Interpreters Everywhere and All the Time

Stefan Marr

PLISS, September 2023

Engineering and Physical Sciences Research Council

The Royal Society

University of Kent

Got a Question?
Please Interrupt Me!

Stefan Marr

Tinkering with interpreters
for 6,000 days, and counting

How it Started

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Course on
Virtual Machines
1st of April 2007

SOM
SimpleObjectMachine

Course Project:
Build a just-in-time
compiler for CSOM

In between:

Many Years of Brussels & Lille

(and Linz, but without the chocolate)

This is THE BEST Chocolate!

How it's Going

University of Kent

SOM SimpleObjectMachine

THE ROYAL SOCIETY

"something like a professor"

still tinkering with interpreters

Who Are We?

Bing Image Creator

Please stand up, if you have used a programming language!

Bing Image Creator

Please stand, **if you have implemented** a programming language!

Bing Image Creator

Please stand,
if it's a language we have or will hear of!

Bing Image Creator

Please stand, **if it's any kind of interpreter!**

Bing Image Creator

Topics Discussed Today

- How are programming languages implemented?
- Types of interpreters
  - Abstract syntax tree
  - Bytecode
- Interpreter optimizations
  - Lookup caching
  - AST/bytecode-level inlining
  - Library lowering, library intrinsification
  - Super nodes, super instructions
  - Self-optimization, bytecode quickening

Got a Question?
Please Interrupt Me!

Disclaimer!

Through the lens of Meta-compilation Systems

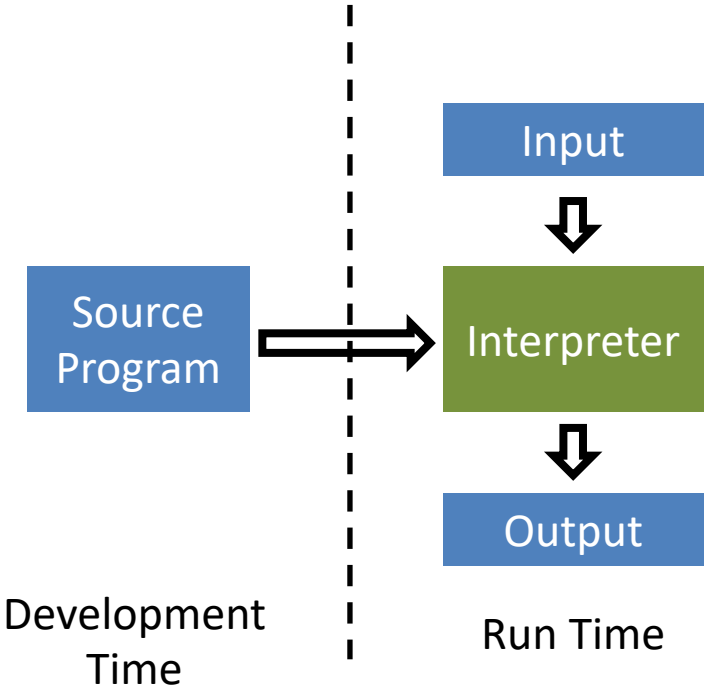
Graal+Truffle and RPython

Bing Image Creator

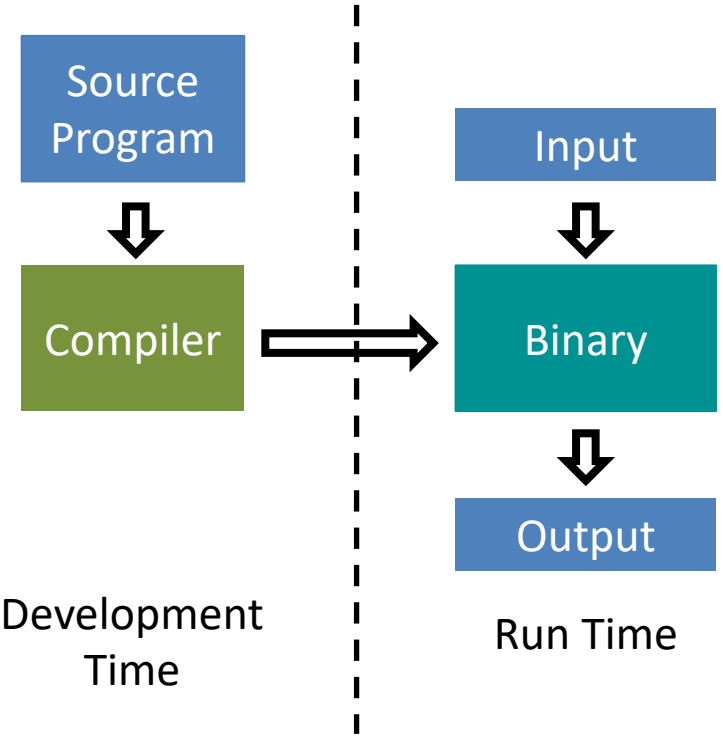
I am not using it today, but highly recommended: https://craftinginterpreters.com/

HOW TO IMPLEMENT A PROGRAMMING LANGUAGE?

Language Implementation Approaches



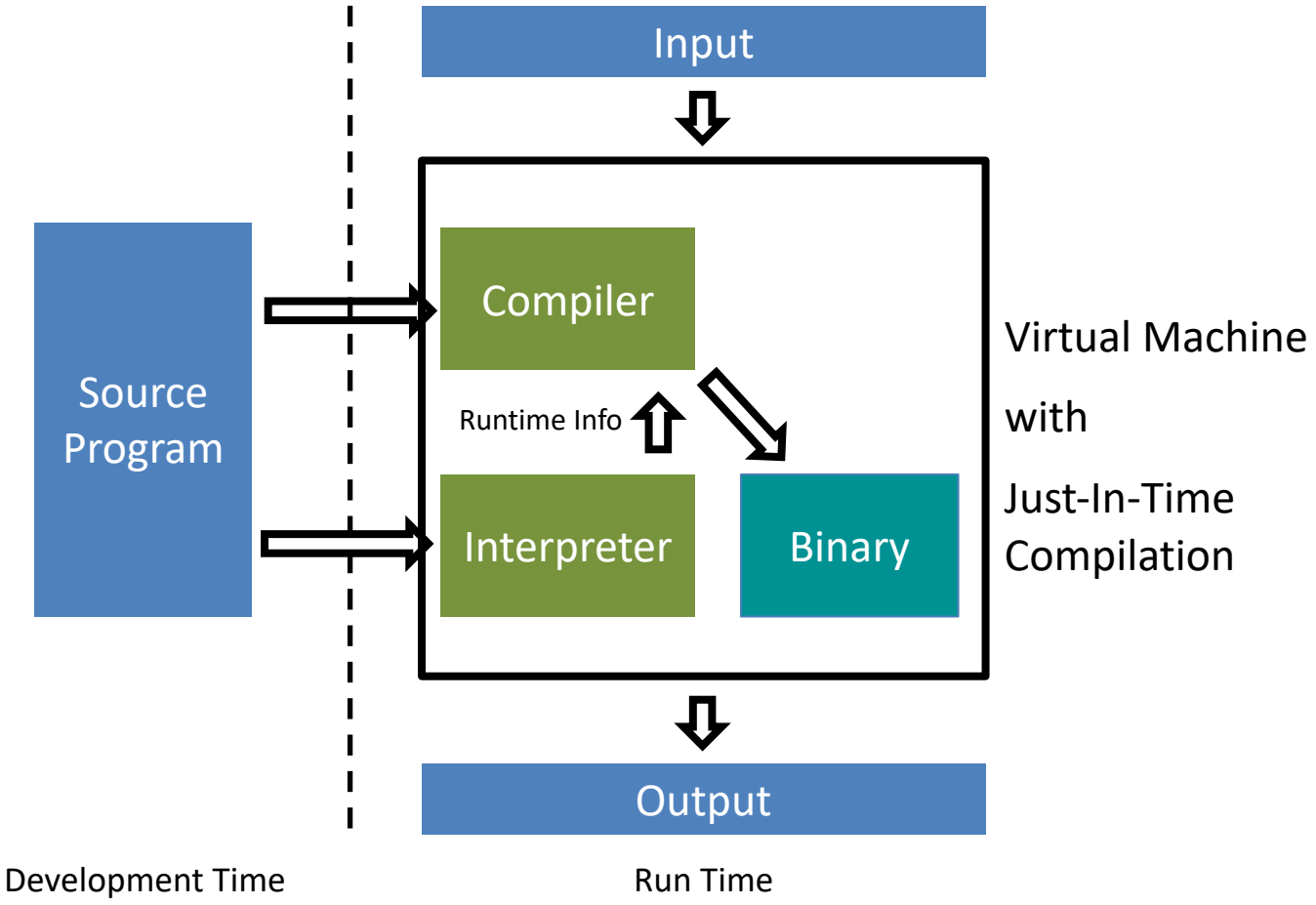
Simple, but often slow



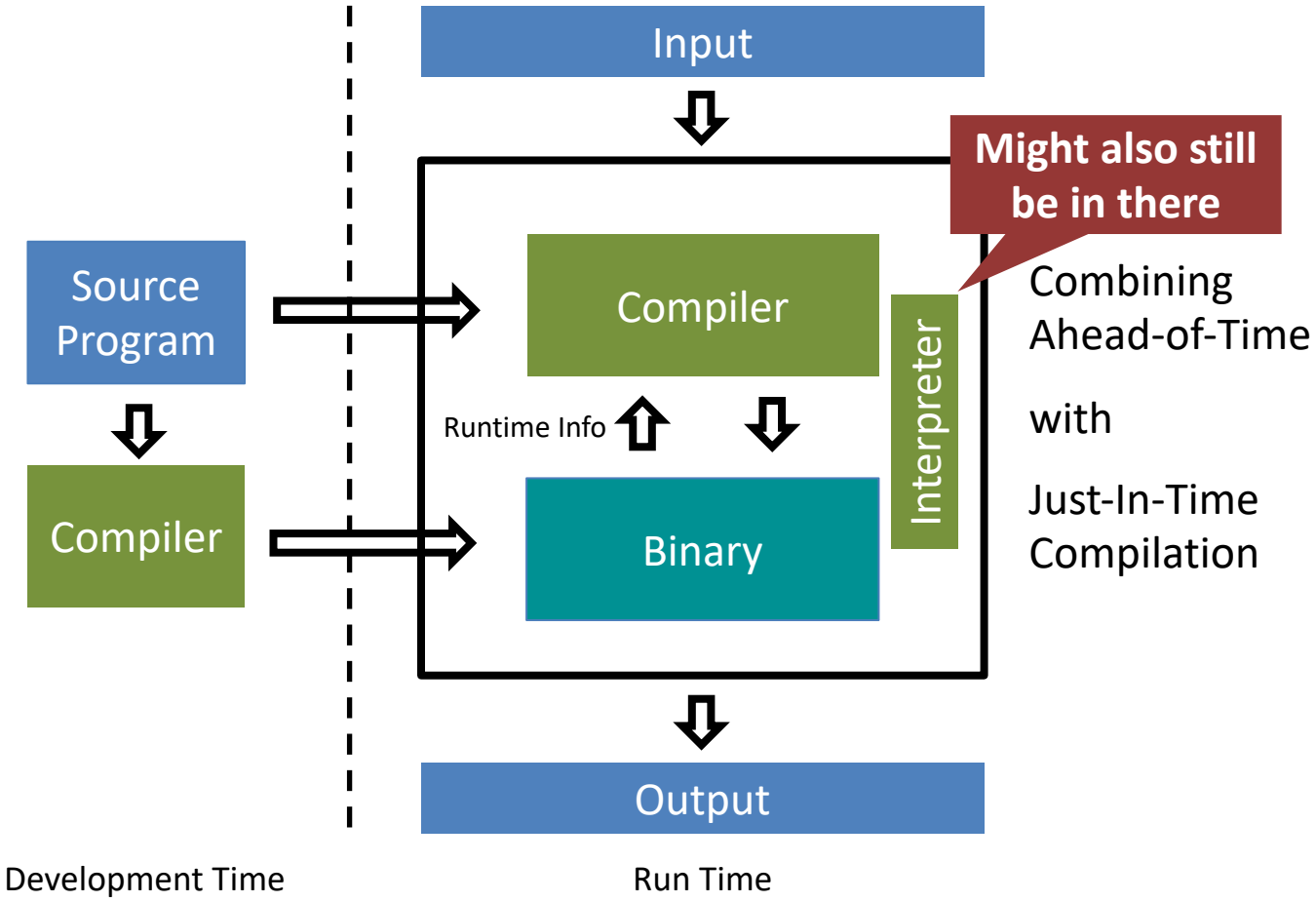
More complex, but often faster

Not ideal for all languages.

Modern Virtual Machines



Virtual Machines with Ahead-of-Time Compilation



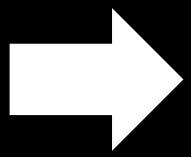
Pointers for this week

- JavaScript AOT Compilation
  Manuel Serrano
- Programming Languages on the Web
  Michael Lippautz
- A Brief Introduction to Just-in-Time Compilation
  by myself

TYPES OF INTERPRETERS

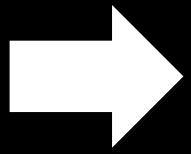
Code Convention

JavaScript-ish



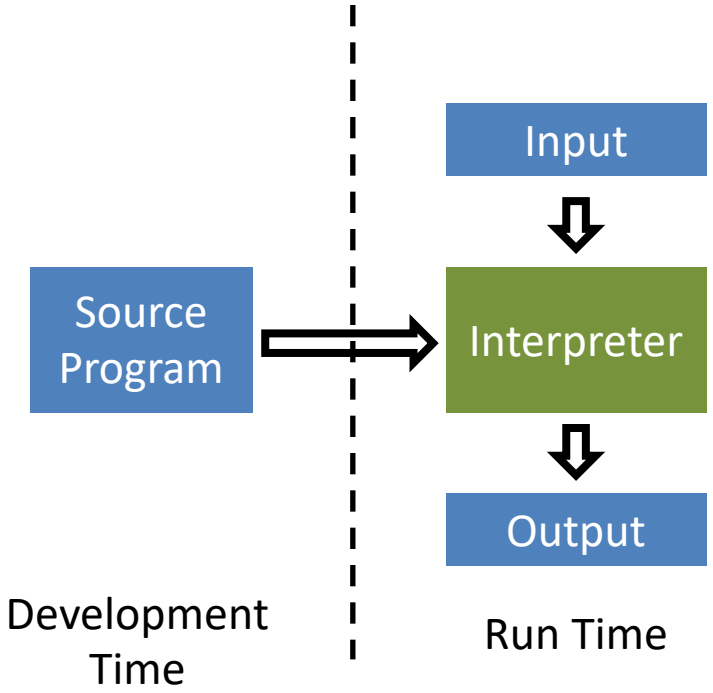
Application Code

Python-ish
with implicit typing



Interpreter Code

Basic Interpretation



From Program to Abstract Syntax Tree

```
function hello() {
  log("Hello");
}
```

