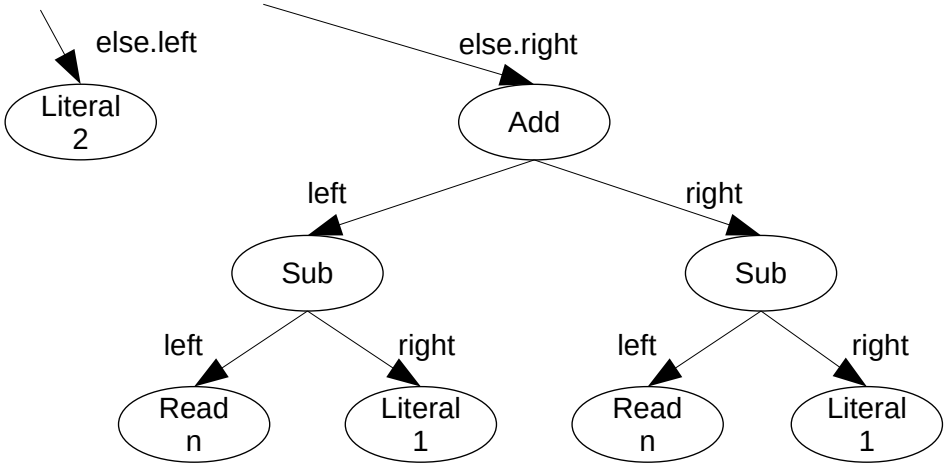



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 7
  - Call inlinings: 5
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 else.left.execute(frame) *
  else.right.execute(frame)
```

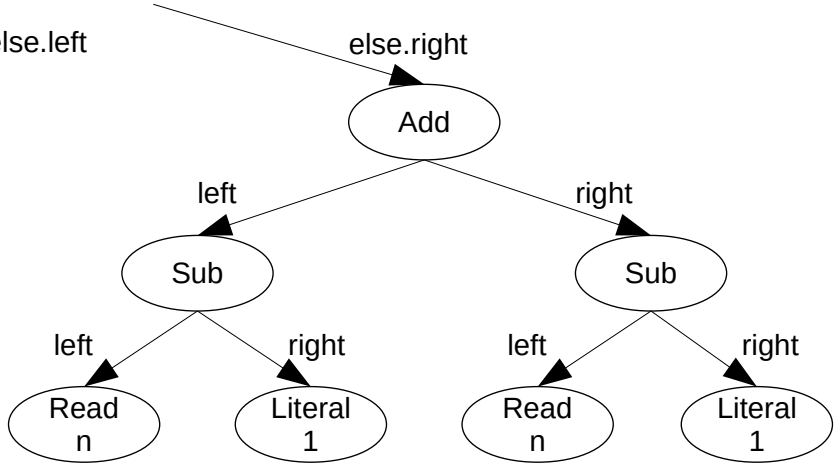



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 7
  - Call inlinings: 6
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 else.left.execute(frame) *
  else.right.execute(frame)
```

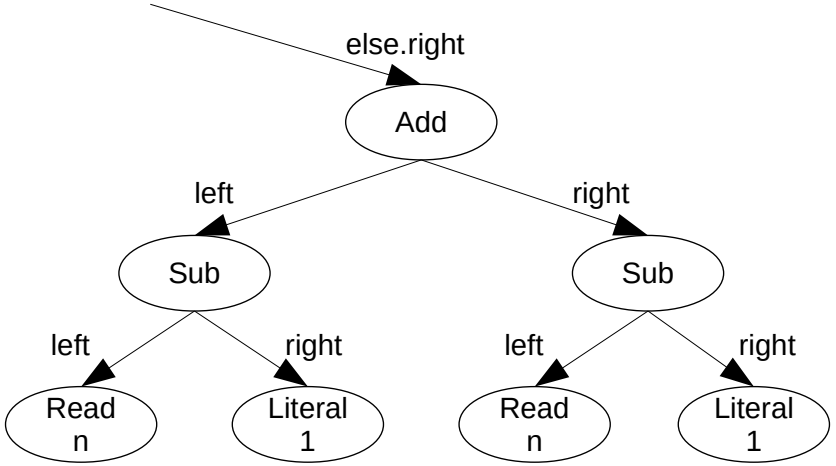



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 8
  - Call inlinings: 6
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.execute(frame)
```

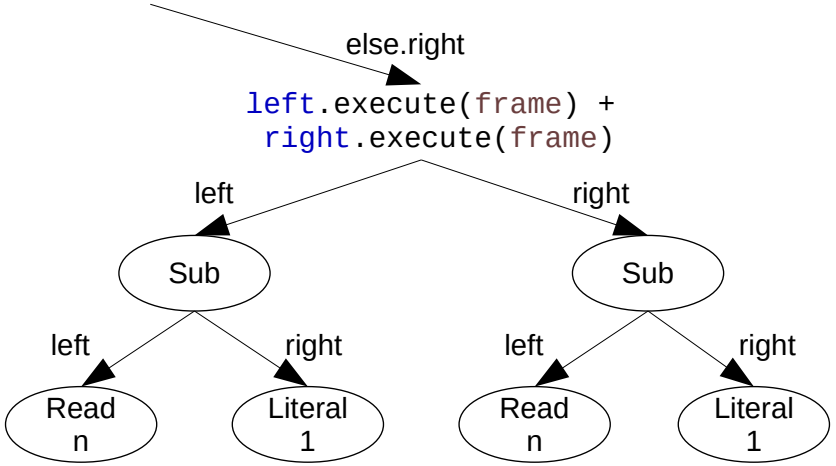



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 8
  - Call inlinings: 7
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.execute(frame)
```

