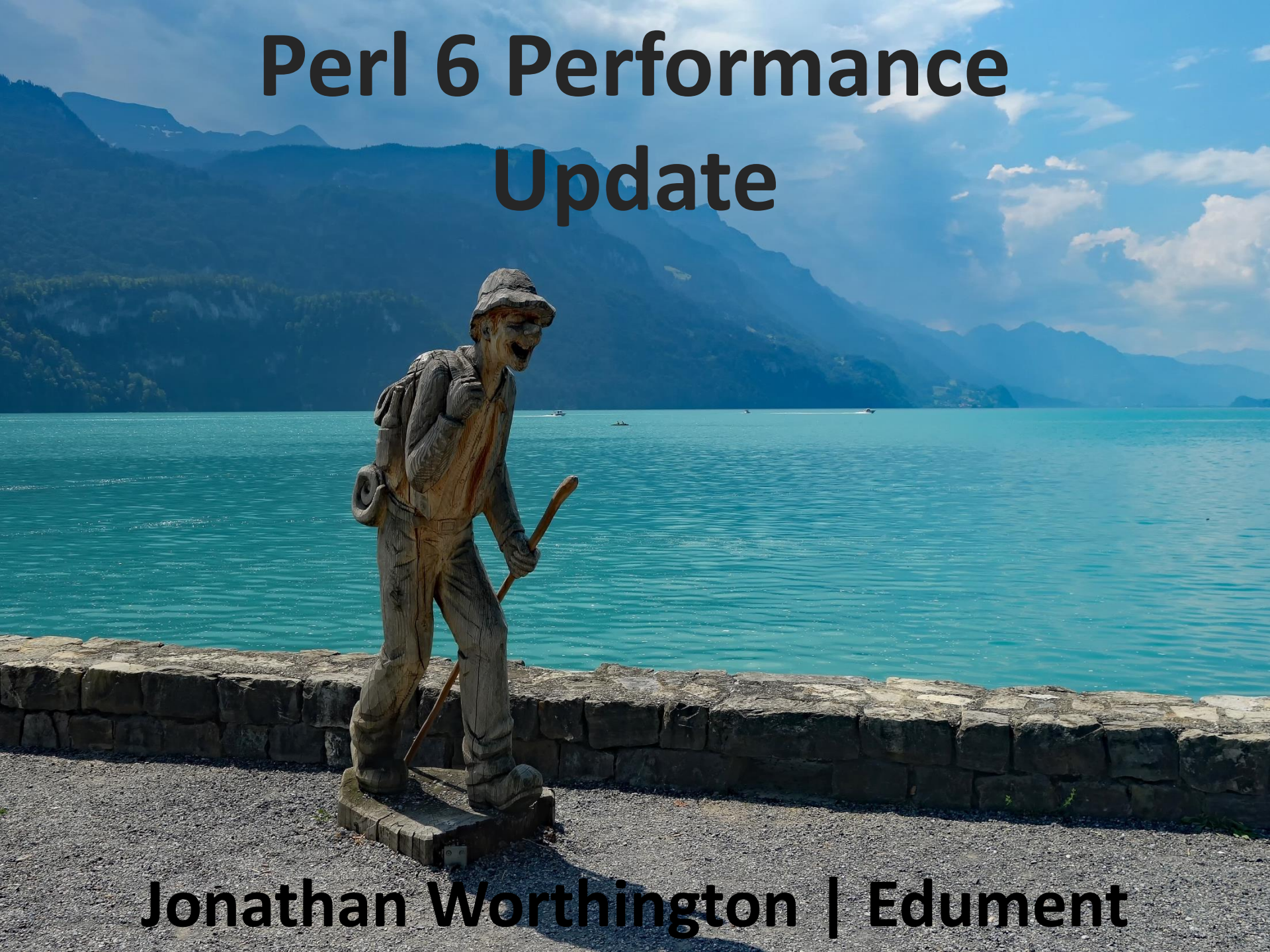
Perl 6 Performance Update



Jonathan Worthington | Edument

The challenge

of running Perl 6 fast

The optimizations

we're performing to rise to them

The results

of various benchmarks

The consequences

for those writing Perl 6 programs today

The plans

for further improvement

The challenge

of running Perl 6 fast

Compiler implemented in Perl 6

Built-ins implemented in Perl 6

Only "native" code is the VM
(MoarVM is written in C)

So to make Perl 6 fast, we must…

So to make Perl 6 fast, we must…

…make Perl 6 fast!

Perl 6 is *very* object-y

Objects ☺

- ✓ Gather together related data and functionality
- ✓ Let us work at a higher level of abstraction
- ✓ Provide polymorphism