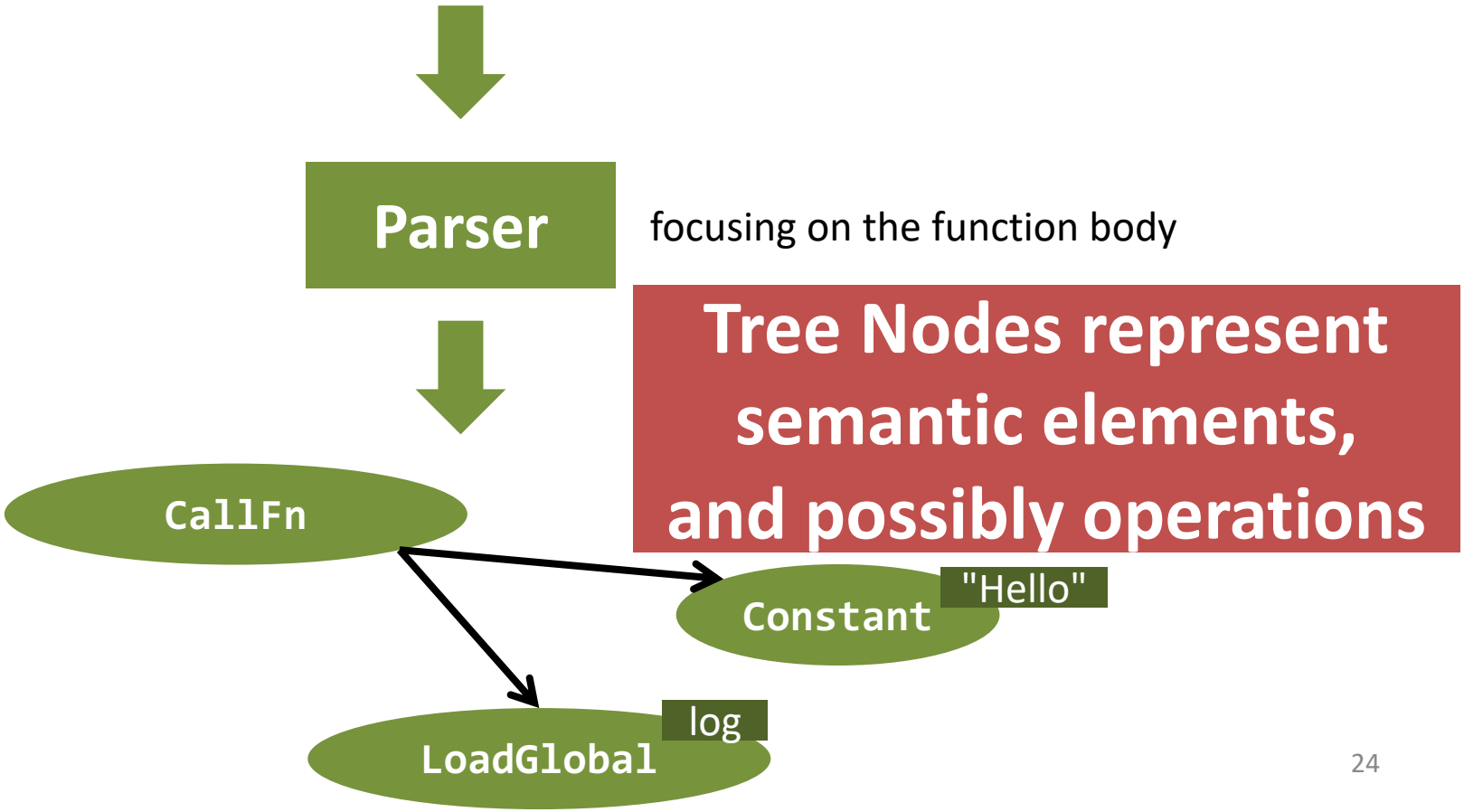
programs are read by a parser

a parser is based on a grammar describing the language

```
functionDeclaration:
  Function Identifier
  '(' formalParameterList? ')'
  '{' functionBody '}'
```

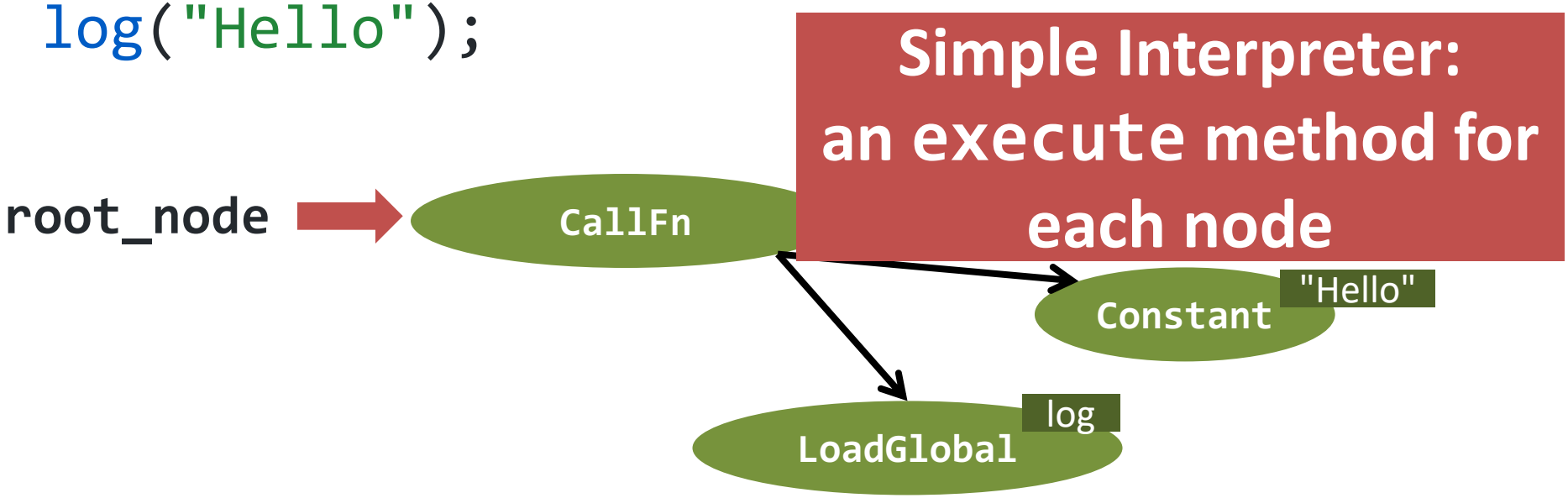
From Program to Abstract Syntax Tree

```
function hello() {
   log("Hello");
}
```



A Simple Abstract Syntax Tree Interpreter

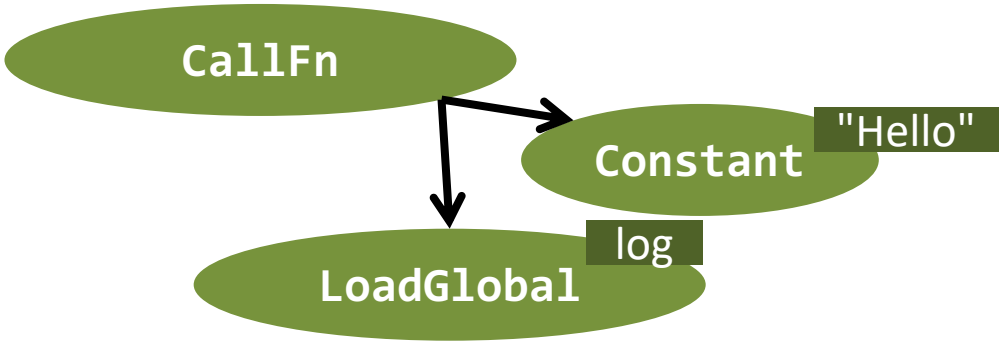
```
log("Hello");
```




```
root_node = parse(file)
root_node.execute(Frame())
```

Implementing `execute()` methods

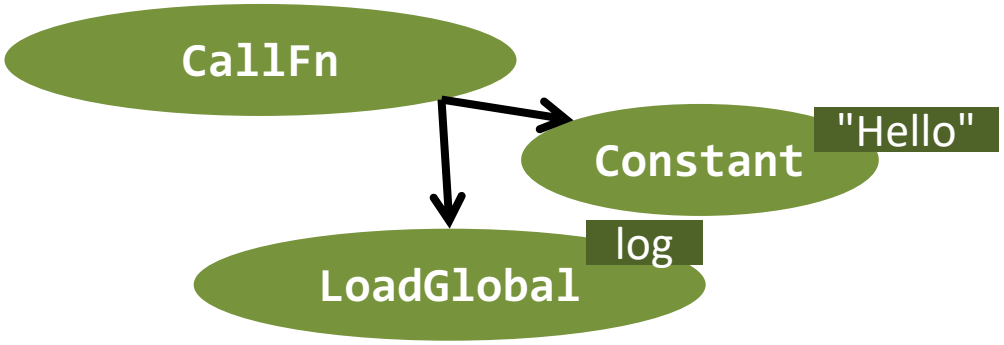
```
log("Hello");
```




```
class Constant:
   final value
   def execute(frame):
      return value

class LoadGlobal:
   final binding
   def __init__(name):
      binding = lookupGlobal(name)
   def execute(frame):
      return binding.value
```

Implementing execute() methods

```
log("Hello");
```



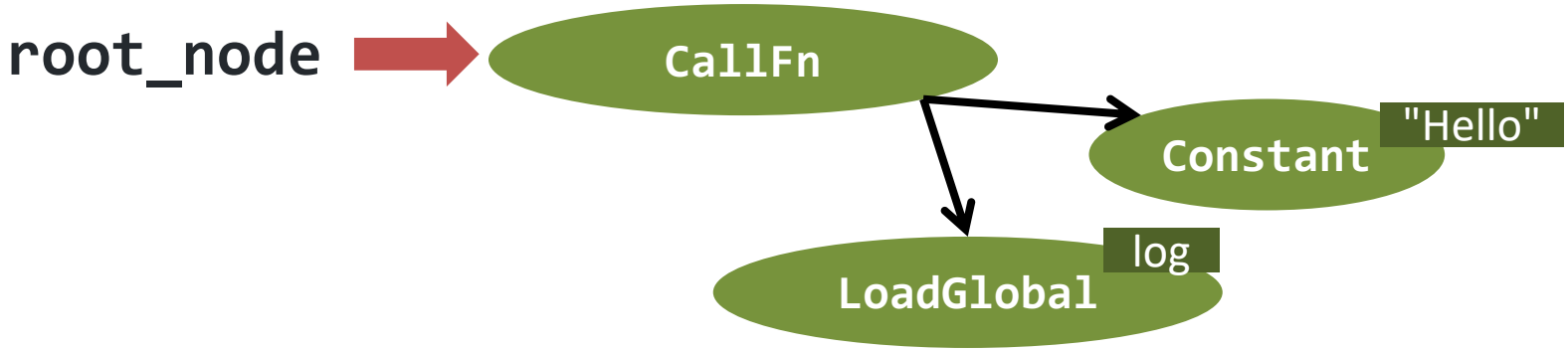

```
class Constant:
    final value
    def execute(frame):
        return value
```

```
class LoadGlobal:
    final binding
    # ...
    def execute(frame):
        return binding.value
```

```
class CallFn:
    node fn
    node argument
def execute(frame):
    fn = fn.execute(frame)
    arg = argument.execute(frame)
    return call(fn, arg)
```

Abstract Syntax Tree Interpreters

```
log("Hello");
```



Simple and Flexible

Any drawbacks?

Abstract Syntax Tree Interpreters

Benefits

- Very close to the source code
- Conceptually simple
- Makes building them “easy”

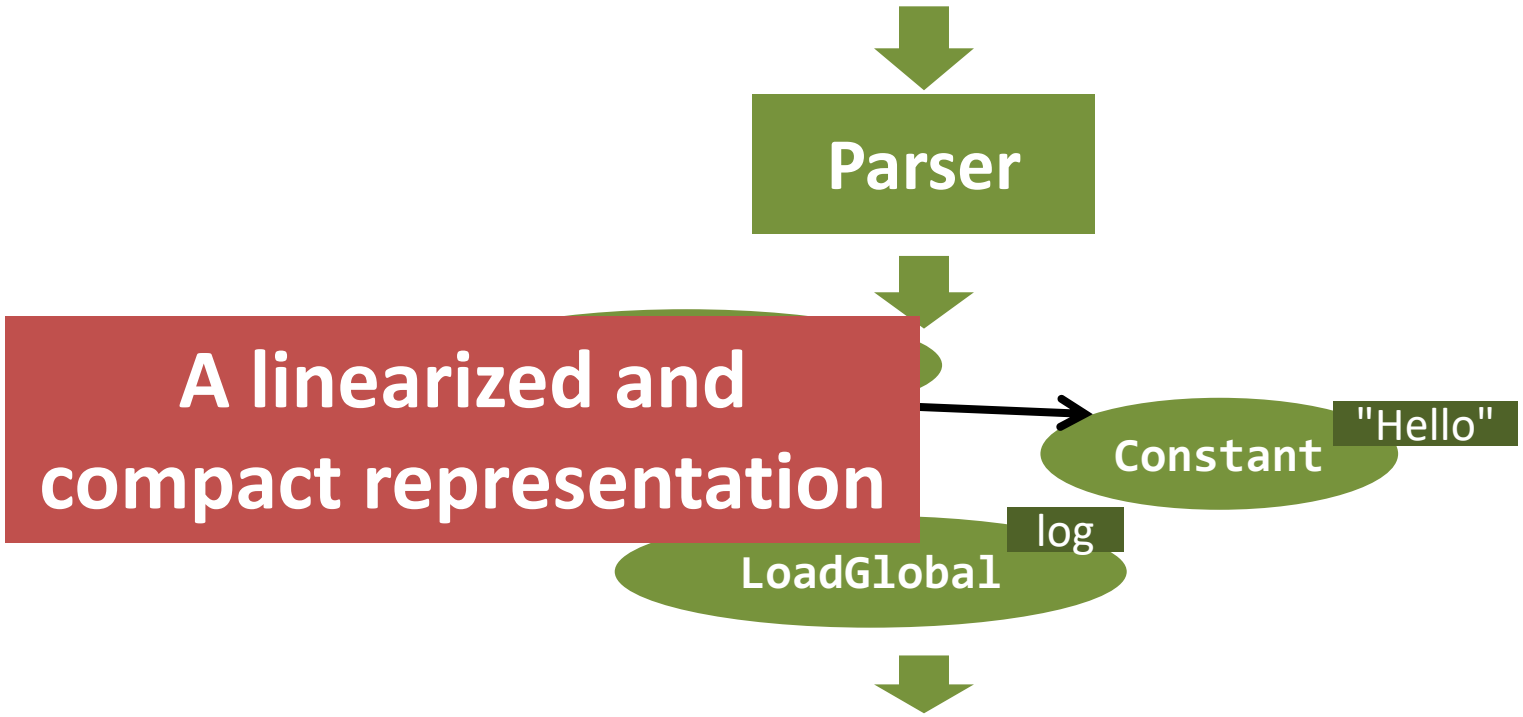
Drawbacks

- A tree node per operation
  - Noncontiguous representation
  - High-ish memory use per node
- Many virtual methods (execution overhead)

BYTECODE INTERPRETERS

From Program to Bytecodes

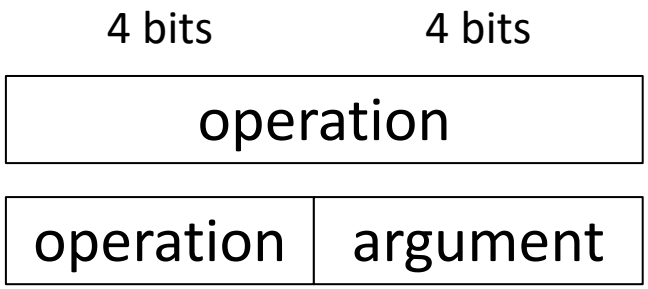
```
log("Hello");
```




```
LoadGlobal "log"
LoadConstant "Hello"
CallFn
```

Bytecode Encoding

- Represent common operations
- In a dense representation



Hypothetical encoding options

```
LoadGlobal   "log"
LoadConstant "Hello"
CallFn
```

translates to

```
1100 0000
1110 0001
1000 0000
```

Classic Bytecode Interpreter

```
LoadGlobal "log"
LoadConstant "Hello"
CallFn
```

```
def interpret():
  idx = 0

  while idx < method.length:
    bc = method[idx]
    switch bc:
      case 0b1000_0000: # CallFn
        arg    = frame.pop()
        fn     = frame.pop()
        result = call(fn, arg)
        frame.push(result)
```

Bytecode Interpreters

Benefits

- Compact and contiguous representation
- Traditionally more efficient than AST interpreters
- Bytecode can be more efficient to parse

Drawbacks

- Requires extra design step
  - Bytecode format
  - Machine type
- Removes execution further from source code
  - Need to resolve mapping during debugging

For some languages, mapping is trivial, e.g., Smalltalk

Research and Literature

- Bytecode design
  - Stack vs. Register
    - Shi, Y., Gregg, D., Beatty, A. & Ertl, M. A. (2005). Virtual machine showdown: stack versus registers. *VEE*'05.
  - Top of stack caching
    - Ertl, M. A. (1995). Stack Caching for Interpreters. *PLDI'95.*
- Direct & indirect threaded interpretation
  - Ertl, M. A. & Gregg, D. (2003). The Structure and Performance of Efficient Interpreters.. *J. Instruction-Level Parallelism*, 5.