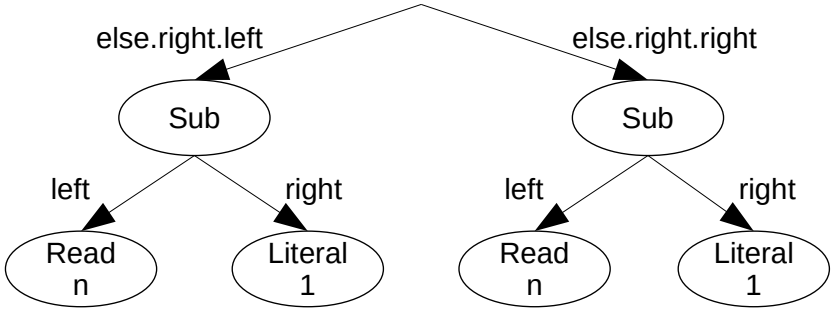



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 9
  - Call inlinings: 7
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * (else.right.left.execute(frame) +
  else.right.right.execute(frame))
```

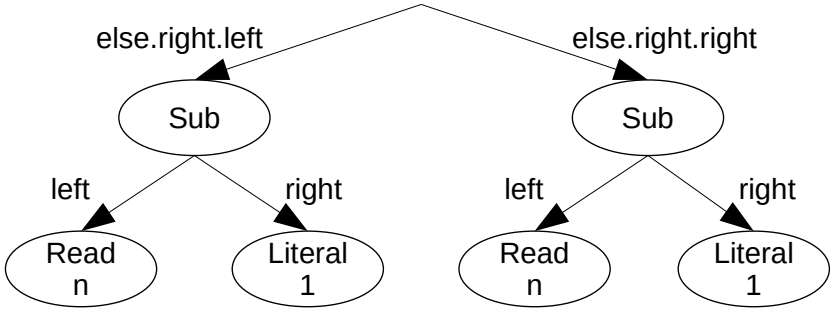



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 9
  - Call inlinings: 8
  - Simplifications: 0

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.left.execute(frame) +
  2 * else.right.right.execute(frame)
```

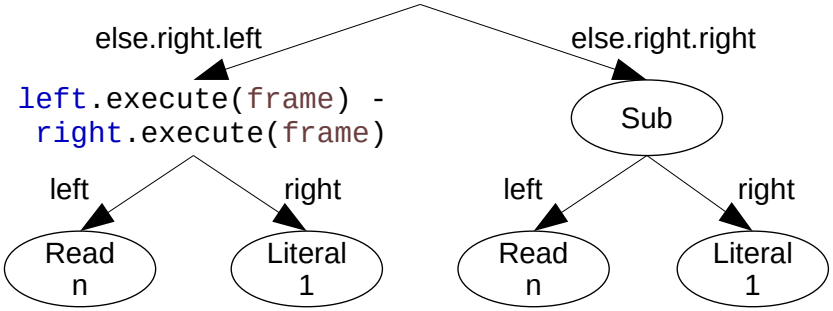



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 9
  - Call inlinings: 8
  - Simplifications: 1

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.left.execute(frame) +
  2 * else.right.right.execute(frame)
```

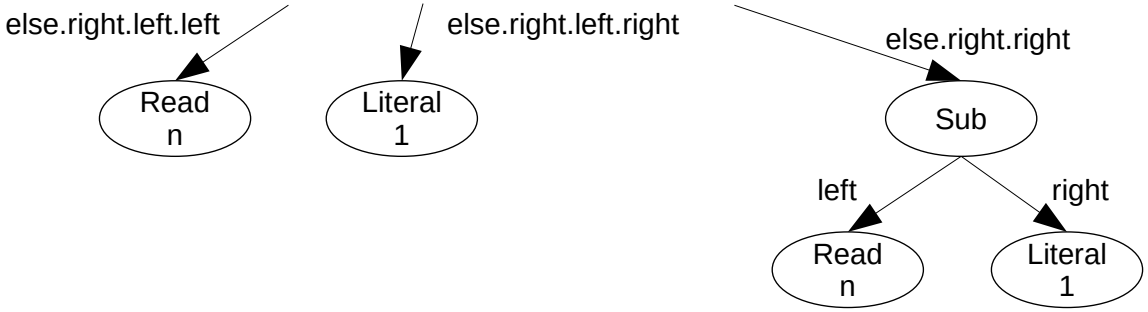



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 10
  - Call inlinings: 8
  - Simplifications: 1

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * (else.right.left.left.execute(frame) -
  else.right.left.right.execute(frame)) +
  2 * else.right.right.execute(frame)
```

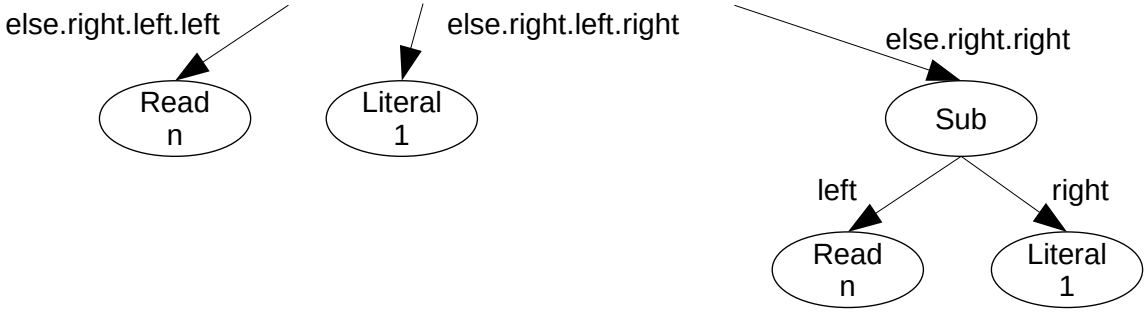



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 10
  - Call inlinings: 9
  - Simplifications: 1