Lots of simple things in Perl 6 are objects

Boxes

Int
Num
Str

Containers

Scalar
Array
Hash

Numeric-ish

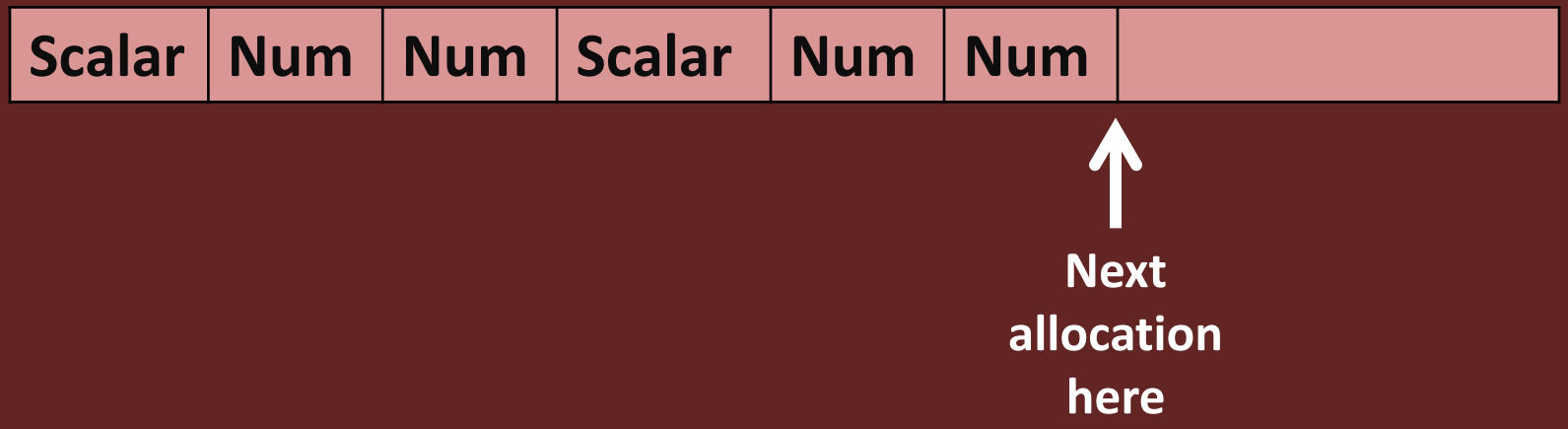
Complex
Date
DateTime
Rat
Range

Objects ☹

- Cost of method resolution
- Allocations mean more memory pressure and more time doing garbage collection
- Harder to analyze/optimize the program

```
for @values -> $v {
    # Allocate a Scalar $sv
    # sin returns a boxed Num
    my $sv = $v.sin;
    # + returns a boxed Num
    do-something(1e0 + $sv);
}
```

Objects are allocated in the GC nursery: a big blob of memory



When it's full, we garbage collect

Obvious consequence:

The quicker we fill the nursery, the more often we have to do GC, and so the more time we spend on GC

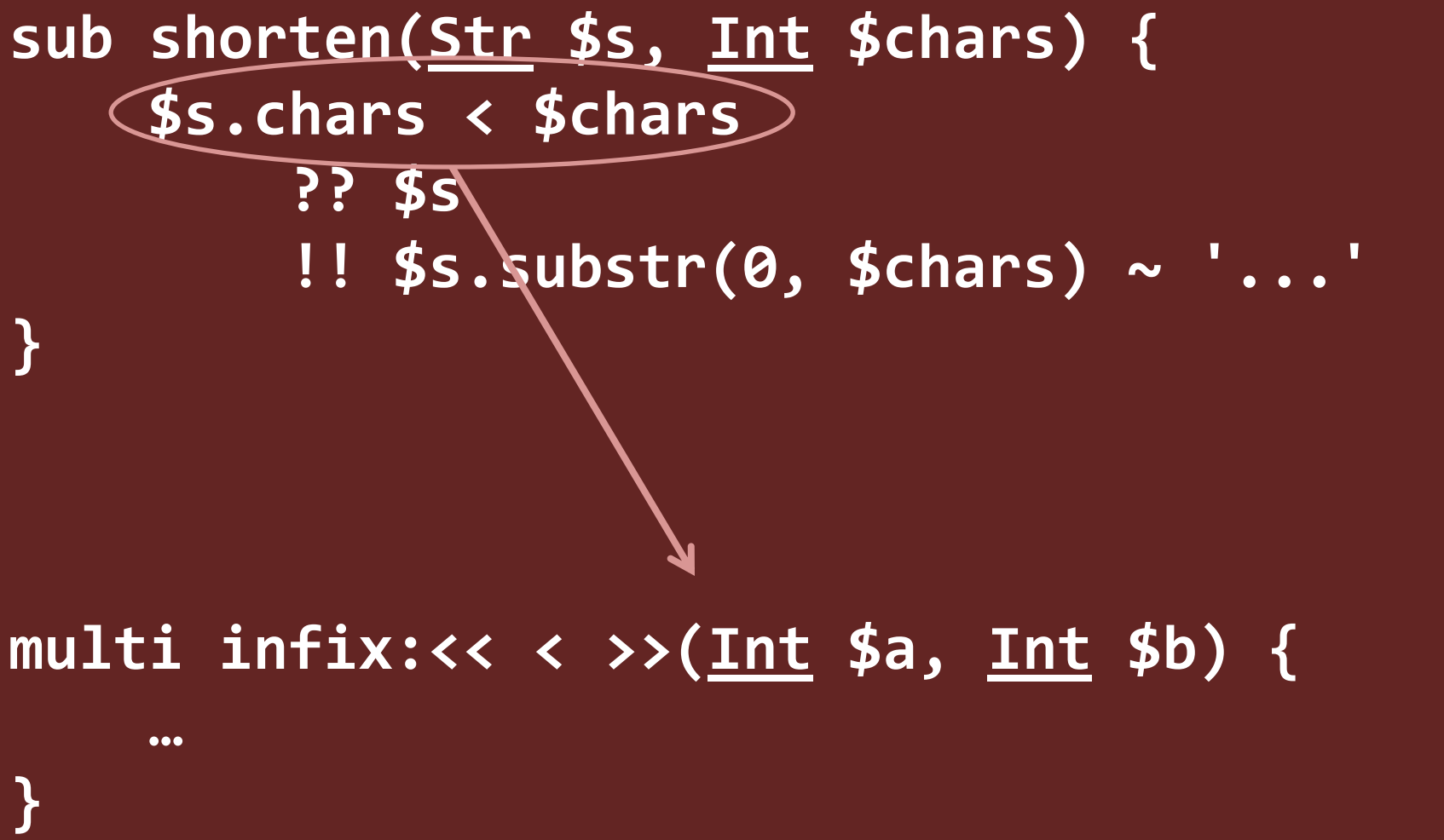
Less obvious consequence:

Objects are spread through memory, so we get lots of CPU cache misses

Perl 6 has types…

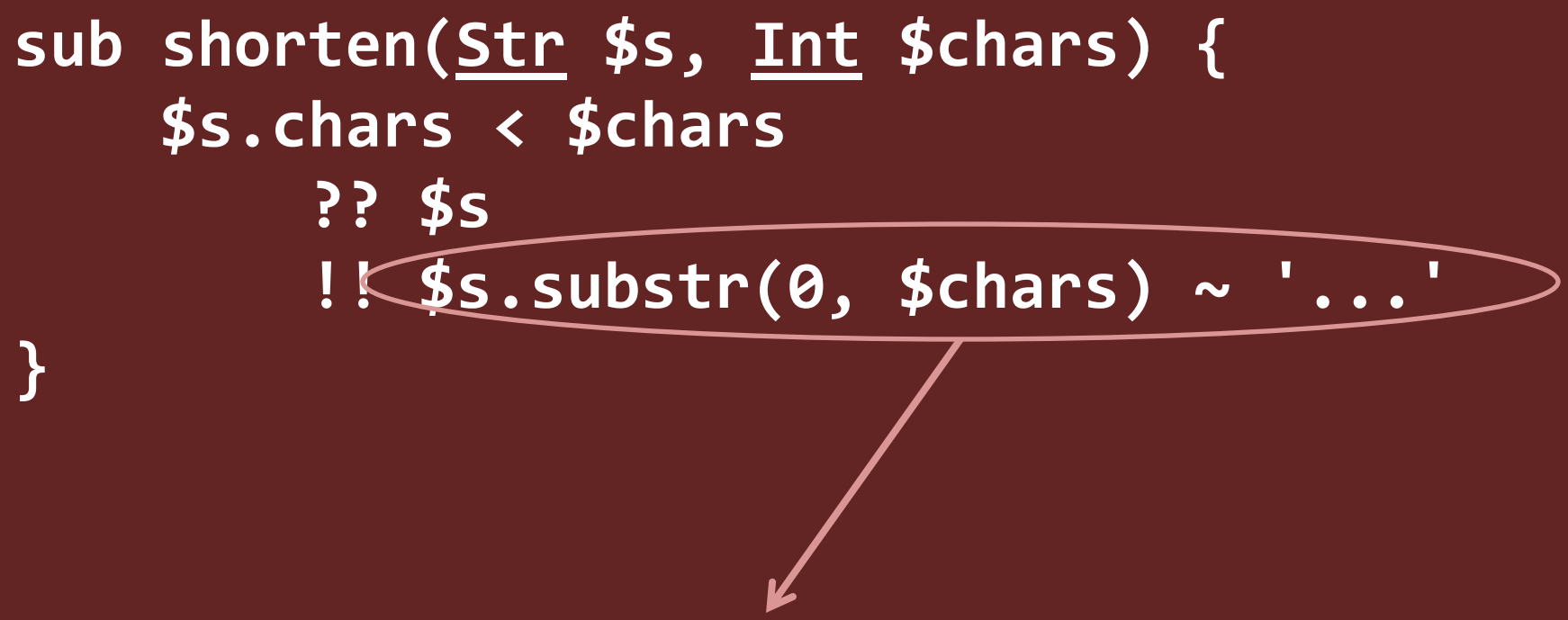
***…and* we often enforce the type constraints at runtime**

```
sub shorten(Str $s, Int $chars) {
    $s.chars < $chars
        ?? $s
        !! $s.substr(0, $chars) ~ '...'
}
```


sub shorten(Str $s, Int $chars) {
$s.chars < $chars
?? $s
!! $s.substr(0, $chars) ~ '...'
}
multi infix:<< < >>(Int $a, Int $b) {
…
}

```
sub shorten(Str $s, Int $chars) {
    $s.chars < $chars
        ?? $s
        !! $s.substr(0, $chars) ~ '...'
}
```

```
method substr(Int $from, Int $chars) {
    ...
}
```


sub shorten(Str $s, Int $chars) {
$s.chars < $chars
?? $s
!! $s.substr(0, $chars) ~ '...'
}
multi infix:<~>(Str $a, Str $b) {
...
}

Most operators are `multi` subs

Array and hash access are a call to a `multi` sub that in turn performs a method call

What if we were to try doing it that way in Perl 5?

```
my $arr = [1,2,3];
my $total = 0;
for (1..10_000_000) {
    $total += $total + $arr->[1] + $arr->[2];
}
print "$total\n";
```

0.509s

```
sub at_pos {
    @_[0]->[@_[1]]
}

sub postcircumfix {
    at_pos(@_[0], @_[1])
}

my $arr = [1,2,3];
my $total = 0;
for (1..10_000_000) {
    $total += $total + postcircumfix($arr, 1) +
        postcircumfix($arr, 2);
}
print "$total\n";
```

8.39s

```
sub at_pos {
    @_[0]->[@_[1]]
}

sub postcircumfix {
    at_pos(@_[0], @_[1])
}

sub infix_plus {
    @_[0] + @_[1]
}

my $arr = [1,2,3];
my $total = 0;
for (1..10_000_000) {
    $total = infix_plus($total,
        infix_plus(postcircumfix($arr, 1),
            postcircumfix($arr, 2)));
}
print "$total\n";
```

11.48s

Except we didn't actually do any multi-dispatch…

Except we didn't actually do any multi-dispatch…

And in Perl 6, `Int` is an object…

Except we didn't actually do any multi-dispatch…

And in Perl 6, `Int` is an object…

And `Int` automatically upgrades to a big integer too…

Except we didn't actually do any multi-dispatch…

And in Perl 6, `Int` is an object…

And `Int` automatically upgrades to a big integer too…

And Perl 6 arrays support laziness!