- Crafting Interpreters, Robert Nystrom https://craftinginterpreters.com/
- Java Virtual Machine specification https://docs.oracle.com/javase/specs/index.html
- Smalltalk's bytecode set http://www.mirandabanda.org/bluebook/bluebook_chapter28.html
- JavaScriptCore Bytecode Design https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/
- V8 Ignition Interpreter design https://v8.dev/docs/ignition

But

AST and Bytecode Interpreters

are still

“Often Slow” Interpreters!

WHY CARE FOR THESE "OFTEN SLOW" INTERPRETERS?

The Problem:

Old assumptions don't hold for today's massive codebases and engineering approaches!

For the last 30 years,
we optimized for
Long-Running
Server Applications

Large Applications, Frequent Changes

> 100 million
lines of Hack/PHP

A new version every
75 minutes[1]

1. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. Guilherme Ottoni, Bin Liu. CGO'21
2. https://shopify.engineering/automatic-deployment-at-shopify
3. https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1
4. https://instagram-engineering.com/let-your-code-type-hint-itself-introducing-open-source-monkeytype-a855c7284881

Large Applications, Frequent Changes

> 100 million lines of Hack/PHP

> 3 million lines of Ruby

A new version every 75 minutes[1]

30-40 times a day (every 36-48min)[2]

1. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. Guilherme Ottoni, Bin Liu. CGO'21
2. https://shopify.engineering/automatic-deployment-at-shopify
3. https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1
4. https://instagram-engineering.com/let-your-code-type-hint-itself-introducing-open-source-monkeytype-a855c7284881

Large Applications, Frequent Changes

> 100 million lines of Hack/PHP

> 3 million lines of Ruby

> million lines of Python[4]

A new version every 75 minutes[1]

30-40 times a day (every 36-48min)[2]

30-50 times a day[3]

1. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. Guilherme Ottoni, Bin Liu. CGO'21
2. https://shopify.engineering/automatic-deployment-at-shopify
3. https://instagram-engineering.com/continuous-deployment-at-instagram-1e18548f01d1
4. https://instagram-engineering.com/let-your-code-type-hint-itself-introducing-open-source-monkeytype-a855c7284881

The Cost of Change

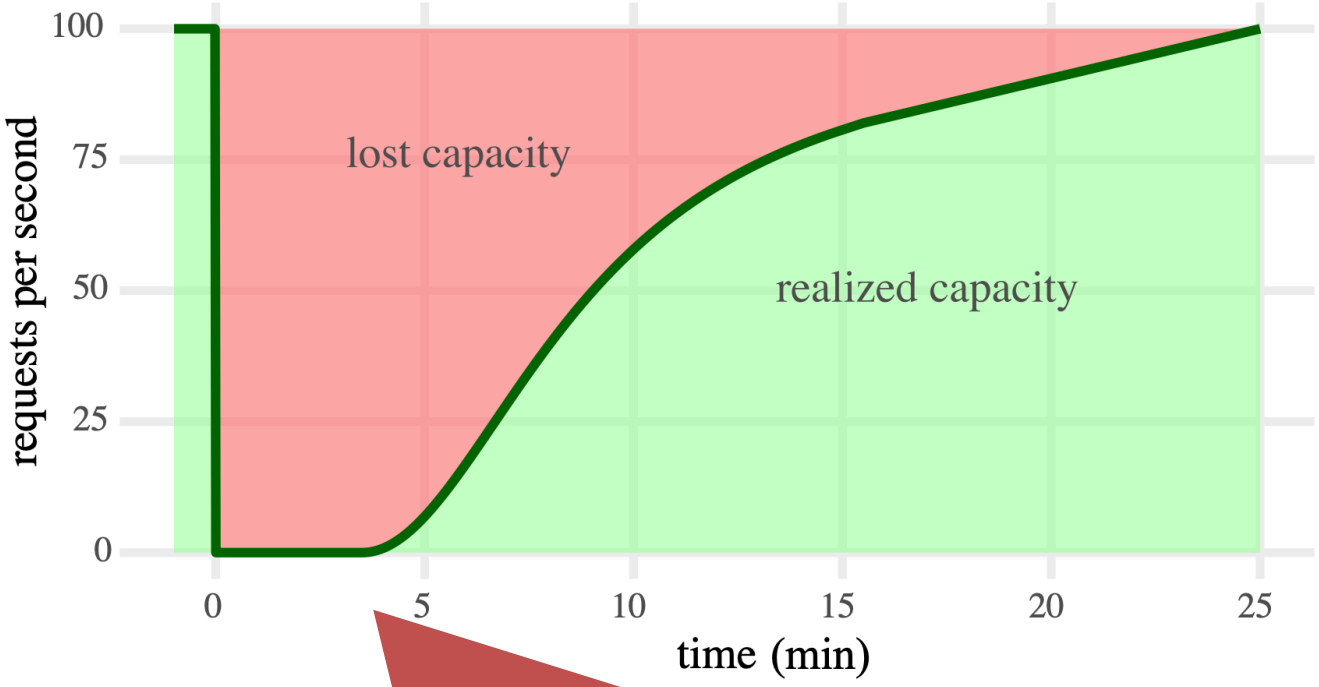


Fig.

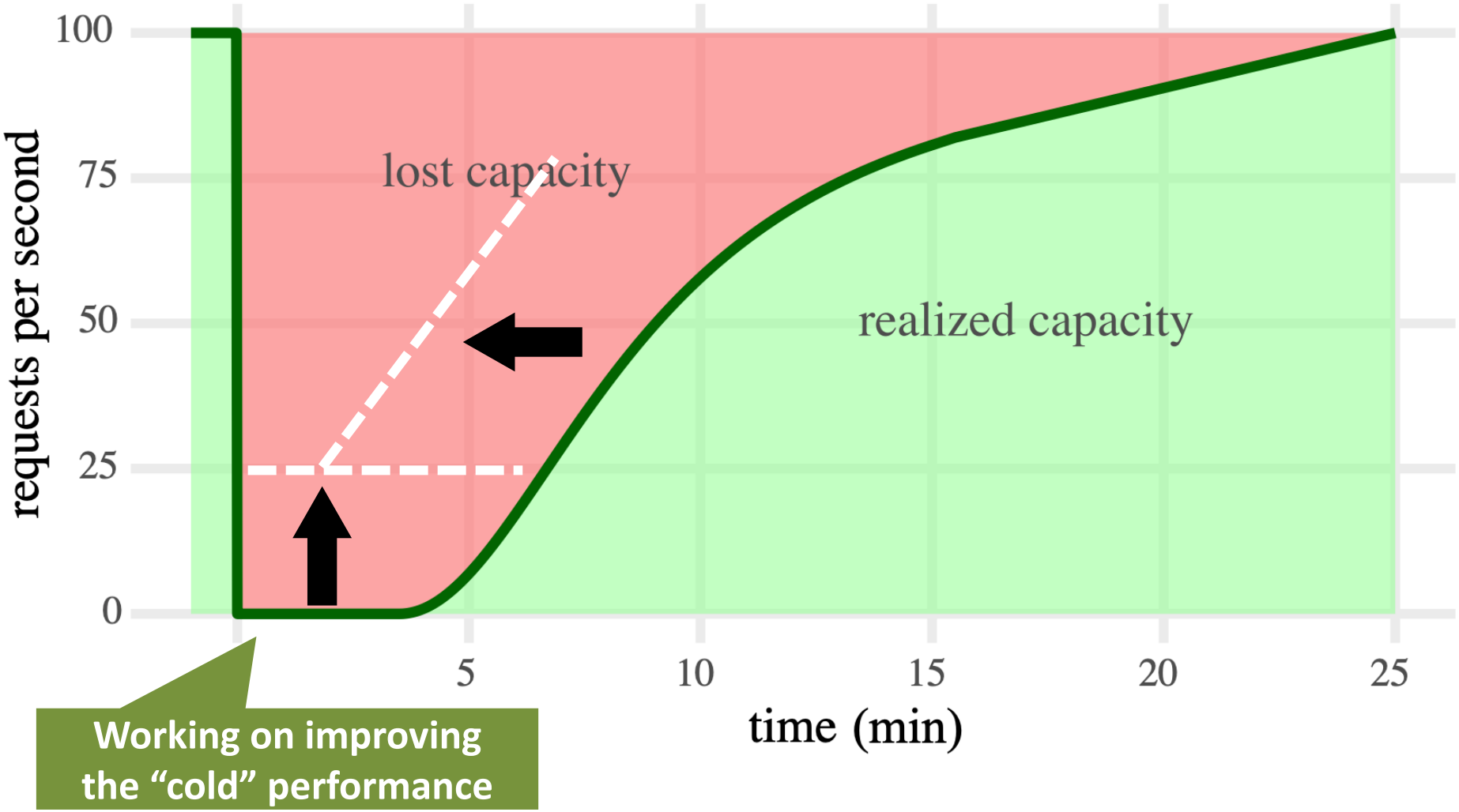
With millions of servers, every update "costs"! (need twice the capacity)

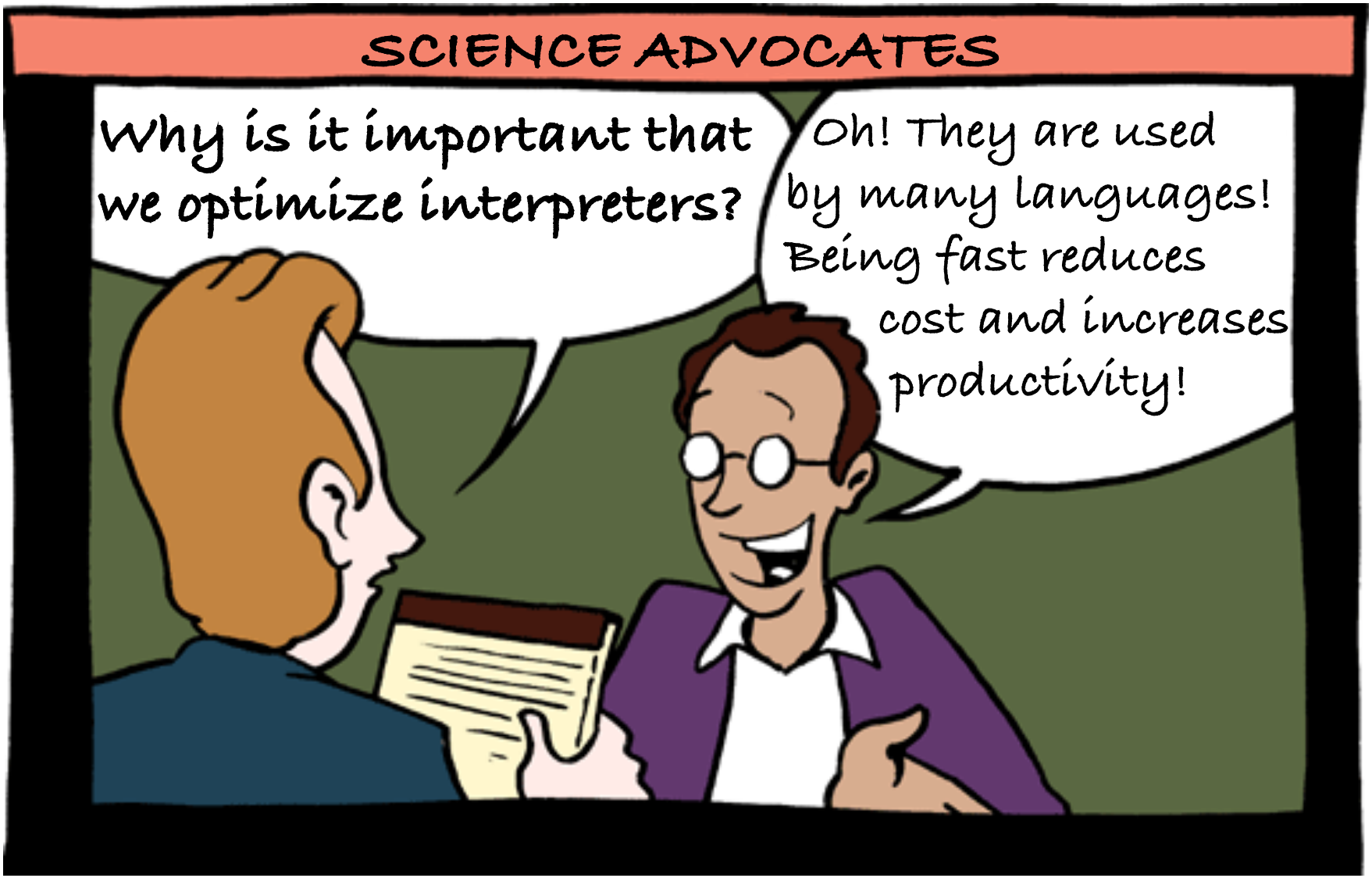
Estimated Energy Bill, in 2020: £500,000,000 (7,170,000 MWh)

HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. Guilherme Ottoni, Bin Liu. CGO'21

https://sustainability.fb.com/report-page/energy/

Building Faster Interpreters





With apologies to SMBC: https://www.smbc-comics.com/?id=2088

And, it's a very hard problem…

With apologies to SMBC: https://www.smbc-comics.com/?id=2088

How Many Interpreters/Compilers are Used During Execution?

- A Calculator Server

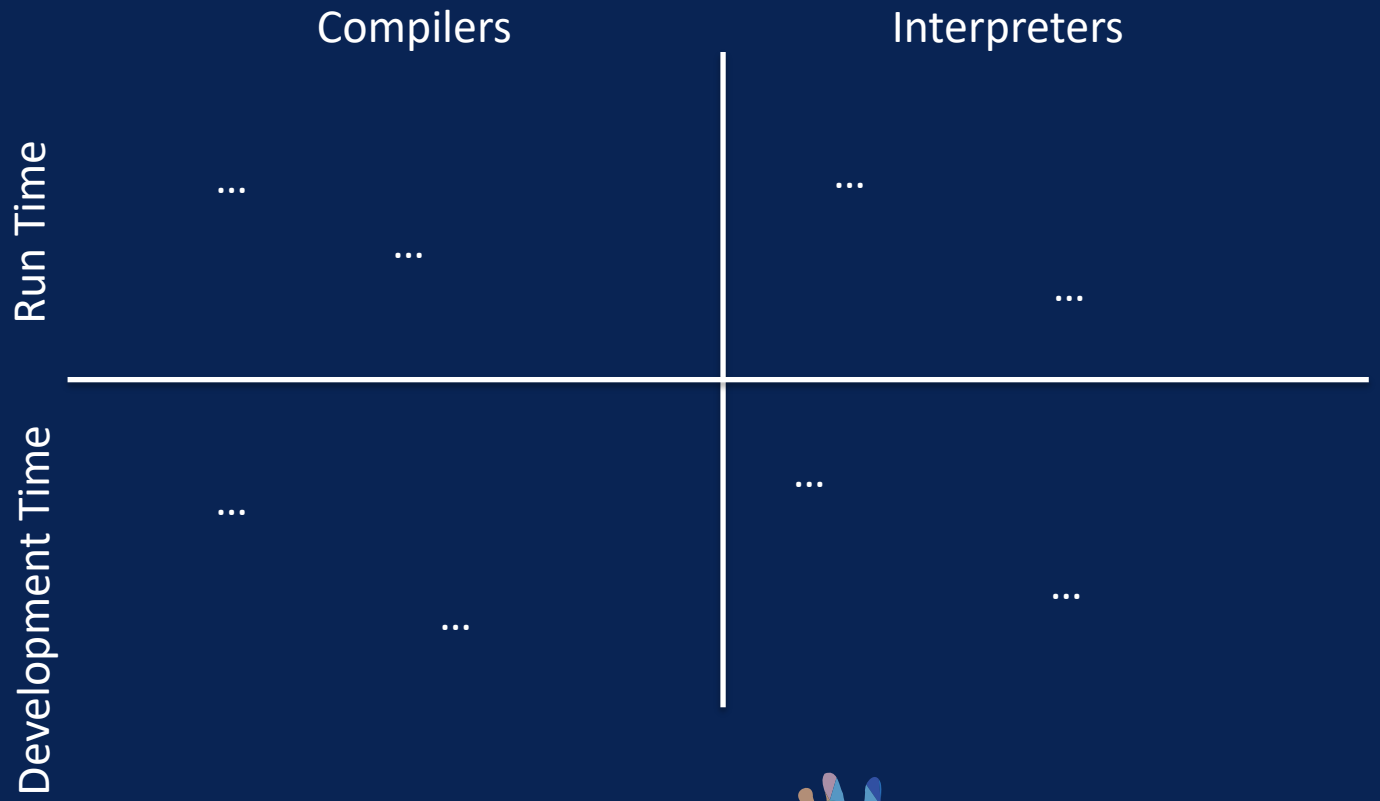
```
{"jsonrpc": "2.0",
 "method": "subtract",
 "params": [42, 23], "id": 1}
```

- Running on top of Node.js
- Running inside QEMU
- On top of a x86-64 processors?