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.left.left.execute(frame) -
  2 * else.right.left.right.execute(frame) +
  2 * else.right.right.execute(frame)
```

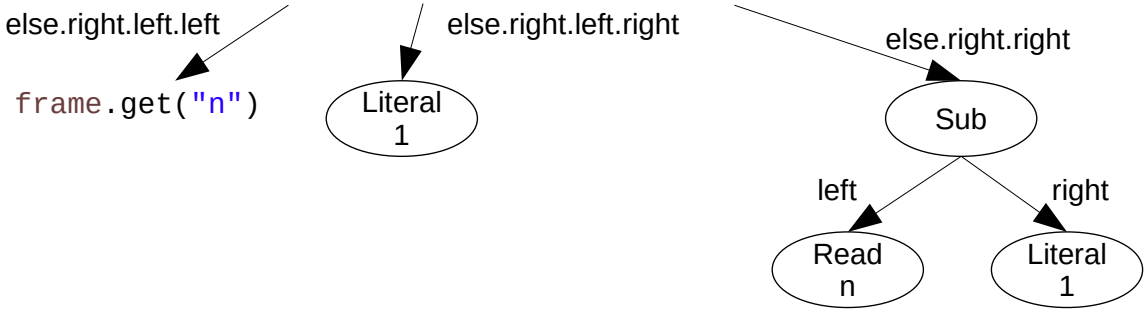



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 10
  - Call inlinings: 9
  - Simplifications: 2

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * else.right.left.left.execute(frame) -
  2 * else.right.left.right.execute(frame) +
  2 * else.right.right.execute(frame)
```

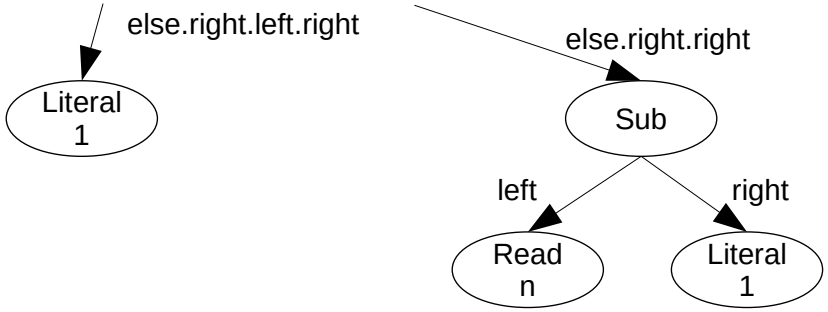



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 11
  - Call inlinings: 9
  - Simplifications: 2

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") -
  2 * else.right.left.right.execute(frame) +
  2 * else.right.right.execute(frame)
```

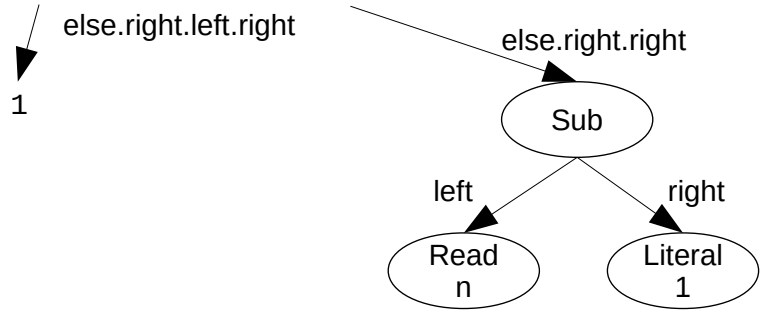



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 11
  - Call inlinings: 10
  - Simplifications: 2

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") -
  2 * else.right.left.right.execute(frame) +
  2 * else.right.right.execute(frame)
```

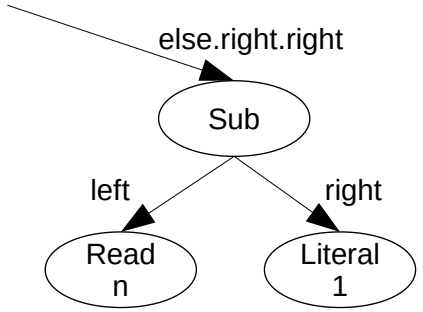



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 12
  - Call inlinings: 10
  - Simplifications: 2

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") -
  2 * 1 +
  2 * else.right.right.execute(frame)
```

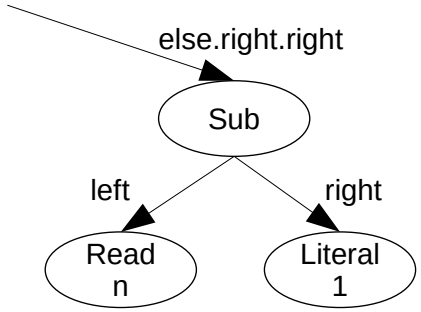



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 12
  - Call inlinings: 11
  - Simplifications: 2

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") - 2 +
  2 * else.right.right.execute(frame)
```

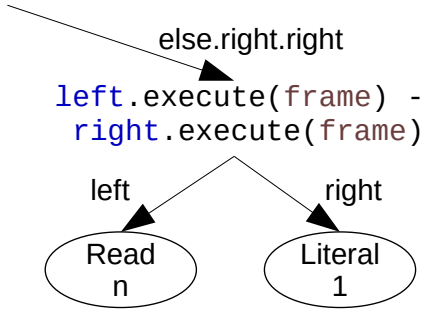



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 12
  - Call inlinings: 11
  - Simplifications: 3

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") - 2 +
  2 * else.right.right.execute(frame)
```

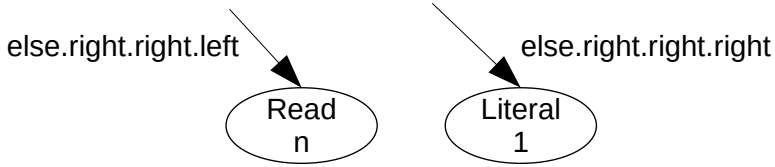



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 13
  - Call inlinings: 11
  - Simplifications: 3

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") – 2 +
  2 * (else.right.right.left.execute(frame) -
   else.right.right.right.execute(frame))
```