So what about Perl 6?

```
my @arr = 1,2,3;
my $total = 0;
for ^10_000_000 {
    $total += @arr[1] + @arr[2];
}
say $total;
```

Christmas release:

10.3s

Christmas release:

10.3s

Faster than the Perl 5
"translation"

Today:

0.886s

Today:

0.886s

Within 1.7x of Perl 5, despite all of the extra abstraction and work

Today:

0.886s

Which 1.7x of Perl 5, despite all of the extra abstraction and work

And a bit faster than the same benchmark in Python

Of course, nobody wants to know why it's challenging to go fast.

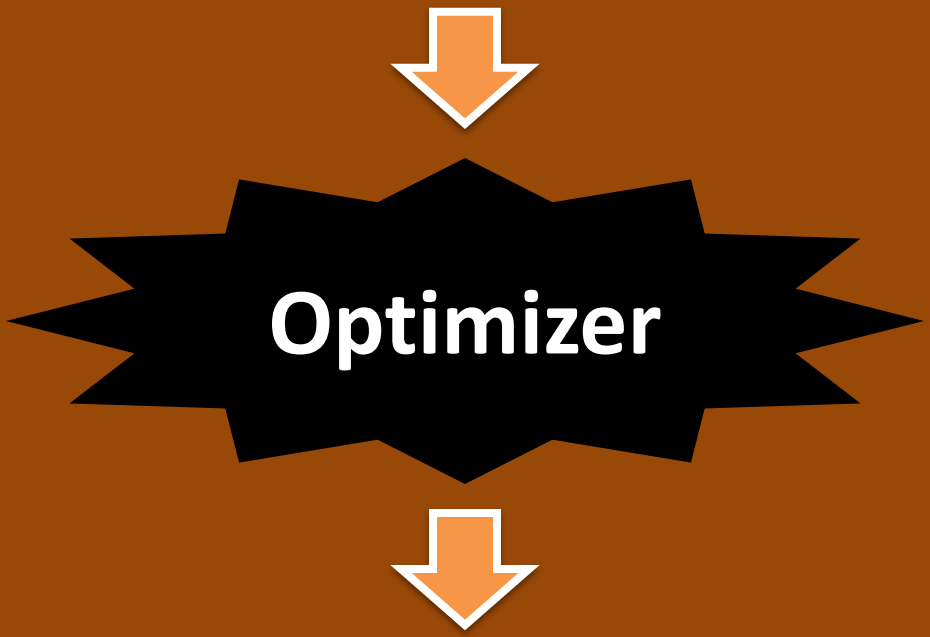
They just want it to be fast.

So, that's what we're doing.

The optimizations

we're performing to rise to the challenges

Programs that we want to develop and maintain



Programs that we want the computer to run

Static optimizer in Rakudo

Dynamic optimizer in MoarVM

Static Optimizations

Rewrites AST into faster constructs

Inlining of native operators

Lexical to local lowering

The static optimizer is...

Mostly doing local transforms

Sticking to cheap analyses, because it doesn't know what's worth a more sophisticated analysis