2 4 6 more




Which Interpreters and Compilers are Involved to Create/Run this Stack?



Scenario

- JSON RPC
- Node.js
- QEMU
- x86-64

Interpreters

Everywhere and All The Time!

5min Break

Next up

How can we optimize interpreters?

HOW CAN WE OPTIMIZE INTERPRETERS?

LOOKUP CACHING

A Class Hierarchy Example

```
class Widget {
  fitsInto(width) {
    return this.width <= width;
  }
}
class Button extends Widget {}
class RadioButton extends Button {}

function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

What does this program do?

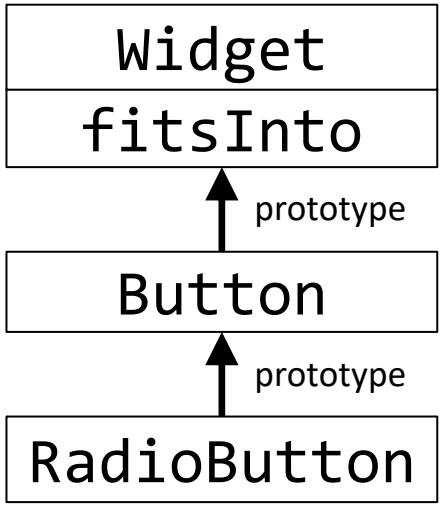
Lookups can be frequent and costly

```
class Widget {
  fitsInto(width) {
    return this.width <=
  }
}
class Button extends Widget {}
class RadioButton extends Button {}

function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

For each `fitsInto` call

hasProperty: 3x

getPrototype: 2x



Solution: Lookup Caching

could be various functions,
but we don't need to do the same lookup repeatedly

```
w.fitsInto(width)
```

isButton function

isLabel function

(in case we see `widget` of different "class")

For AST: chain of nodes
For bytecode: a side table

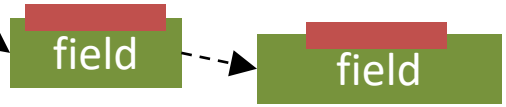
caches
a guard check + lookup result

Any other language features we could use this for?

Useful when lookup is expensive, and essential for JIT compilation

Lookup Caching Field Access

```
w.width = width   // write
width   = w.width // read
```



e.g. for languages where one can add/remove fields

Inlining Trivial Methods

```
class Widget {
  #height;
  getHeight() {
    return this.height;
  }
  getAnswer() {
    return 42;
  }
}
const b = new Butto
b.getHeight()
b.getAnswer()
```

Extra benefit: avoid call overhead

Already mentioned in the ‘91 paper…

Lookup Cache

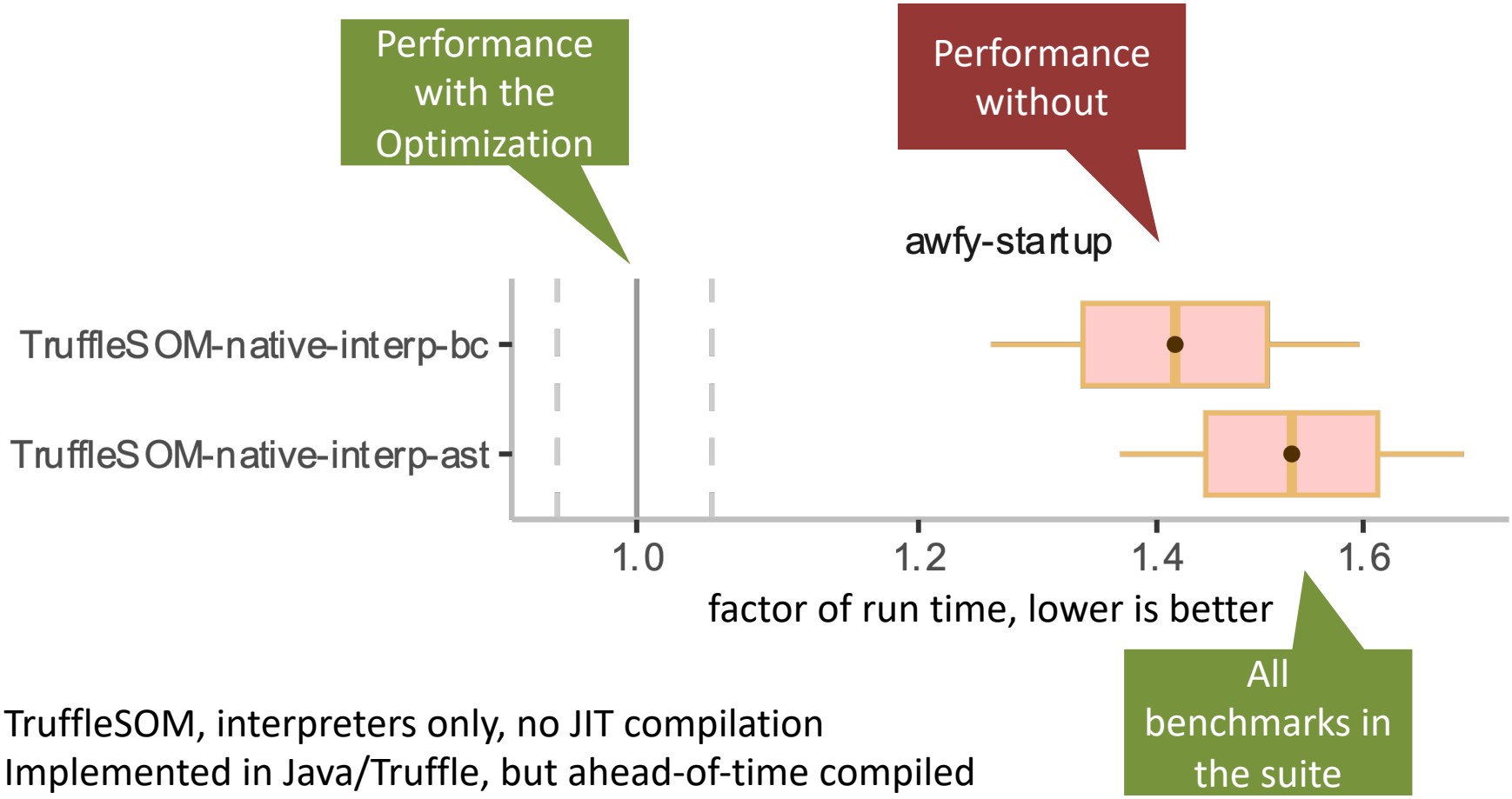
- Can be used for
  - Method/function lookup
  - Field access
  - Getters, setters, method returning constants, other “trivial” methods
  - Object creation when `new` is a method
  - Reflective operations, caches can be nested…

Disclaimer!

Through the lens of Meta-compilation Systems

Graal+Truffle and RPython

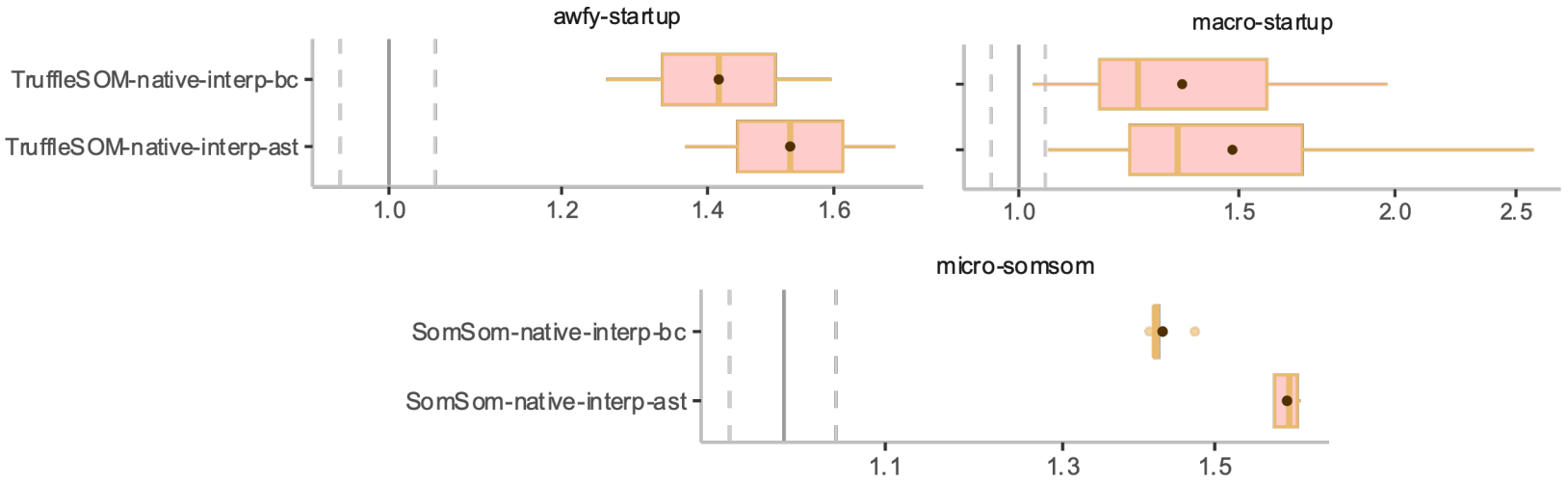
Performance Benefit for the Interpreter



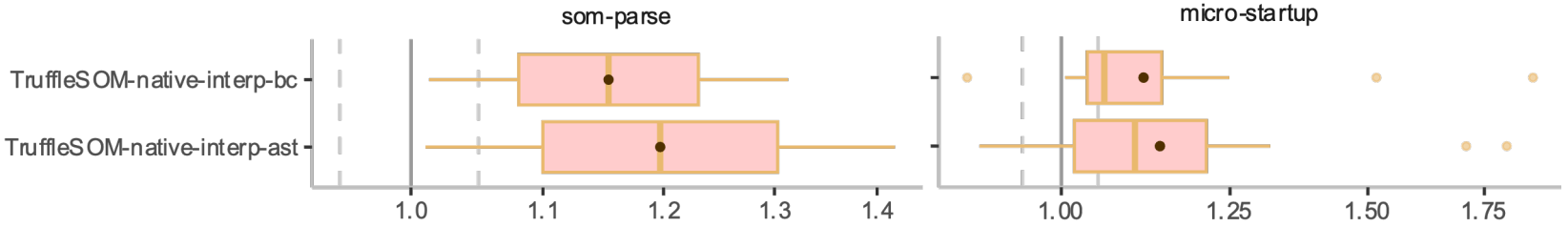
TruffleSOM, interpreters only, no JIT compilation
Implemented in Java/Truffle, but ahead-of-time compiled

https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..b63dff02a25c0e5d72dcd589c6afe38479d05adc

Performance Benefit for the Interpreter



Larger Benchmarks take about 1.5x more time without Lookup Caching



https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..b63dff02a25c0e5d72dcd589c6afe38479d05adc

Performance Benefit, More Details

micro-somsom

Executor: SomSom-native-interp-ast

		#M	median time in ms	time diff %
List		1	20434.63	62
Loop		1	14287.21	63
Mandelbrot		1	321.64	59
Queens		1	12918.73	61
Recurse		1	12934.17	59

https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..b63dff02a25c0e5d72dcd589c6afe38479d05adc

Performance Benefit, More Details

macro-startup

Executor: TruffleSOM-native-interp-ast

		#M	median time in ms	time diff %
DeltaBlue		5	51.62	38
GraphSearch		5	33.50	30
Json		5	78.03	20
NBody		5	80.62	158
PageRank		5	22.68	6
Richards		5	417.26	80

https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..b63dff02a25c0e5d72dcd589c6afe38479d05adc

Research and Literature

- **Efficient Implementation of the Smalltalk-80 System.**
  Deutsch, L. P. & Schiffman, A. M. (1984). *POPL'84*
- **Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches.**
  Hölzle, U., Chambers, C. & Ungar, D. (1991). *ECOOP'91*
- **Zero-Overhead Metaprogramming: Reflection and Metaobject Protocols Fast and without Compromises.**
  Marr, S., Seaton, C. & Ducasse, S. (2015). *PLDI'15*
- **Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications.**
  Kaleba, S., Larose, O., Jones, R., & Marr, S. (2022). *DLS'22*

- Optimizing prototypes in V8
  https://mathiasbynens.be/notes/prototypes
- https://mathiasbynens.be/notes/shapes-ics
- https://mrale.ph/blog/2012/06/03/explaining-js-vms-in-js-inline-caches.html

INLINING OF CONTROL STRUCTURES

Languages where Control Flow is Realized with Methods?