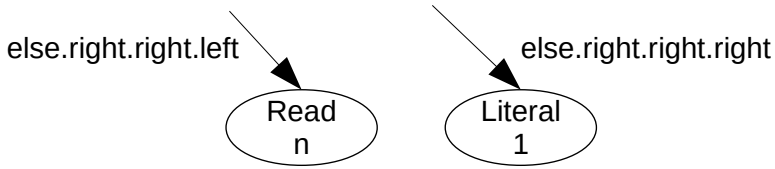



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 13
  - Call inlinings: 12
  - Simplifications: 3

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") – 2 +
  2 * else.right.right.left.execute(frame) -
  2 * else.right.right.right.execute(frame)
```

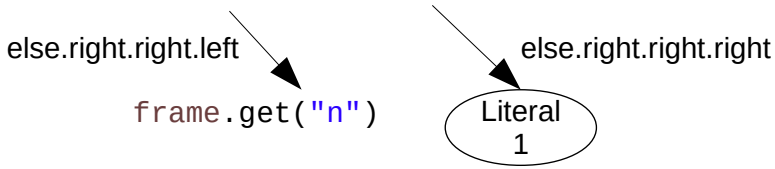



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 13
  - Call inlinings: 12
  - Simplifications: 4

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") – 2 +
  2 * else.right.right.left.execute(frame) -
  2 * else.right.right.right.execute(frame)
```

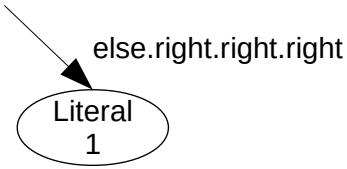



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 14
  - Call inlinings: 12
  - Simplifications: 4

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 2 * frame.get("n") – 2 +
  2 * frame.get("n") -
  2 * else.right.right.right.execute(frame)
```

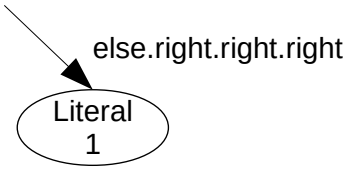



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 14
  - Call inlinings: 13
  - Simplifications: 4

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") - 2 +
  2 * else.right.right.right.execute(frame)
```

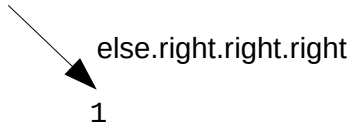



```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 14
  - Call inlinings: 13
  - Simplifications: 5

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") – 2 +
  2 * else.right.right.right.execute(frame)
```




```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 13
  - Simplifications: 5

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") – 2 +
  2 * 1
```

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 14
  - Simplifications: 5

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") – 2 + 2
```

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 14
  - Simplifications: 6

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") – 4
```

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 14
  - Simplifications: 7

And so on …

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") – 4
```

- Escape analysis:
  - frame is not assigned to any field ✓
  - frame is no longer provided to any execute call ✓

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 14
  - Simplifications: 7

```
frame = new Frame()
frame.addSlot("n")

frame.get("n") == 0 ?
 0 :
 4 * frame.get("n") - 4
```

- Escape analysis:
  - ☐ frame is not assigned to any field ✓
  - ☐ frame is no longer provided to any execute call ✓
- With this:
  - ☐ frame can be removed
  - ☐ n can be treated like a local variable

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n - 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - ☐ Bytecode parsings: 15
  - ☐ Call inlinings: 14
  - ☐ Simplifications: 7

```
n == 0 ?
 0 :
 4 * n - 4
```

Final bytecode

- Escape analysis:
  - frame is not assigned to any field ✓
  - frame is no longer provided to any execute call ✓
- With this:
  - frame can be removed
  - n can be treated like a local variable

```
foo(n) {
  if (n == 0) 0
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - Bytecode parsings: 15
  - Call inlinings: 14
  - Simplifications: 7

```
n == 0 ?
 0 :
 4 * n - 4
```

Final bytecode

- Escape analysis:
  - ☐ frame is not assigned to any field ✓
  - ☐ frame is no longer provided to any execute call ✓
- With this:

What if we do this for large ASTs?

Example: zlib with ~ 70 000 AST nodes

```
foo(n) {
  if (n
  else 2 * bar(n – 1)
}

bar(n) {
  n + n
}
```

- Counters:
  - ☐ Bytecode parsings: 15
  - ☐ Call inlinings: 14
  - ☐ Simplifications: 7

Motivation

- Partial evaluation is slow for large, complex programs

- Benchmark partial evaluation
- Future attempts to speed up partial evaluation
- Compare results before and after

Solution

- Benchmarking framework using JMH [3]:
  - user provides AST and call arguments
  - base class initializes compiler and performs partial evaluation

- PELang:
  - simple, optimized Truffle language
  - stable to guarantee comparability
  - ASTs are programatically created