The dynamic optimizer is responsible for the

big

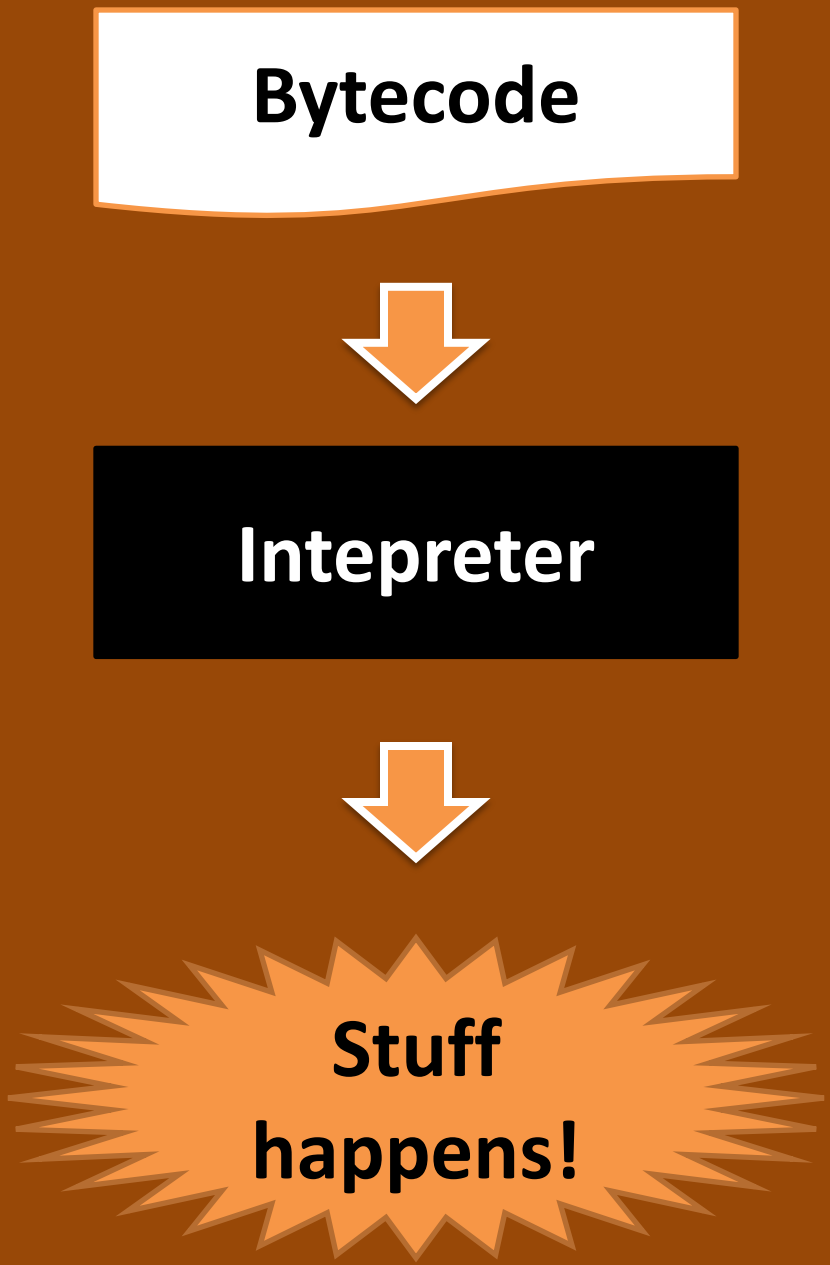
improvements

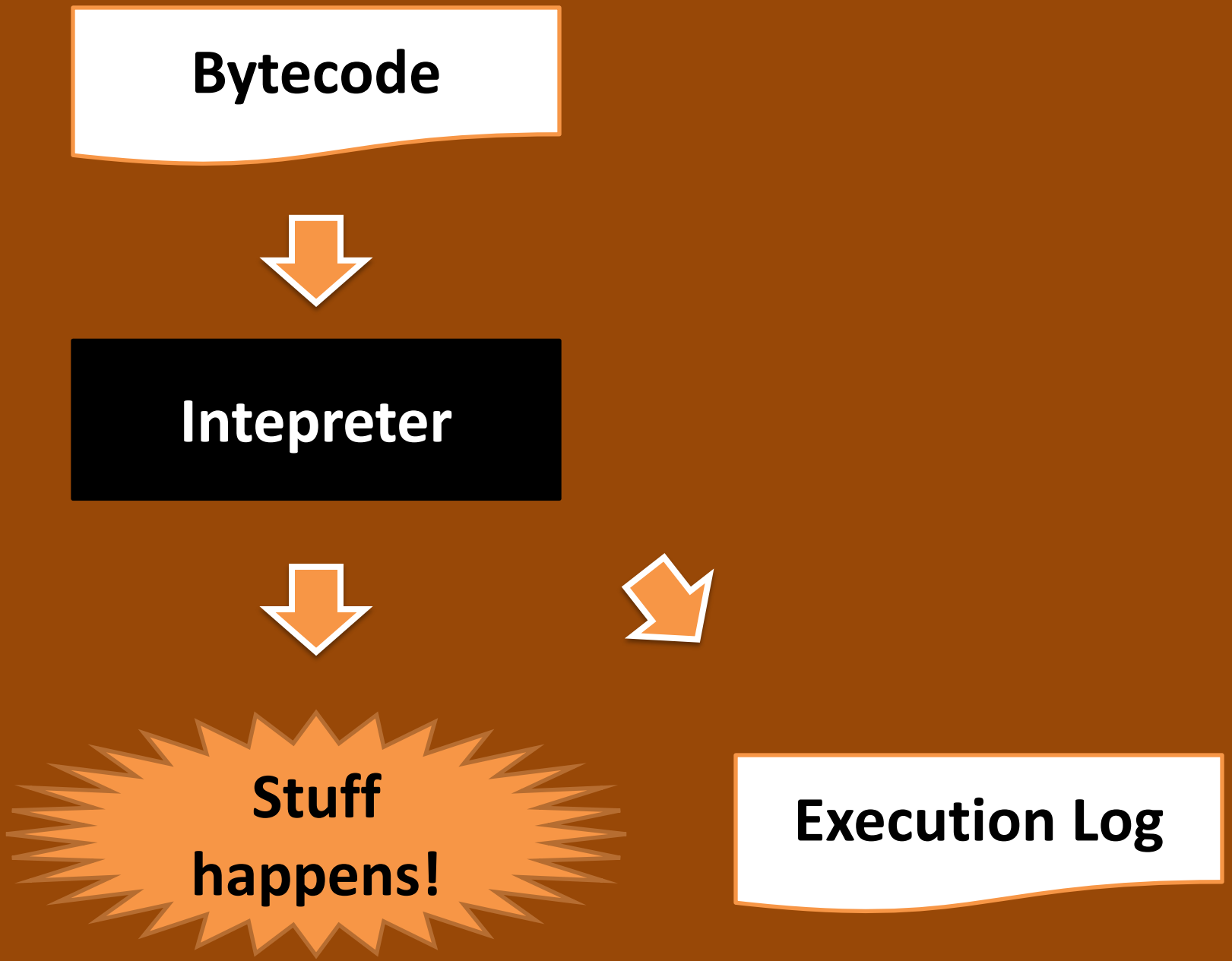
On the array access benchmark, it gives a