Languages

…

…

…

…

Examples of Methods

…

…

…

…

Let's Go all In on Polymorphic Methods!

```
class True {
  ifTrue(lambda) {
    return lambda.call();
  }
  ifTrueElse(
      trueL, falseL) {
    return trueL.call();
  }
}
```

```
class False {
  ifTrue(lambda) {
    return null;
  }
  ifTrueElse(
      trueL, falseL) {
    return falseL.call();
  }
}
```

```
true.ifTrue(() => {
  // a bit clunky in JS syntax,
  // but works…
});
```

Iterating with Methods

```
class Range {
  each(lambda) {
    let i = this.start;
    while (i < this.end) {
      lambda.call(i);
      i += 1;
    }
    return null;
  }
}

(1..10).each(n => log(n));
```

Can be applied to all control flow

- `if/for/while`
- `each/map/filter`
- `computeIfAbsent`

Inlining the Simple Cases Only

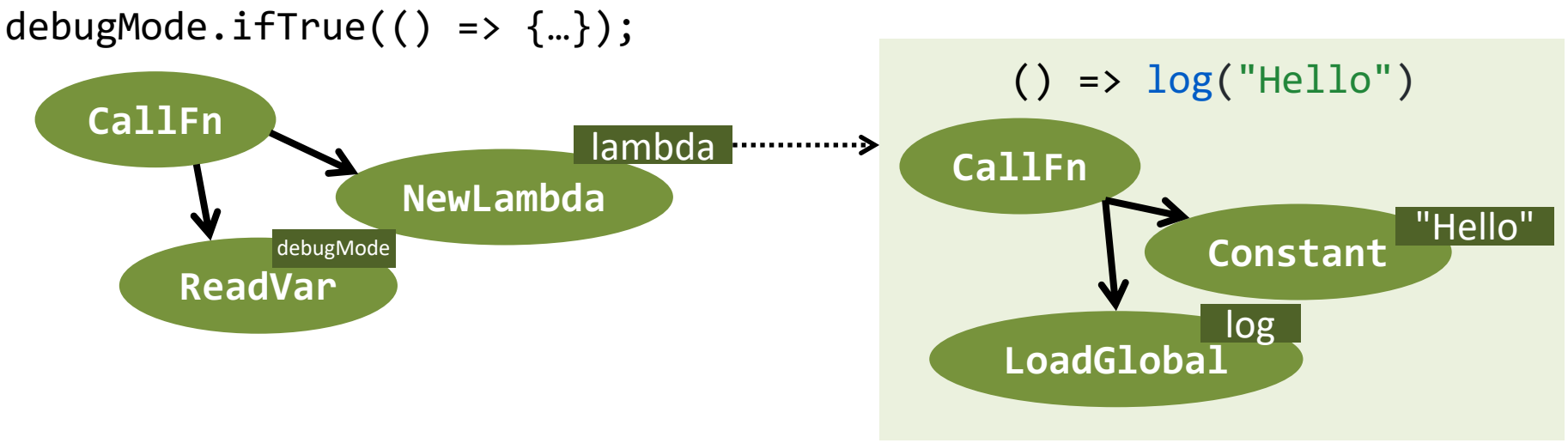
constant lambda, right there in the code

```
debugMode.ifTrue(() => {
  log("Hello");
});

(1..10).each(n => log(n));
```

ensure correctness as we do with lookup caches

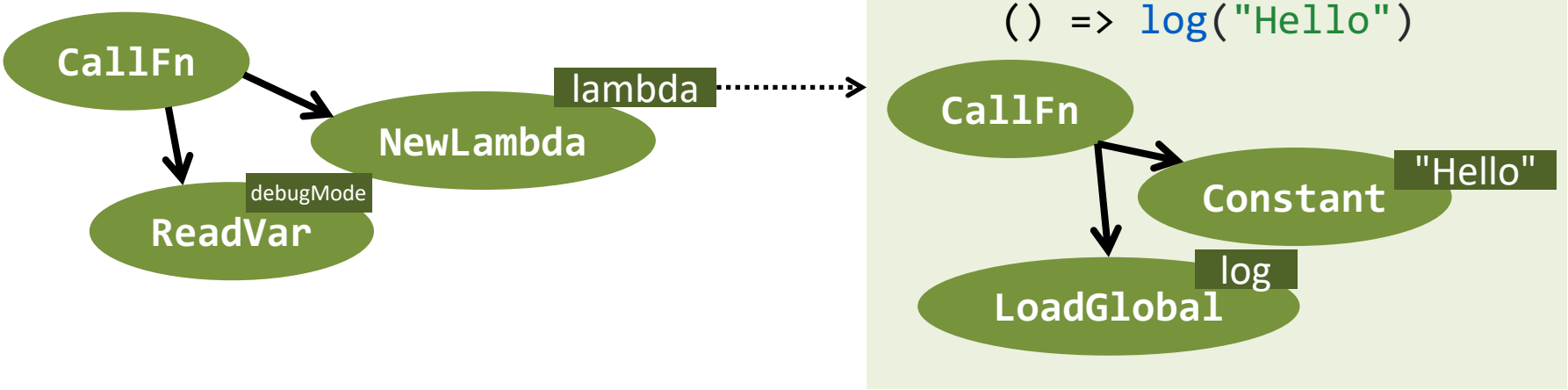
Inlining `ifTrue`

```
debugMode.ifTrue(() => {
  log("Hello");
});
```



Inlining `ifTrue`

```
debugMode.ifTrue(() => {…});
```



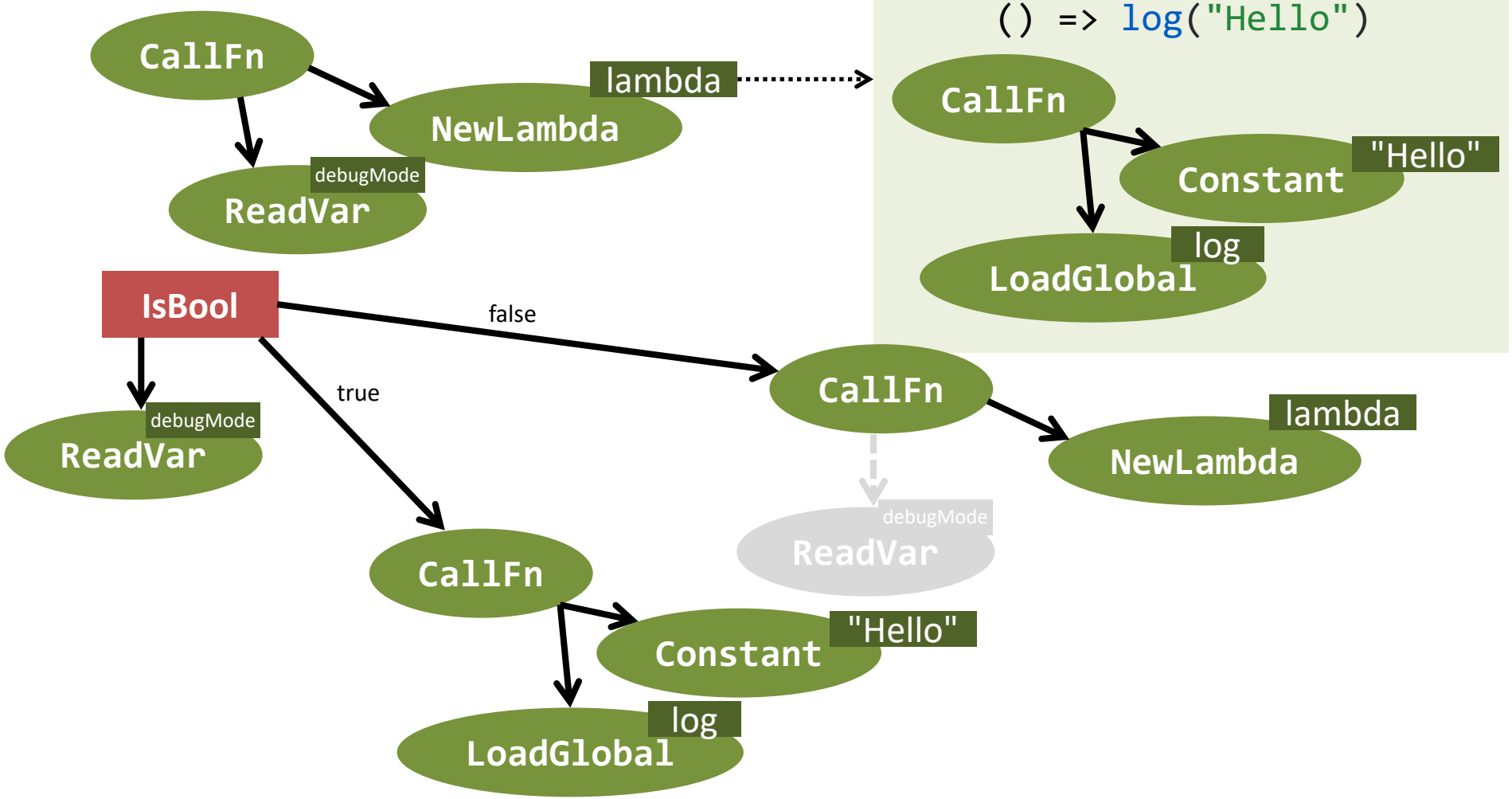
For inlining this, we need to

- Insert the guard
- Keep the old nodes as fallback
- Copy and adapt the body c

Get's more interesting with lexical scopes and variables

Inlining `ifTrue`

```
debugMode.ifTrue(() => {…});
```



Inlining `ifTrue`

```
debugMode.ifTrue(() => {…});
```

```
    LoadVar    debugMode
    NewLambda  lambda
    CallFn     ifTrue
```

```
    () => log("Hello")
```

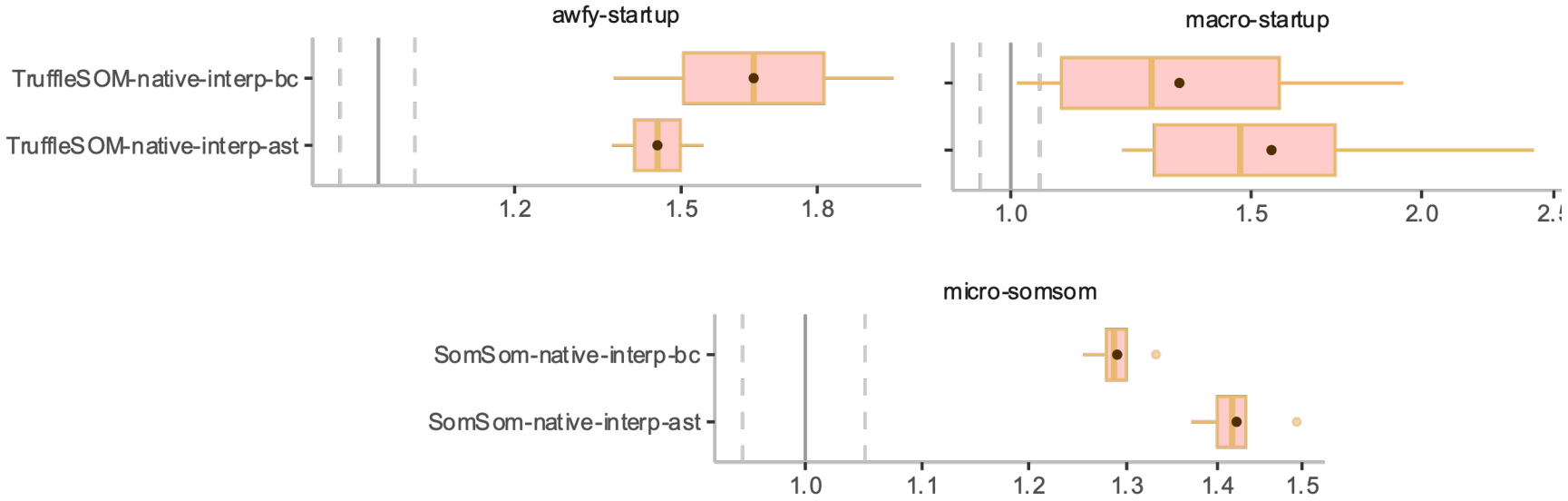
```
    LoadGlobal   "log"
    LoadConstant "Hello"
    CallFn
```

Inlined Version

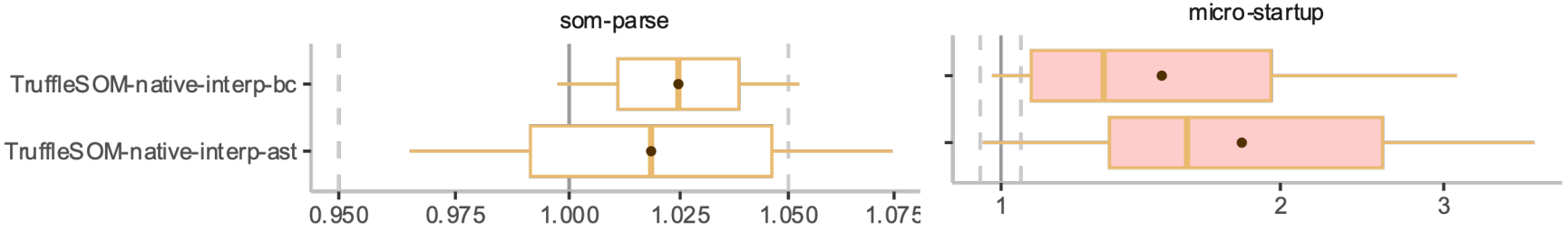
```
    LoadVar    debugMode

    TopIsBool
    JumpIfFalse -> fls

    Pop // remove debugMode from stack

    LoadGlobal   "log"
    LoadConstant "Hello"
    CallFn

fls: NewLambda lambda
    CallFn     ifTrue
```

Looping gets more complicated, but works similarly

Performance Benefit for the Interpreter



Larger Benchmarks take about 1.4x more time without Inlining of Control Structures



https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..9c4d5af762cc61f9e8d79b29528931663813a22b

Some More Details

macro-startup

Executor: TruffleSOM-native-interp-ast

		#M	median time in ms	time diff %
DeltaBlue		5	50.09	34
GraphSearch		5	32.28	25
Json		5	114.52	77
NBody		5	37.66	21
PageRank		5	51.94	142
Richards		5	374.90	62

https://rebench.dev/TruffleSOM/compare/ff89b10bccd49dd0b79933003e3230ae99e0f727..9c4d5af762cc61f9e8d79b29528931663813a22b

micro-startup

Executor: TruffleSOM-native-interp-ast

		#M	median time in ms	time diff %
Bounce		5	61.48	31
BubbleSort		5	69.00	161
Dispatch		5	64.02	62
Fannkuch		5	49.87	124
Fibonacci		5	80.89	55
FieldLoop		5	28.53	54
IntegerLoop		5	41.26	5
List		5	29.91	118
Loop		5	220.85	219
Mandelbrot		5	76.28	201
Permute		5	94.17	24
Queens		5	64.69	50
QuickSort		5	97.02	155
Recurse		5	70.86	53
Sieve		5	101.11	205
Storage		5	57.33	25
Sum		5	109.92	159
Test		10	237.39	6
TestGC		10	113.96	-4
Towers		5	38.65	32
TreeSort		5	87.05	120
WhileLoop		5	107.62	276

SOM SimpleObjectMachine

LIBRARY LOWERING
LIBRARY INTRINSIFICATION

Integer class in SOM

```
class Integer {
    = (argument) primitive
    < (argument) primitive

    <>(argument) { return !(this = argument); }
    > (argument) { return (this >= argument).and(() => this <> argument); }
    >=(argument) { return !(this < argument); }
    <=(argument) { return (this < argument).or(() => this = argument); }
    negative()   { return this < 0; }

    max(otherInt) {
      return (this < otherInt).ifTrueElse(otherInt, this);
    }

    min(otherInt) {
      return (this > otherInt).ifTrueElse(otherInt, this);
    }
}
```

Built into the interpreter

Normal code, part of the standard library

Array class in SOM

```
class Array {
  copy() { return copy(1); }
  copy(start) { return copy(start, this.length); }
  copy(start, end) {
    const result = Array.new(end - start + 1);
    let i = 1.
    (start..end).each((e) => {
      result[i] = e;
      i += 1;
    });
    return result;
  }
}
```

Array.copy, conceptually simple, but implemented by naïve library functions

How to get this to be fast?

Implementing Library Methods in the Virtual Machine

Standard Library

```
class Integer {
    <=(argument) { return (this < argument).or(() => this = argument); }

    negative()   { return this < 0; }

    max(otherInt) {
      return (this < otherInt).ifTrueElse(otherInt, this);
    }
}
```

Implementation in VM

```
class IntegerPrimitives:
  def lessThanOrEqual(a: long, b: long):
    return a <= b

  def negative(a: long):
    return a < 0

  def max(a: long, b: long):
    if a < b:
      return b
    return a
```

- Standard library should be “immutable”
- Changes to standard library won’t show effect
- Debuggers and profilers will show unexpected behavior
- Only works for standard library, optimization impact limited

Any drawbacks of this approach?