Implementation

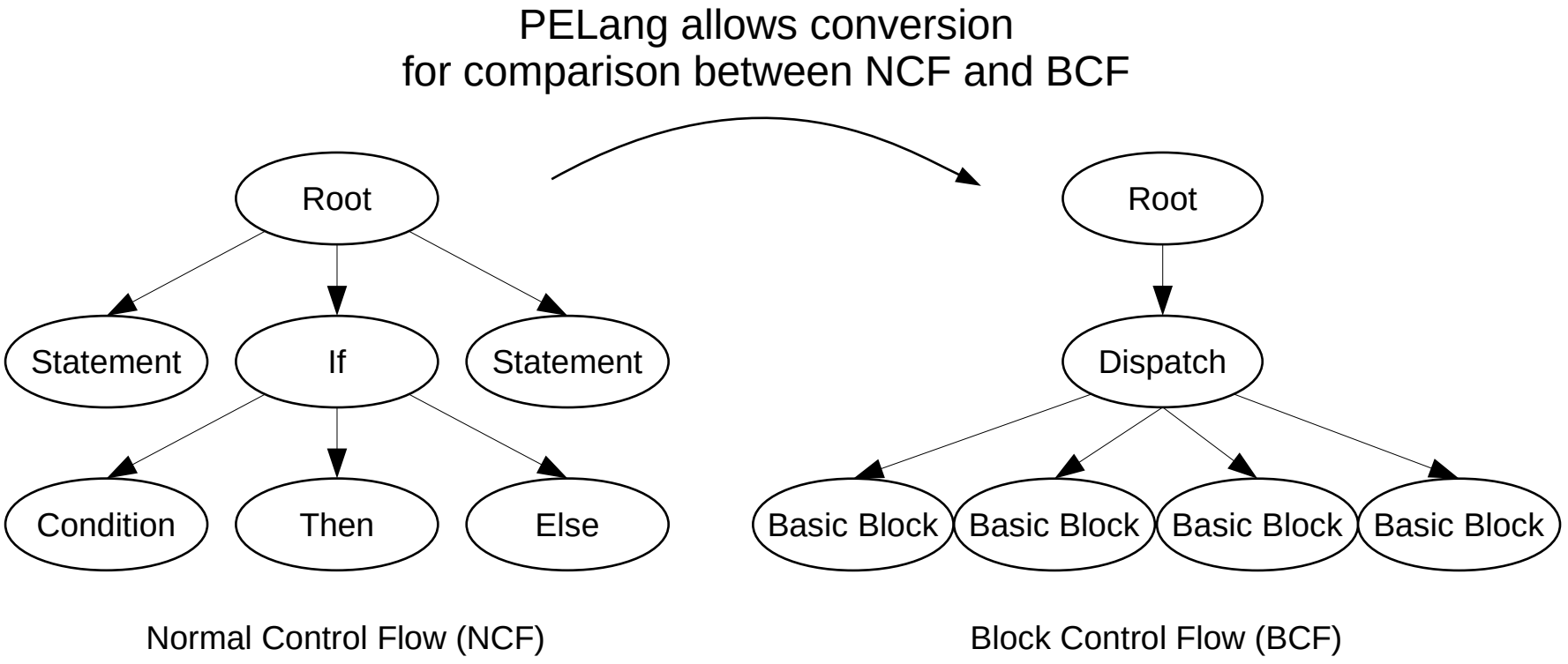
```
public class PELangNCFBenchmarks {

  // other benchmark classes
  …

  public static class BinaryTrees extends PartialEvaluationBenchmark {

    @Override
    protected RootNode rootNode() {
      // use the binary trees sample AST
      return PELangSample.binaryTrees();
    }

    @Override
    protected Object[] callArguments() {
      // call binary trees with depth 10
      return new Object[]{10L};
    }

  }

}
```

Implementation

```
@State(Scope.Benchmark)
public abstract class PartialEvaluationBenchmark {

  // compiler and call target
  …

  // compiler and call target initialization
  public PELangBenchmark() { … }

  // overwritten by sub-class to provide AST
  protected abstract RootNode rootNode();

  // can be overwritten by sub-class to provide call arguments
  protected Object[] callArguments() {
    return new Object[0];
  }

  @Setup
  public void warmupAST() {
    for (int i = 0; i < AST_WARMUPS; i++)
      callTarget.call(callArguments);
  }

  @Benchmark
  public StructuredGraph createGraph() {
    return partialEvaluator.createGraph(callTarget, …);
  }

}
```

Unstructured Control Flow

- Bytecode interpreters use different kinds of ASTs:
  - to support programs with unstructured control flow