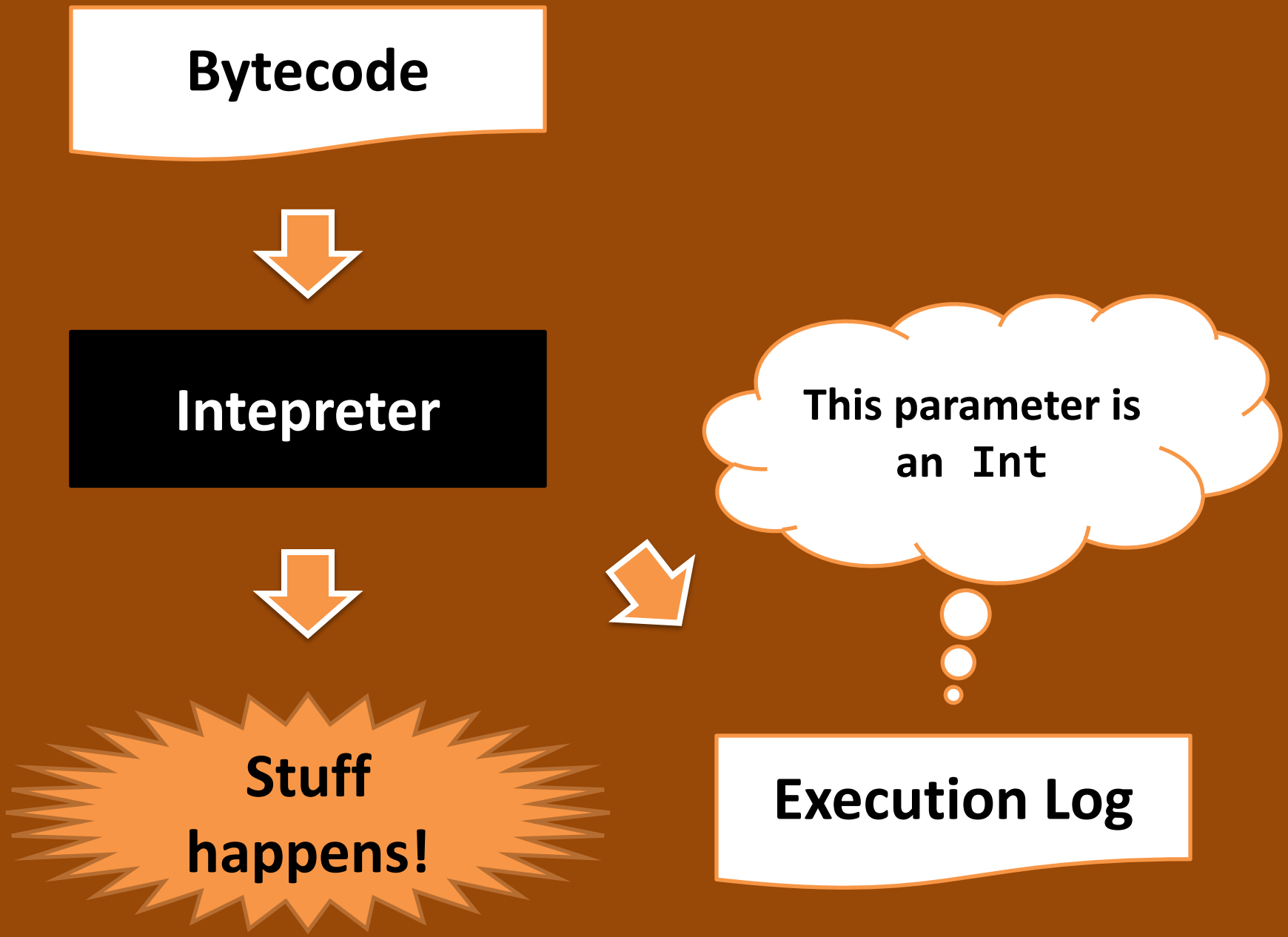
30x

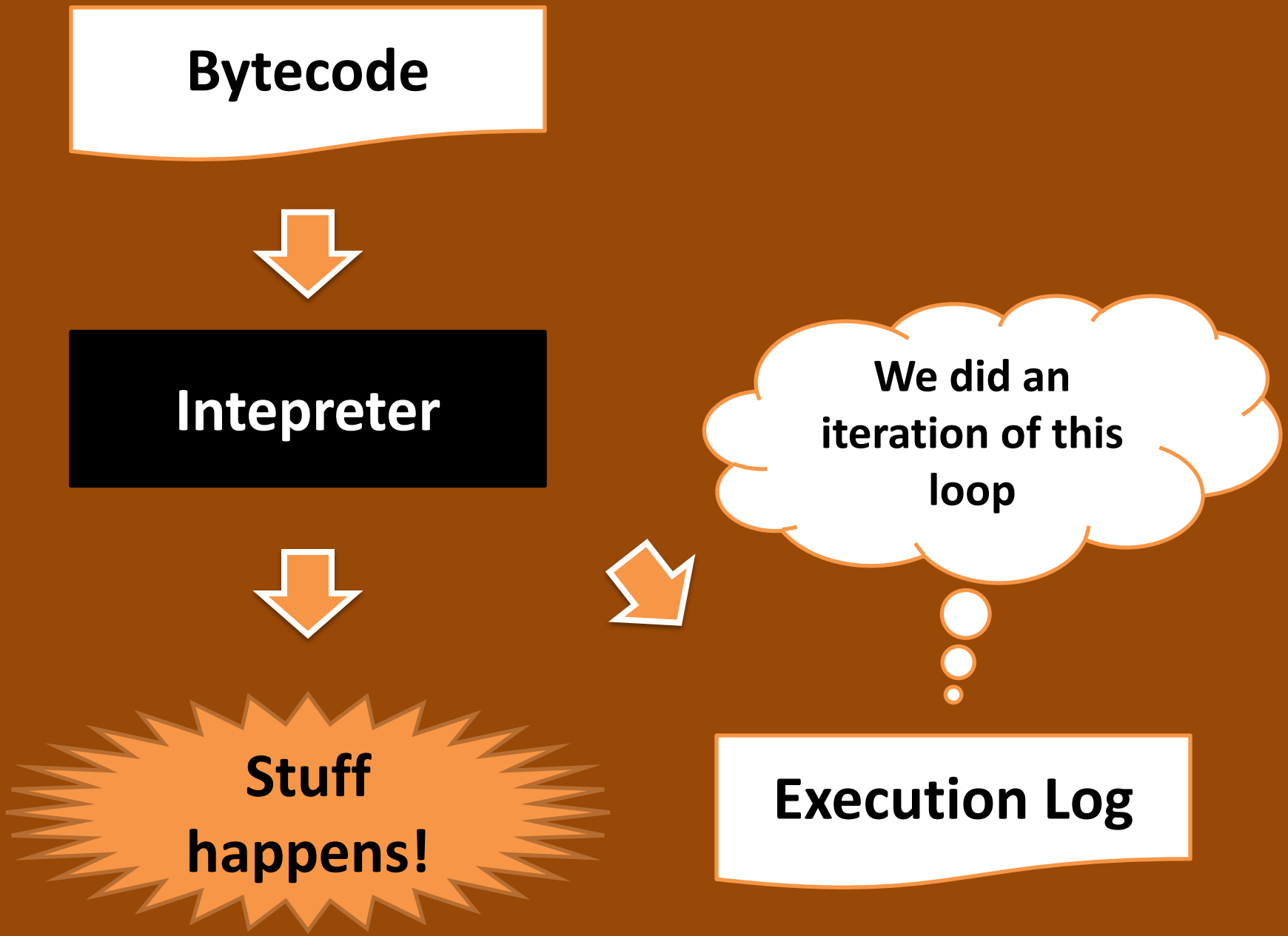