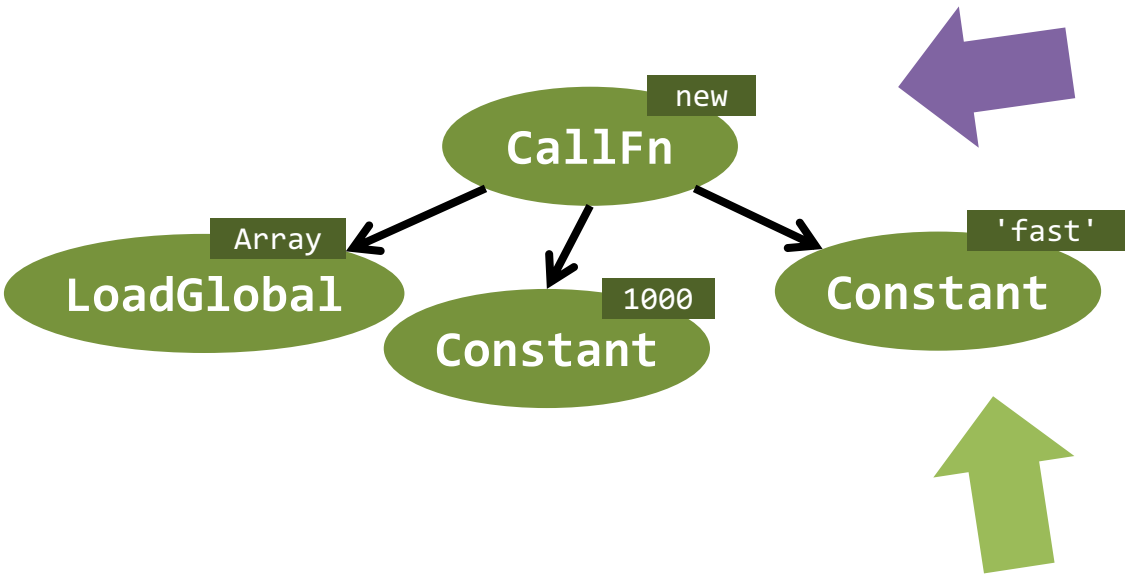
Optimistic and Very Aggressive Library Lowering for Array class

```
createSomeArray() { return Array.new(1000, 'fast fast fast'); }

class Array {
  static new(size, lambda) {
    return new(size).setAll(lambda);
  }

  setAll(lambda) {
    forEach((i, v) -> { this[i] = lambda.eval(); });
  }
}

class Object {
  eval() { return this; }
}
```

Some Object-Oriented Language

Optimizing for Object Values

```
createSomeArray() { return Array.new(1000, 'fast fast fast'); }
```



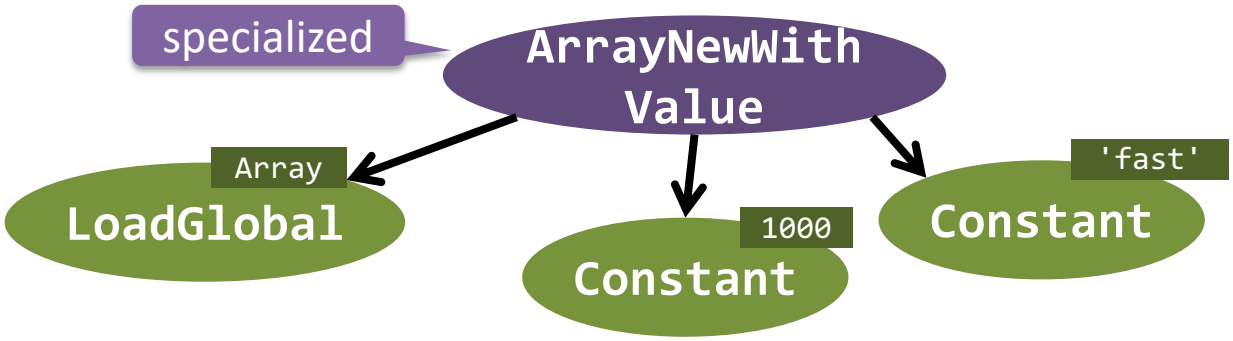
Self-optimizing new(size, lambda)

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }
```

```
def UninitArrNew.execute(frame):
  size := size_expr.execute(frame)
  val  := val_expr.execute(frame)
  return specialize(size, val).
    execute_evaluated(frame, size, val)

def UninitArrNew.specialize(size, val):
  if val instanceof Lambda:
    return replace(StdMethodInvocation())
  else:
    return replace(ArrNewWithValue())
```

Specialized new(size, lambda)

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }
```

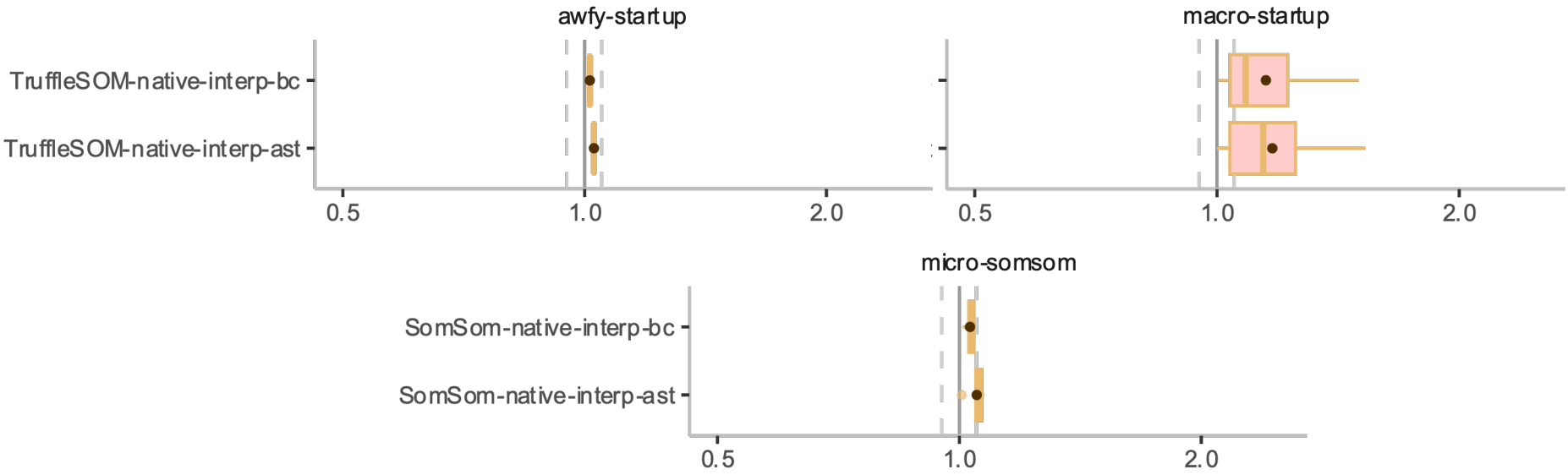



```
def ArrNewWithValue.execute_evaluated(frame, size,
val):
  return Array([val] * 1000)
```

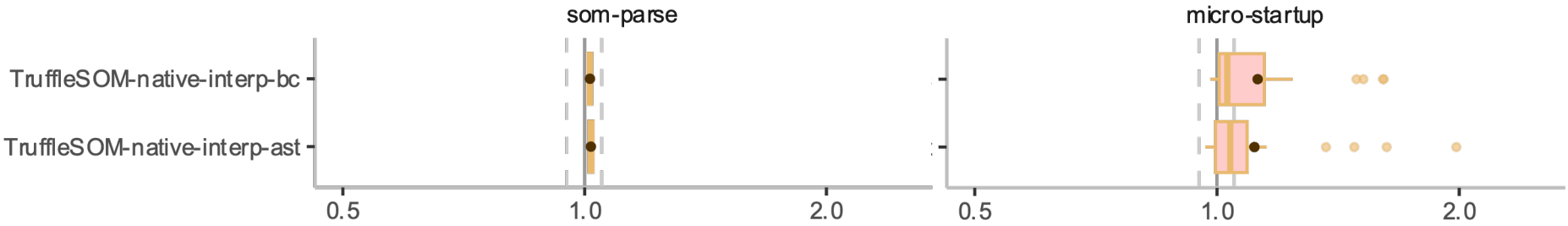
1 specialized node vs.

```
1000x `this[i] = lambda.eval()`
1000x `eval() { return this; }`
```

Performance Benefit



Reaching the point where optimizations are very specific.
But still great for some benchmarks!



https://rebench.dev/TruffleSOM/compare/df38363920b732ca62d81ec7f816f66c2531fa21..b7354a765662d883af9eeb880947a2cc482c7a49

macro-startup

Executor: TruffleSOM-native-interp-ast

		#M	median time in ms	time diff %
DeltaBlue		5	37.97	3
GraphSearch		5	32.27	26
Json		5	98.46	53
NBody		5	33.10	7
PageRank		5	26.71	22
Richards		5	229.73	0

micro-startup

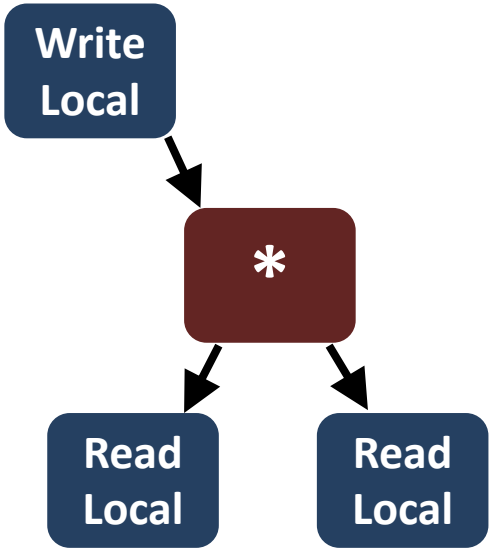
Executor: TruffleSOM-native-interp-ast

		#M	median time in ms	time diff %
Bounce		5	64.21	37
BubbleSort		5	27.79	5
Dispatch		5	39.23	-1
Fannkuch		5	43.95	98
Fibonacci		5	52.27	0
FieldLoop		5	18.80	0
IntegerLoop		5	39.74	-1
List		5	14.29	4

SUPERNODES AND SUPERINSTRUCTIONS

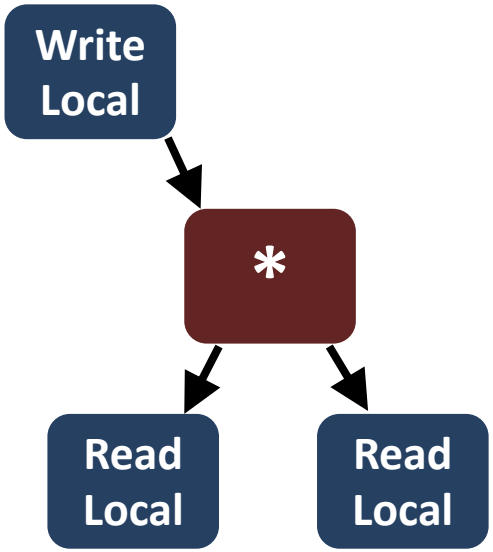
Optimize Code Patterns

```
function mandelbrot() {
  let zr = 0.0;
  // while ... while ...
    zrzr = zr * zr;
  // ...
}
```

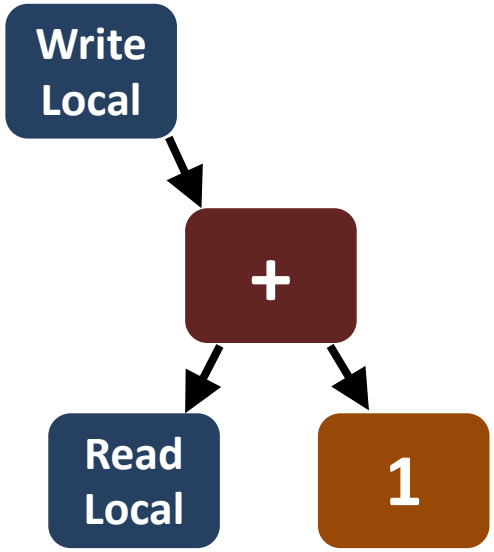



```
class ReadLocal(Node):
  readonly slot_index
  def execute(frame) {
    return frame.read(slot_index)
```

Applies to Various Node Patterns

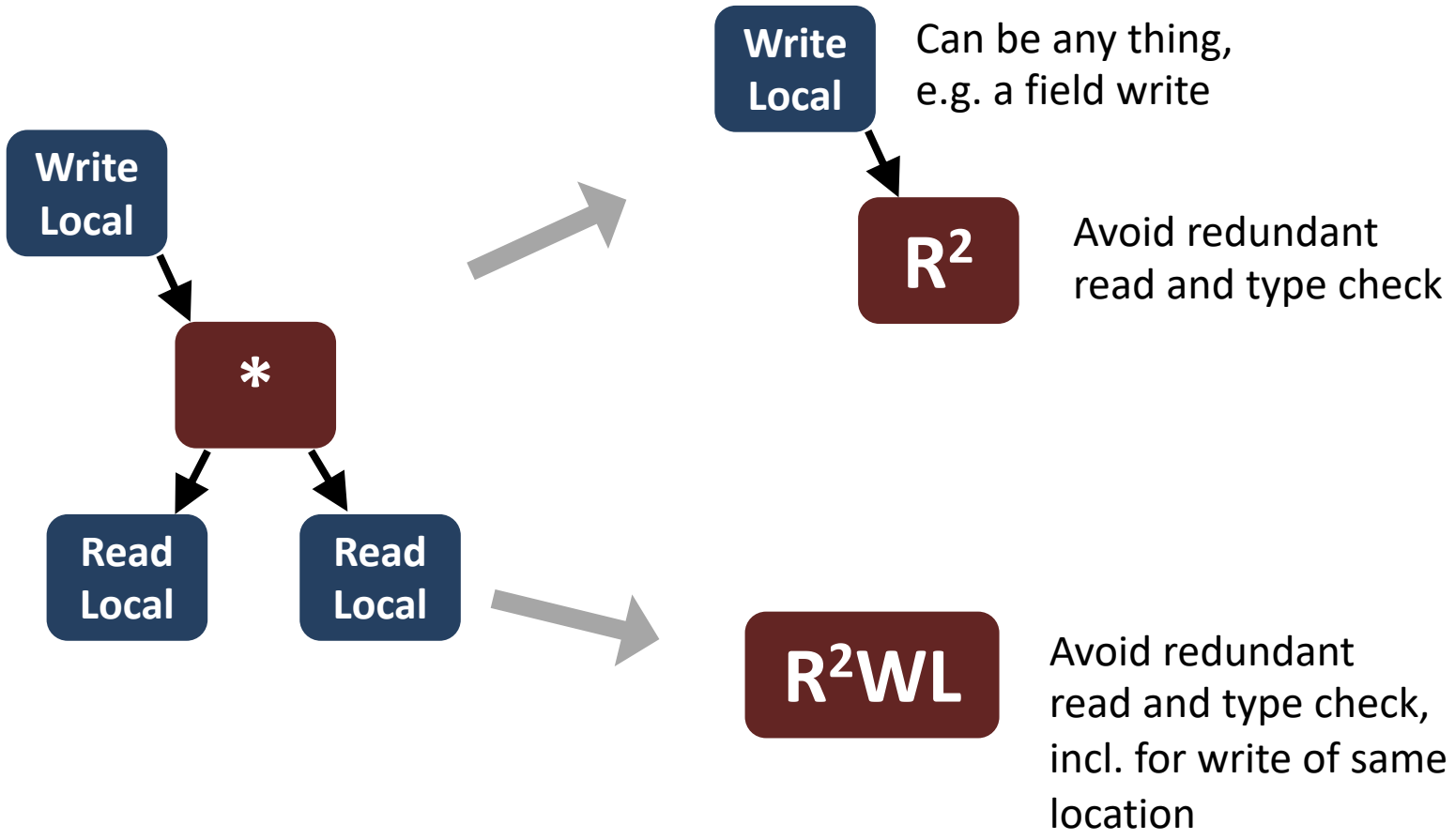



```
zrzr = zr * zr
```




```
i = i + 1
```

Create Different Square Super Nodes for Different Situations



18 Super Nodes in Total

Variations of

- Method activation with `this` as argument
  - `this.method(arg1, arg2)`
- Increment/decrement operations
  - `var += n`
- Compute square
  - `var = d * d`
- String equality with constant
  - `var == 'constant'`

Very “focused” experiment

Results for Super Nodes

	median time in ms	time diff %
DeltaBlue	39.53	-2
GraphSearch	27.48	-3
Json	58.84	-14
NBody	33.62	-5
PageRank	24.94	-11
Richards	250.72	-11