SubstrateVM Integration

- JMH ported to SubstrateVM
- For comparison between GraalVM and SubstrateVM

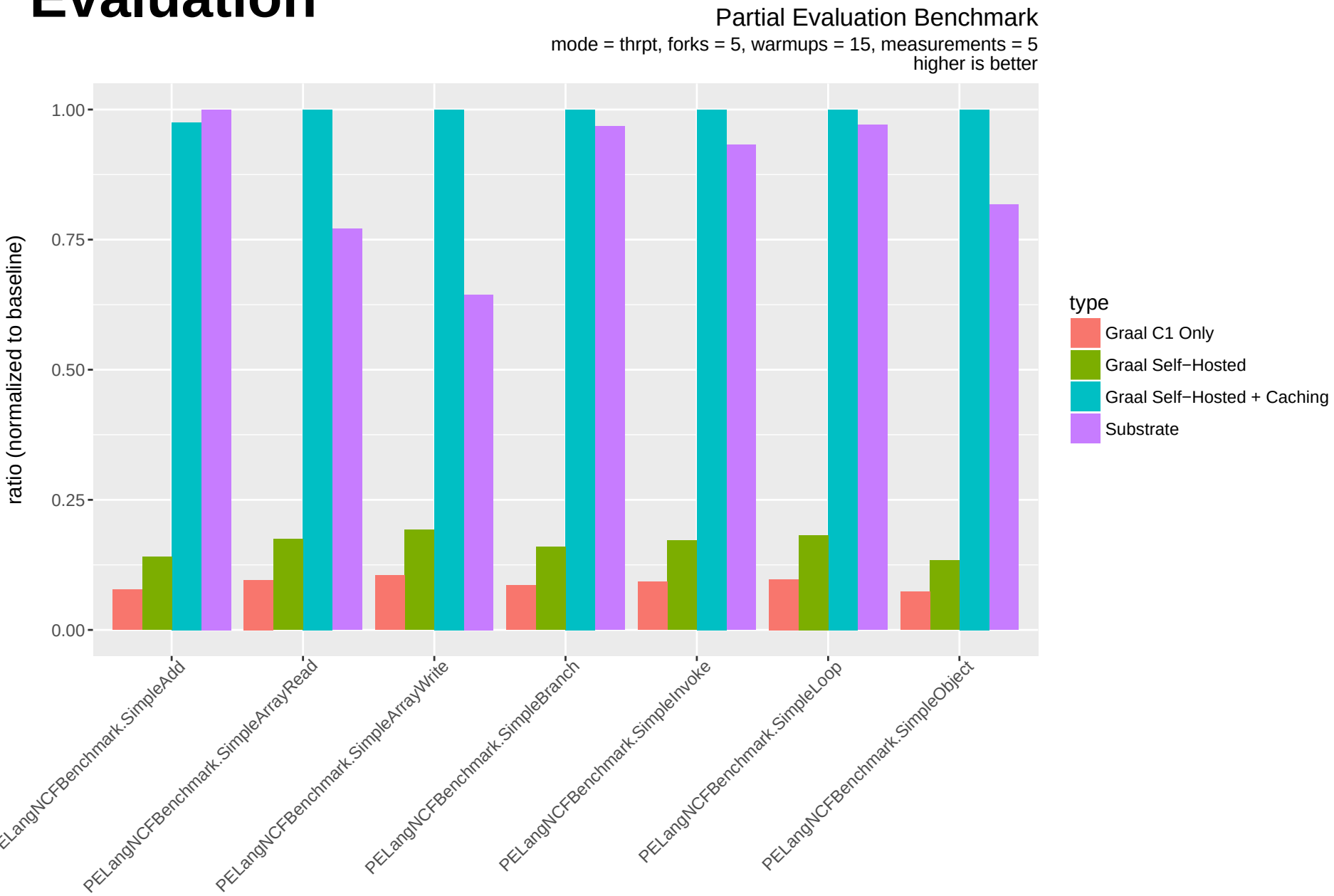
```
$ mx native-image --tool:truffle --tool:jmh -jar pelang-benchmarks.jar
   classlist:   4,855.53 ms
       (cap):   1,446.17 ms
       setup:   6,853.47 ms
         ...
     [total]: 164,286.51 ms

$ ./pelang-benchmarks PELangNCFBenchmark.BinaryTrees.createGraph
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
...
Iteration   4: 11.851 ops/s
Iteration   5: 11.555 ops/s

Result "PELangNCFBenchmark.BinaryTrees.createGraph":
  11.023 ±(99.9%) 2.777 ops/s [Average]
  (min, avg, max) = (9.998, 11.023, 11.851), stdev = 0.721
  CI (99.9%): [8.245, 13.800] (assumes normal distribution)
```

Evaluation

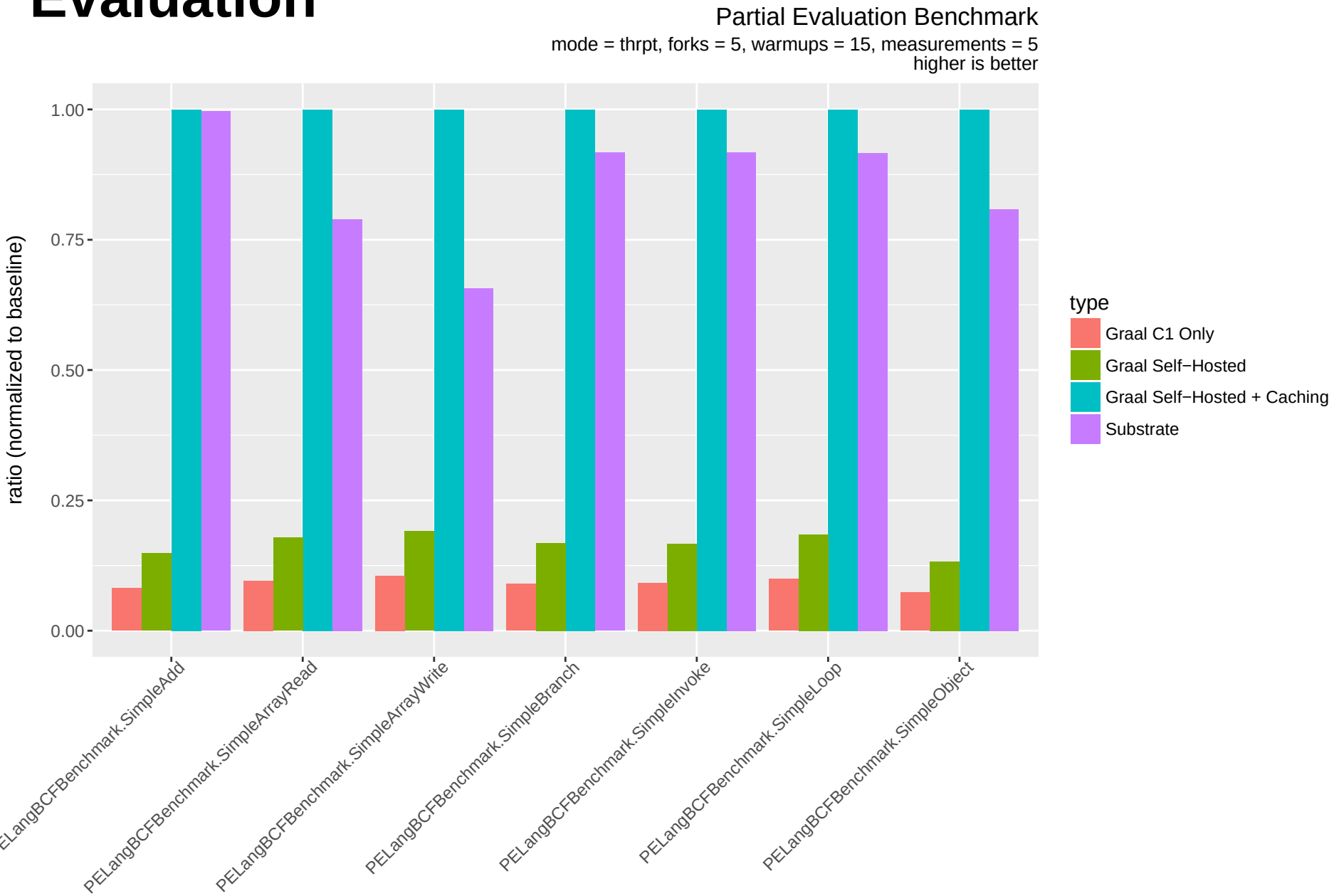
- Based on examples used for testing
- Measured throughput of partial evaluation