speedup

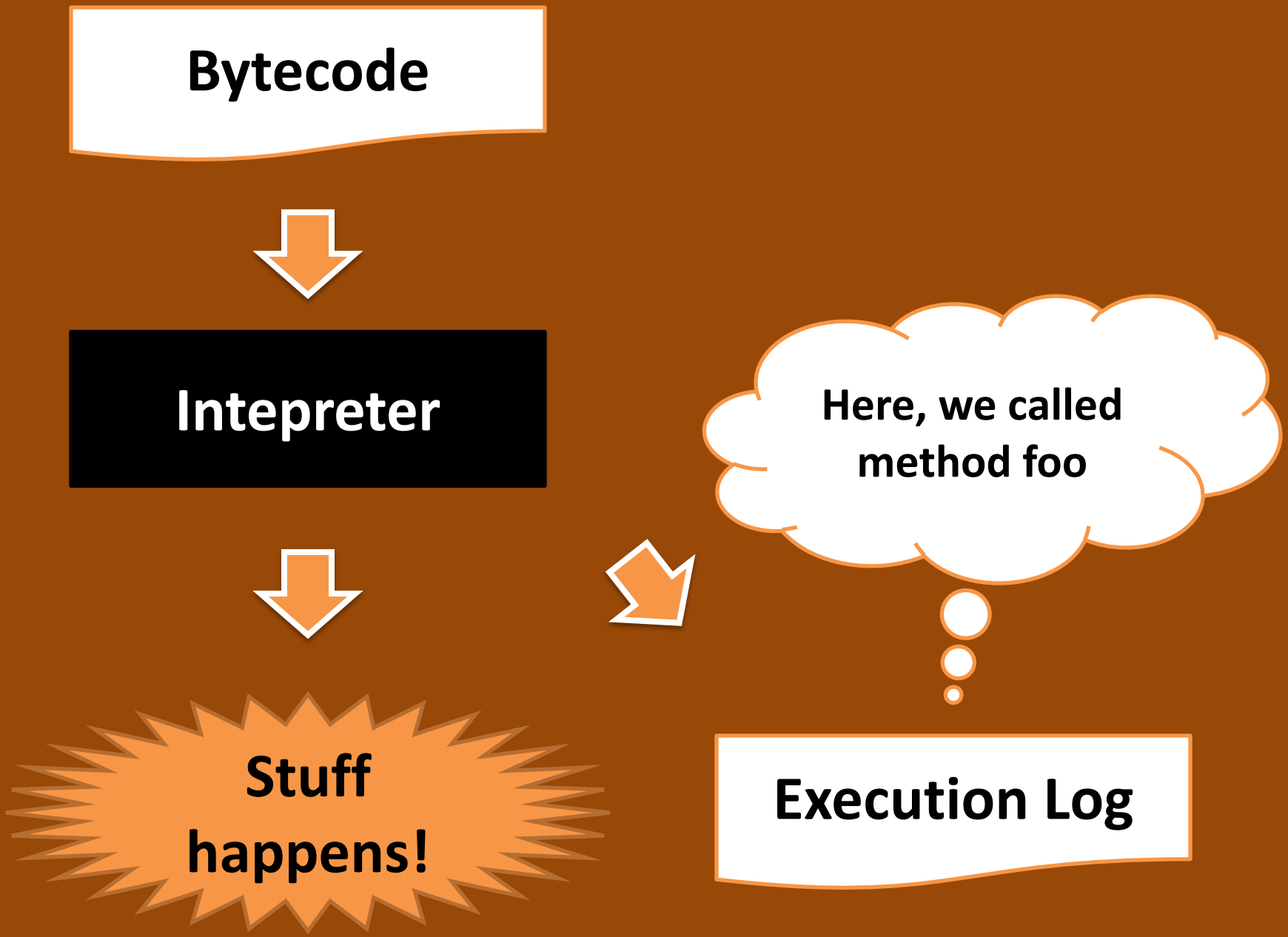
How?