- Optimizing small patterns can give a nice gain
- But not generalizable…

	median time in ms	time diff %
Bounce	47.07	-11
Dispatch	36.34	-19
Fibonacci	49.37	-11
FieldLoop	19.78	-56
Loop	43.69	-56
Mandelbrot	30.52	-26
Permute	77.10	-10
Queens	45.96	-6
QuickSort	41.27	-9
Recurse	43.65	-10
Sieve	34.31	-15
Sum	32.81	-34
Test	184.28	-6
Towers	28.55	-10
WhileLoop	29.98	-29

SOM SimpleObjectMachine

For Bytecodes: Superinstructions

```
i = i + 1
```

bytecode to add 1

bytecode to add 1 to field

```
LoadField    i      LoadField  i       IncField i
LoadConstant 1      Inc
CallFn       +      StoreField i
StoreField   i
```

Research and Literature

- **Optimizing an ANSI C interpreter with superoperators**
  Proebsting, T. A. (1995). *POPL'95*
- **Combining Stack Caching with Dynamic Superinstructions**
  Ertl, M. A. & Gregg, D. (2004). *IVME'04*
- **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters.**
  Casey, K., Ertl, M. A. & Gregg, D. (2007). *TOPLAS'07*
- **Less Is More: Merging AST Nodes To Optimize Interpreters.**
  Larose, O., and Kaleba, S. & Marr, S. (2022). *MoreVMs'23*

Dynamic Languages

```
cnt + 1
```

ECMAScript Specification Sec. 11.6.1

```
left = ToPrimitive(GetValue(cnt))
right = ToPrimitive(GetValue(1))

if (IsString(left) || IsString(right)) {
  return ToString(left).concat(ToString(right))
}

return ToNumber(left) + ToNumber(right)
```

Dynamic Languages

```
cnt + 1
```

ECMAScript Specification Sec. 11.6.1

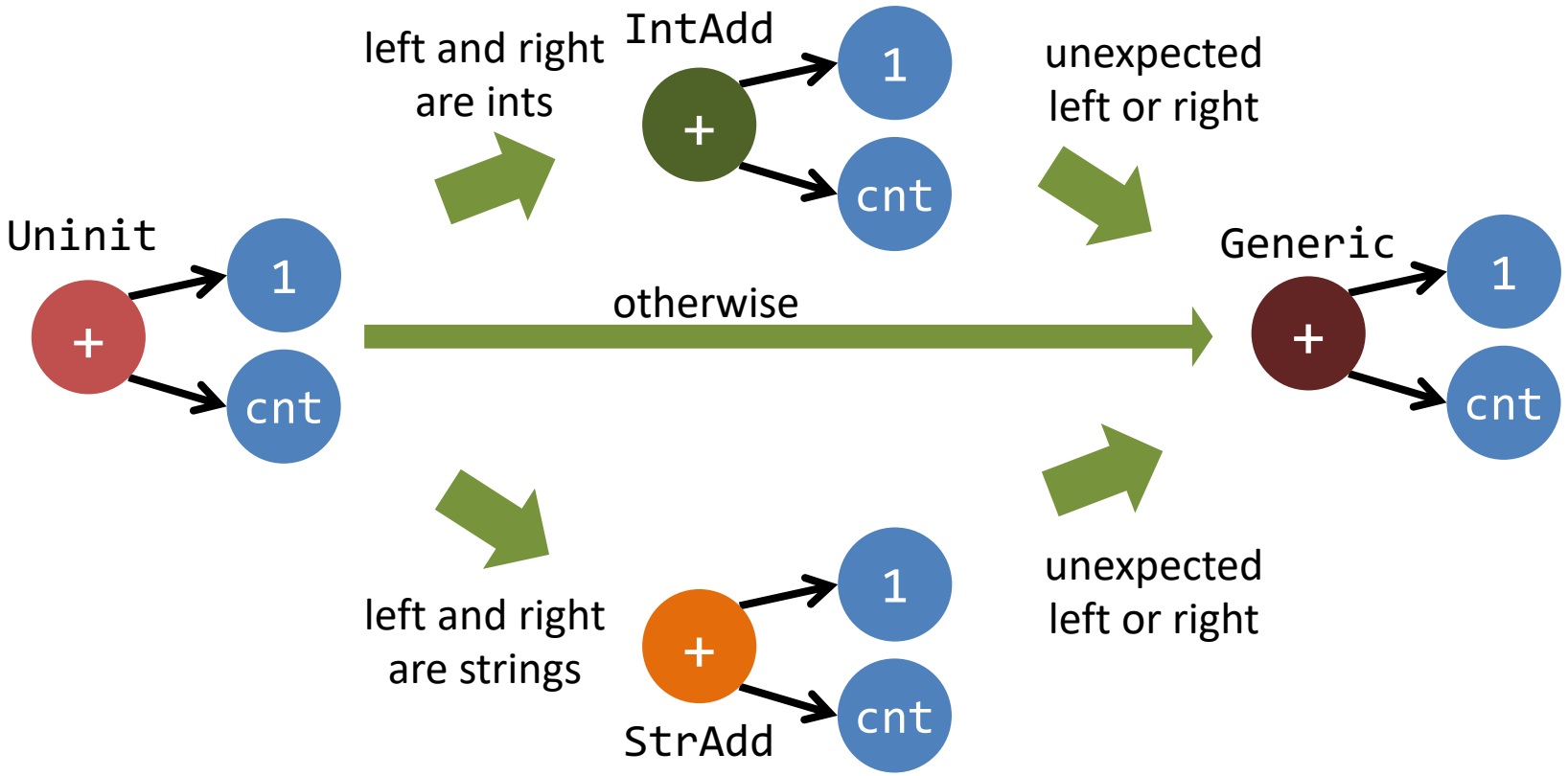
```
left = ToPrimitive(GetValue(cnt))
right = ToPrimitive(GetValue(1))

if (IsString(left) || IsString(right)) {
  return ToString(left).concat
}

return ToNumber(left) + ToNum
```

Could we optimize this somehow?

?

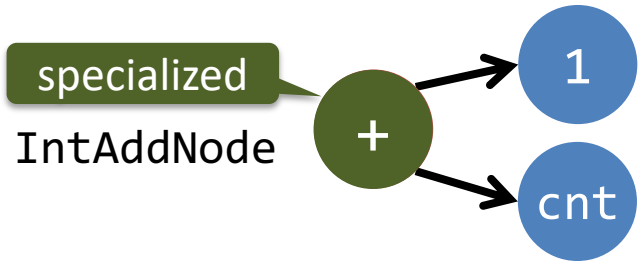
SPECULATIVE OPTIMIZATION

Self Optimization and Speculation



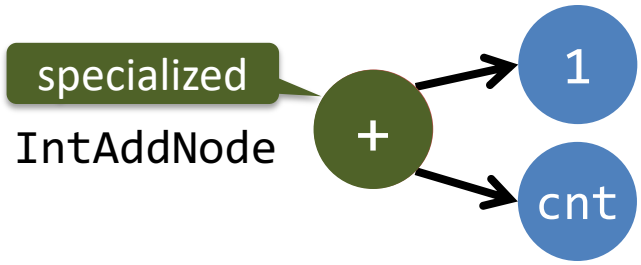
observe run-time value
select optimization of first execution

Self Optimization and Speculation




```
class UninitAdditionNode(BinaryNode):
def execute(frame):
  lVal = left.execute(frame)
  rVal = right.execute(frame)
  if type(lVal) == int and type(rVal) == int:
    return replace(IntAddNode(self)).
                  execute_evaluate(lVal, rVal)
```

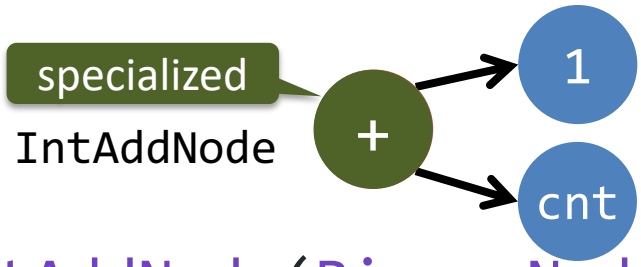
Self Optimization and Speculation




```
class IntAddNode(BinaryNode):
  def execute(frame):
    lVal = left.execute(frame)
    rVal = right.execute(frame)
    if type(lVal) == int and type(rVal) == int:
      return lVal + rVal
    else:
      return deoptimize(lVal,
```

Simplified Version

Self Optimization and Speculation




```
class IntAddNode(BinaryNode):
  def execute(frame):
    try:
      lVal = left.execute_int(frame)
    except UnexpectedResult, exp:
      return deoptimize(exp)
    try:
      rVal = right.execute_i
    except UnexpectedResult,
      return deoptimize(lVal
    return lVal + rVal
```

Check moved down tree, and potentially eliminated

Bytecode Quickening

```
i = i + 1
```

run with `i` being an int

```
LoadField     i
LoadConstant  1
IntInc
StoreField    i
```

```
LoadField     i
LoadConstant  1
CallFn        +
StoreField    i
```

run with `i` being a str

```
LoadField     i
LoadConstant  1
StrAdd
StoreField    i
```

Rewriting constrained by bytecode format!
And the linear format!

Some Possible Self-Optimizations

- Type profiling and specialization

```
cnt + 1
```

- Lookup caching

- Value caching
- Operation inlining

- Library Lowering

Summary

Self Optimization

- For AST interpreters
- Specialize nodes
  - Fast run-time check
  - Avoid complex general case
  - A *local* optimization
- Uses deoptimization to fall back to general case

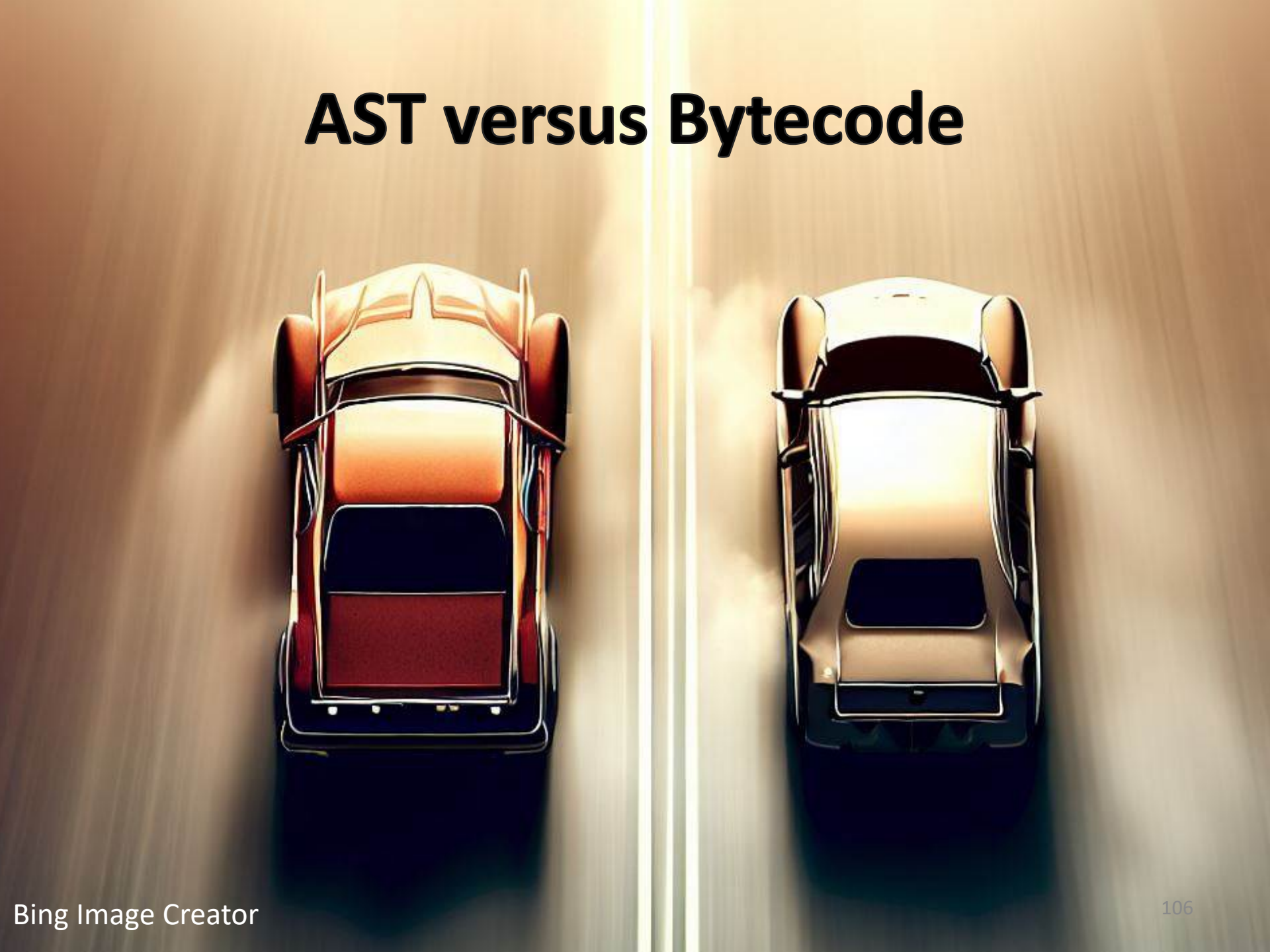
Speculation

- Assume future behavior is same as past behavior
- Requires fallback strategy (deoptimization)
- Used with self optimization, quickening, and JIT compilation

Research and Literature

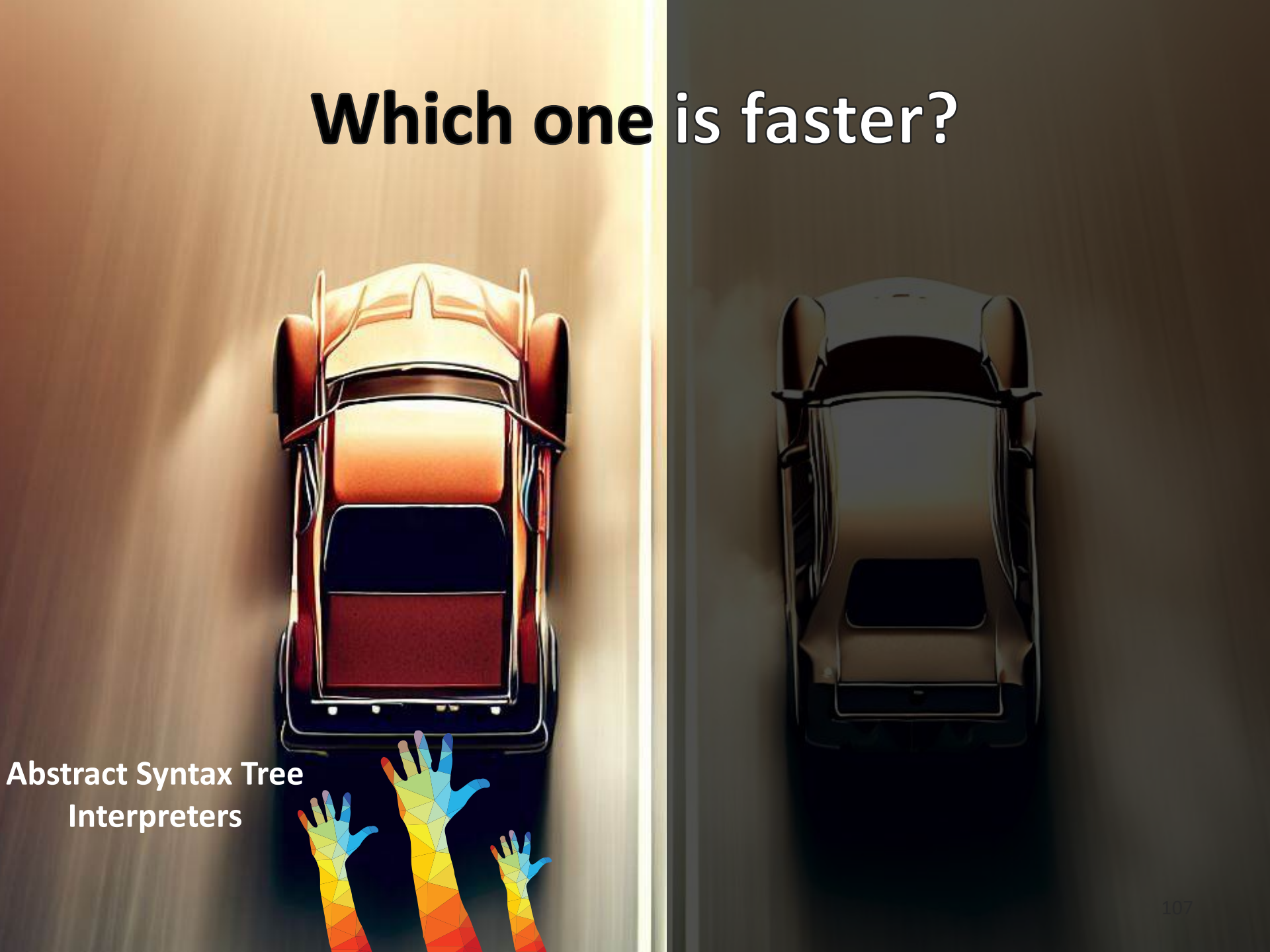
- **AST interpreters**
  - **Self-Optimizing AST Interpreters**
    Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D. & Wimmer, C. (2012). *DLS'12*
- **Bytecode interpreters**
  - **Efficient Interpretation Using Quickening**
    Brunthaler, S. (2010). *DLS'10*
  - **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters**
    Casey, K., Ertl, M. A. & Gregg, D. (2007). *ACM Trans. Program. Lang. Syst.*, 29, 37.
  - **Multi-Level Quickening: Ten Years Later.**
    Brunthaler, S. (2021) arXiv:2109.02958v1
- **JIT compilers, deoptimization**
  - **An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes.**
    Chambers, C., Ungar, D. & Lee, E. (1989). OOPSLA'89