- Comparison between NCF and BCF variants
- Comparison between different configurations:
  - Graal C1 only
  - Graal self-hosted
  - Graal self-hosted with graph caching
  - Substrate

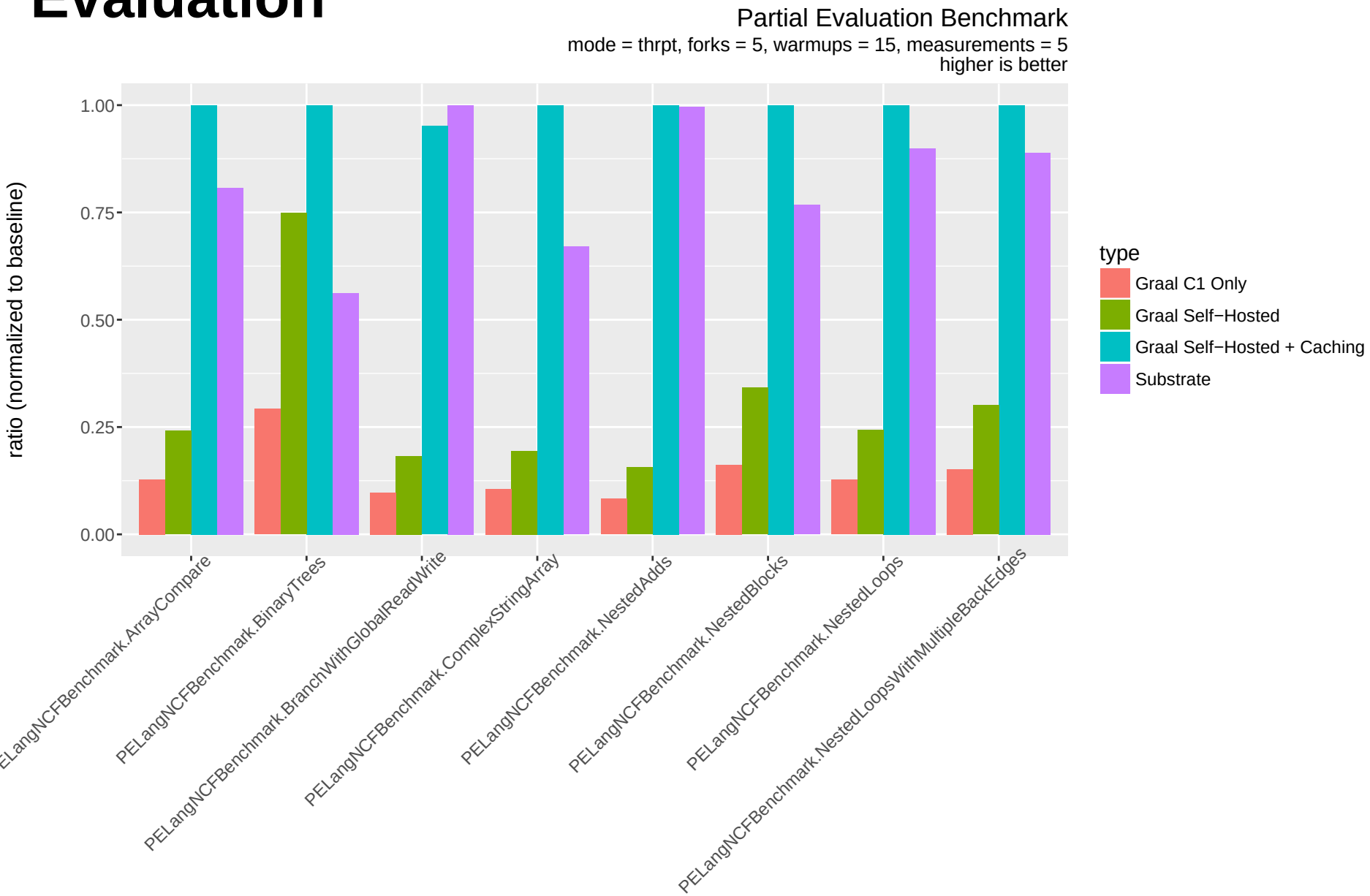
Evaluation


Partial Evaluation Benchmark
mode = thrpt, forks = 5, warmups = 15, measurements = 5
higher is better
ratio (normalized to baseline)
1.00
0.75
0.50
0.25
0.00
PELangNCFBenchmark.SimpleAdd
PELangNCFBenchmark.SimpleArrayRead
PELangNCFBenchmark.SimpleArrayWrite
PELangNCFBenchmark.SimpleBranch
PELangNCFBenchmark.SimpleInvoke
PELangNCFBenchmark.SimpleLoop
PELangNCFBenchmark.SimpleObject
type
Graal C1 Only
Graal Self−Hosted
Graal Self−Hosted + Caching
Substrate