AST versus Bytecode



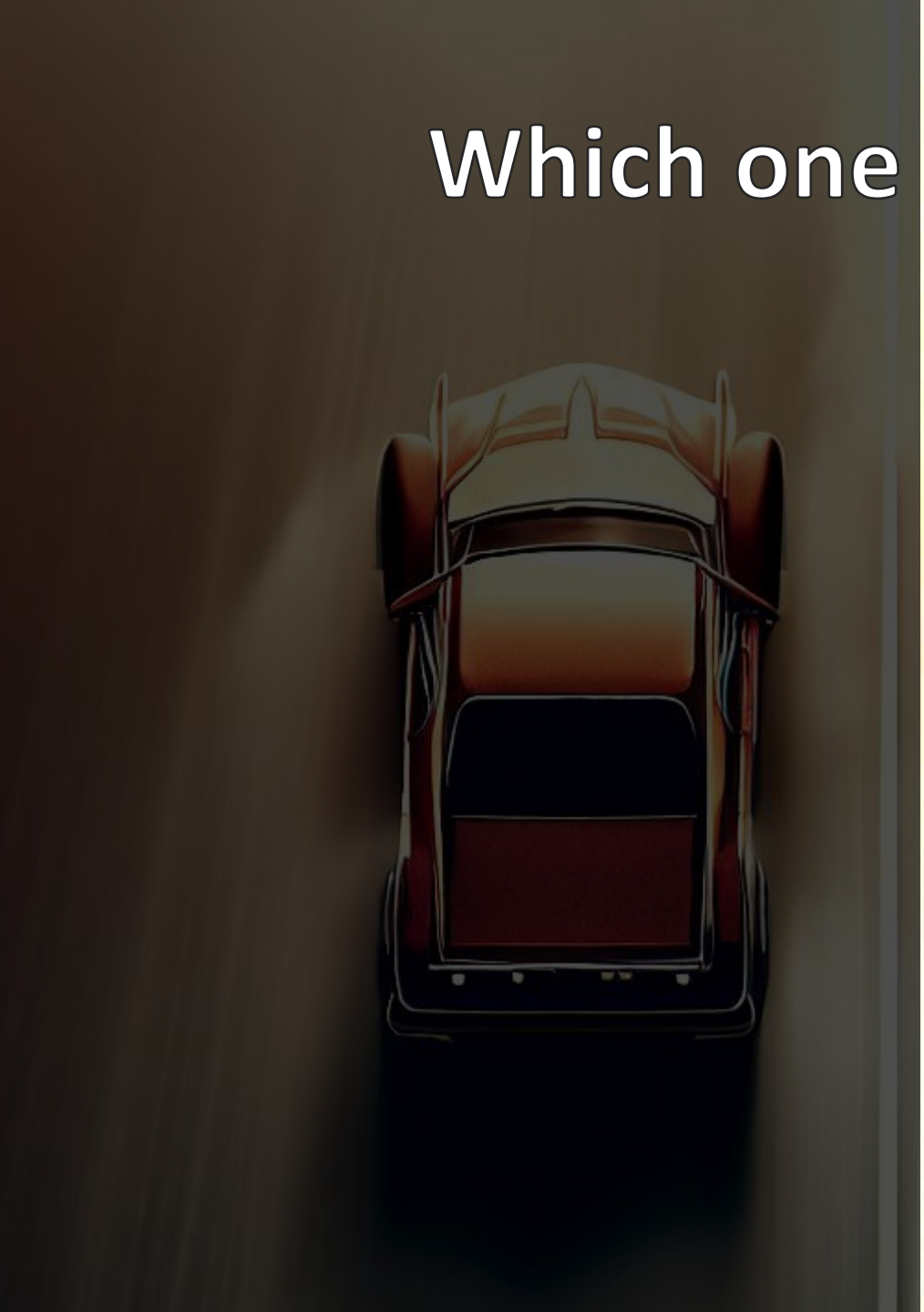
Bing Image Creator

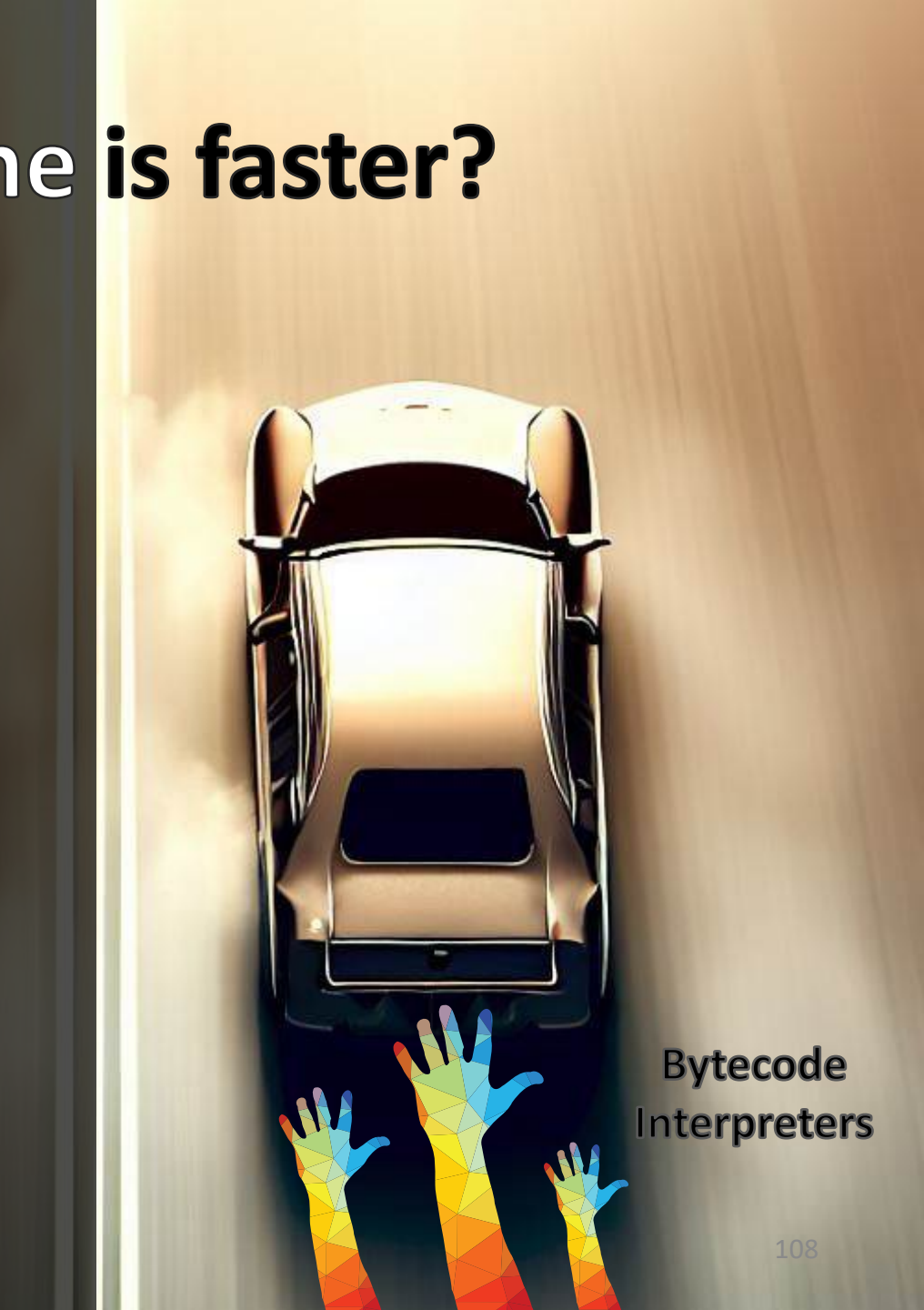
Which one is faster?



Abstract Syntax Tree Interpreters

Which one is faster?





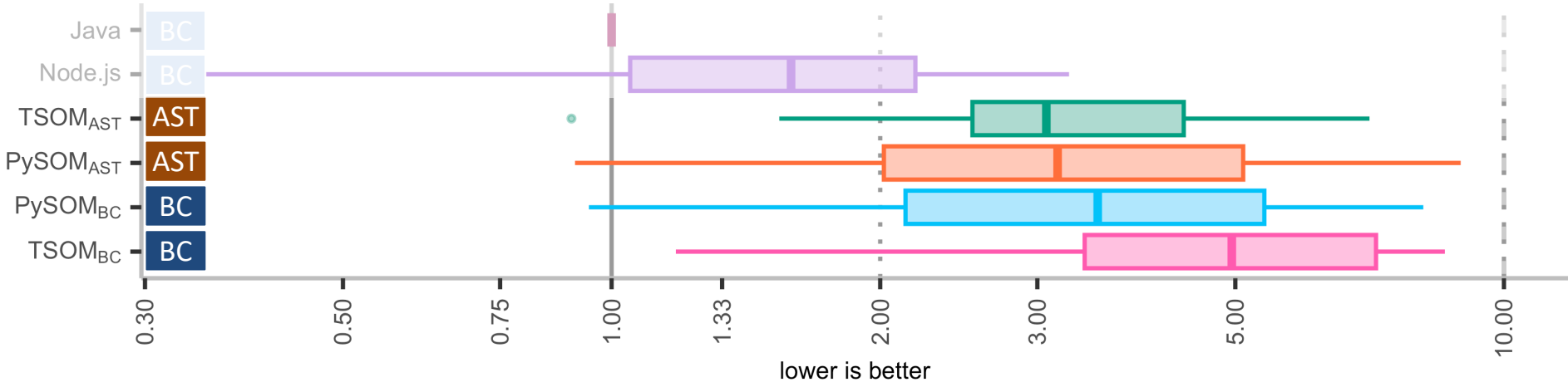
Bytecode Interpreters

Disclaimer!

Through the lens of Meta-compilation Systems

Graal+Truffle and RPython

AST vs Bytecode Interpreters for Metacompilation Systems

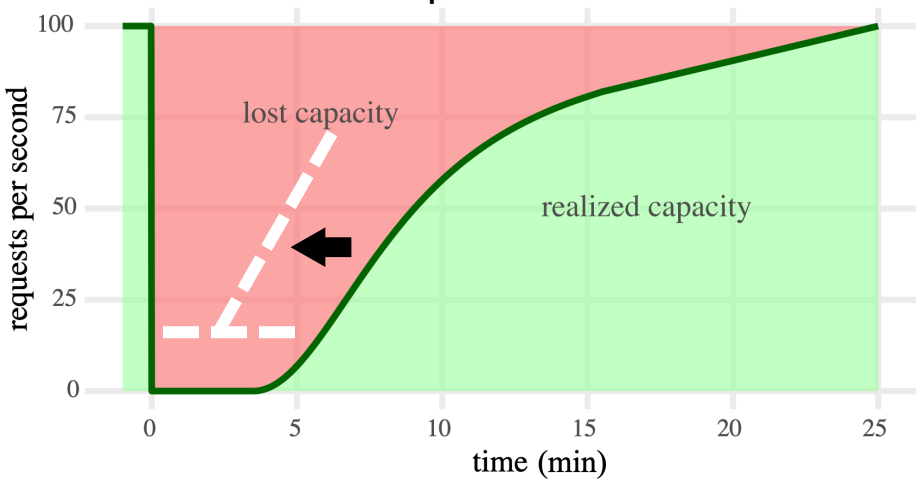


AST interpreters can be surprisingly fast!
Though, bytecodes are much more compact in memory.

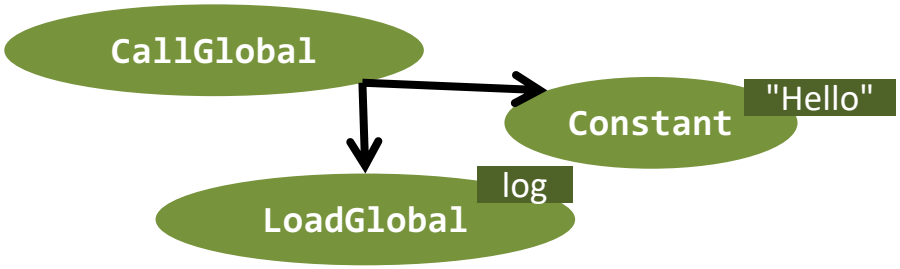
AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. O. Larose, S. Kaleba, H. Burchell, S. Marr. OOPSLA'23. to appear
https://stefan-marr.de/downloads/oopsla23-larose-et-al-ast-vs-bytecode-interpreters-in-the-age-of-meta-compilation.pdf

WARP UP

Interpreter performance is critical for overall performance

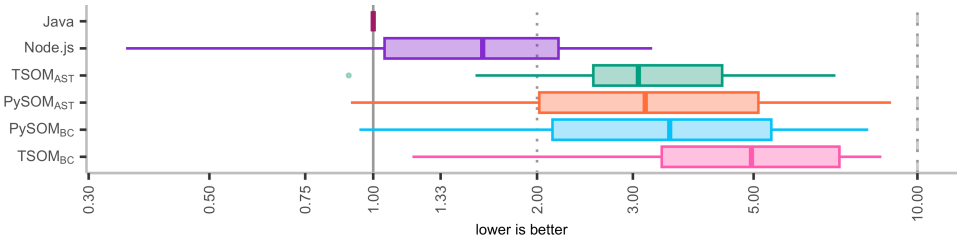


Most are AST or Bytecode Interpreters




```
LoadGlobal    "log"
LoadConstant  "Hello"
CallFn
```

Can reach similar performance (in metacompilation systems)



AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. O. Larose, S. Kaleba, H. Burchell, S. Marr. OOPSLA'23. to appear https://stefan-marr.de/downloads/oopsla23-larose-et-al-ast-vs-bytecode-interpreters-in-the-age-of-meta-compilation.pdf

Lookup Caching is Generic, and Highly Effective

```
w.fitsInto(width)
```

function

Inlining also effective,
but gets complicated quickly

Library Lowering/Intrinsification,
Supernodes/instructions,
… get more and more specific
Should be driven by need!