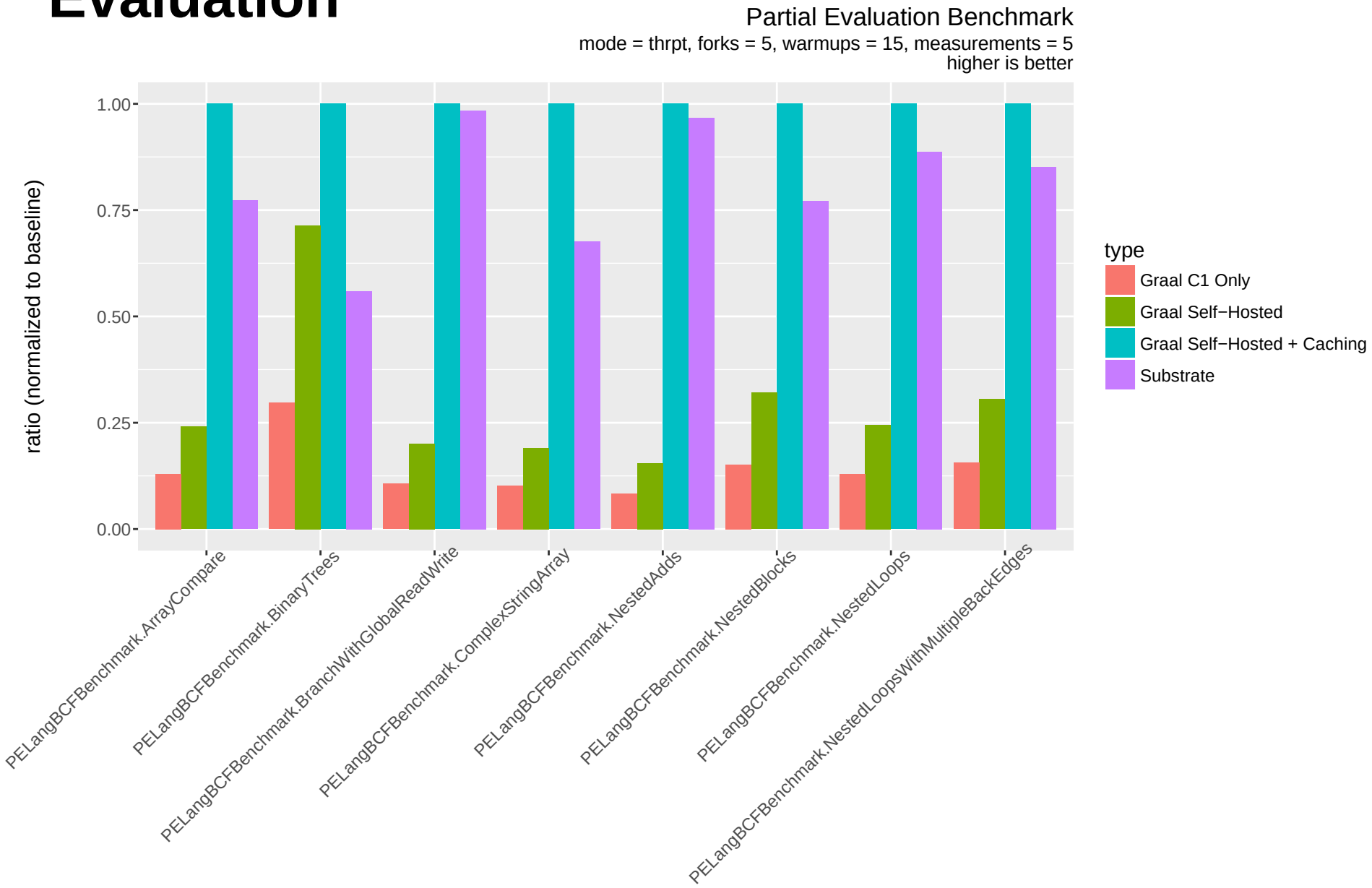
Evaluation


Partial Evaluation Benchmark
mode = thrpt, forks = 5, warmups = 15, measurements = 5
higher is better
ratio (normalized to baseline)
1.00
0.75
0.50
0.25
0.00
PELangBCFBenchmark.SimpleAdd
PELangBCFBenchmark.SimpleArrayRead
PELangBCFBenchmark.SimpleArrayWrite
PELangBCFBenchmark.SimpleBranch
PELangBCFBenchmark.SimpleInvoke
PELangBCFBenchmark.SimpleLoop
PELangBCFBenchmark.SimpleObject
type
Graal C1 Only
Graal Self−Hosted
Graal Self−Hosted + Caching
Substrate

Evaluation


Partial Evaluation Benchmark
mode = thrpt, forks = 5, warmups = 15, measurements = 5
higher is better
ratio (normalized to baseline)
1.00
0.75
0.50
0.25
0.00
PELangNCFBenchmark.ArrayCompare
PELangNCFBenchmark.BinaryTrees
PELangNCFBenchmark.BranchWithGlobalReadWrite
PELangNCFBenchmark.ComplexStringArray
PELangNCFBenchmark.NestedAdds
PELangNCFBenchmark.NestedBlocks
PELangNCFBenchmark.NestedLoops
PELangNCFBenchmark.NestedLoopsWithMultipleBackEdges
type
Graal C1 Only
Graal Self−Hosted
Graal Self−Hosted + Caching
Substrate

Evaluation


Partial Evaluation Benchmark
mode = thrpt, forks = 5, warmups = 15, measurements = 5
higher is better
ratio (normalized to baseline)
1.00
0.75
0.50
0.25
0.00
type
Graal C1 Only
Graal Self−Hosted
Graal Self−Hosted + Caching
Substrate
PELangBCFBenchmark.ArrayCompare
PELangBCFBenchmark.BinaryTrees
PELangBCFBenchmark.BranchWithGlobalReadWrite
PELangBCFBenchmark.ComplexStringArray
PELangBCFBenchmark.NestedAdds
PELangBCFBenchmark.NestedBlocks
PELangBCFBenchmark.NestedLoops
PELangBCFBenchmark.NestedLoopsWithMultipleBackEdges

Future Work

- More benchmarks with larger, complex examples
- Speed up partial evaluation:
  - by implementing second Futamura projection
- Comparison of benchmark results before and after

Q/A

References

- [1] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software (Onward! 2013). ACM, New York, NY, USA, 187-204. DOI=http://dx.doi.org/10.1145/2509578.2509581
- [2] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. Higher-Order and Symbolic Computation, 12(4), 381-391. DOI=https://doi.org/10.1023/A:1010095604496
- [3] http://openjdk.java.net/projects/code-tools/jmh/