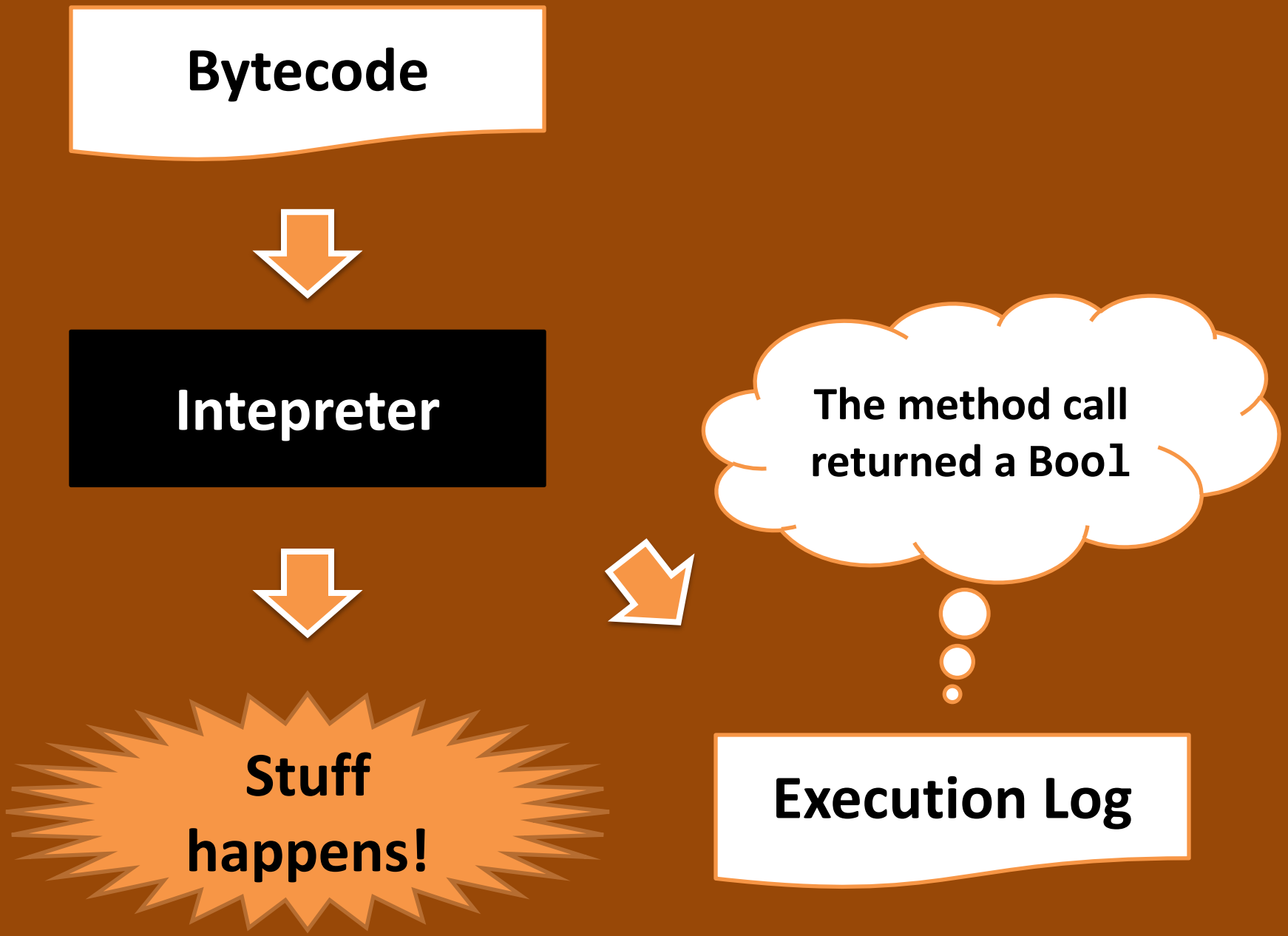

Bytecode
Intepreter
Stuff
happens!


Bytecode
Intepreter
Stuff happens!
Execution Log


Bytecode
Intepreter
Stuff happens!
This parameter is an Int
Execution Log

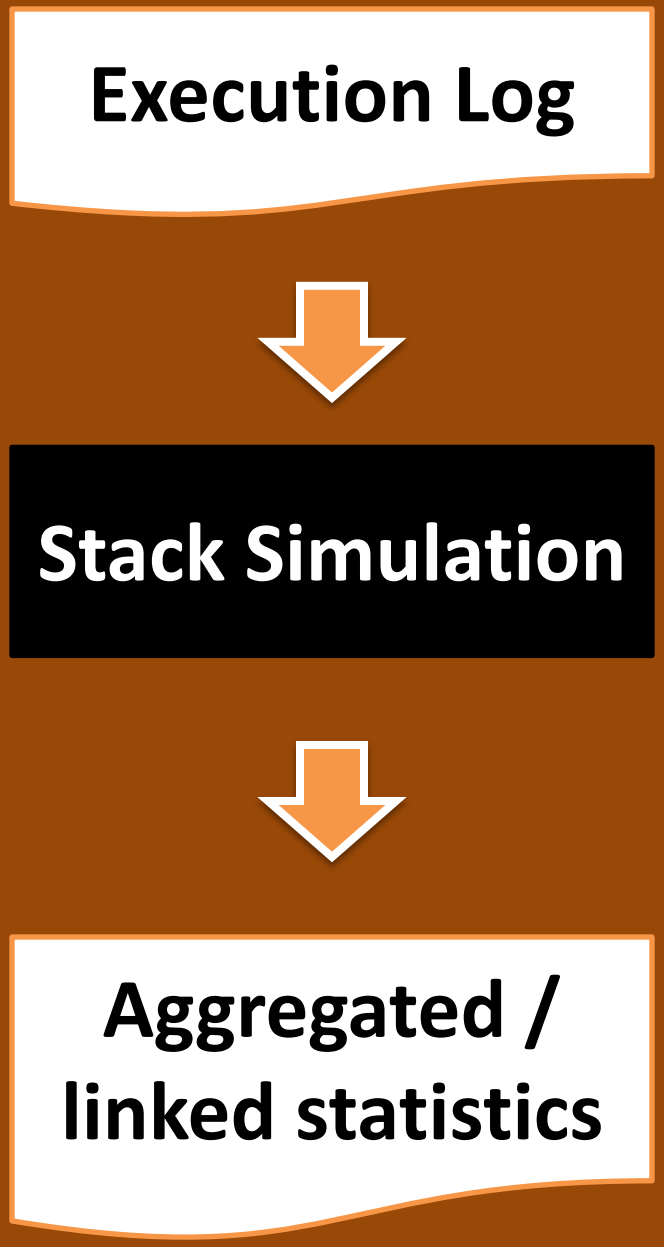

Bytecode
Intepreter
We did an iteration of this loop
Stuff happens!
Execution Log

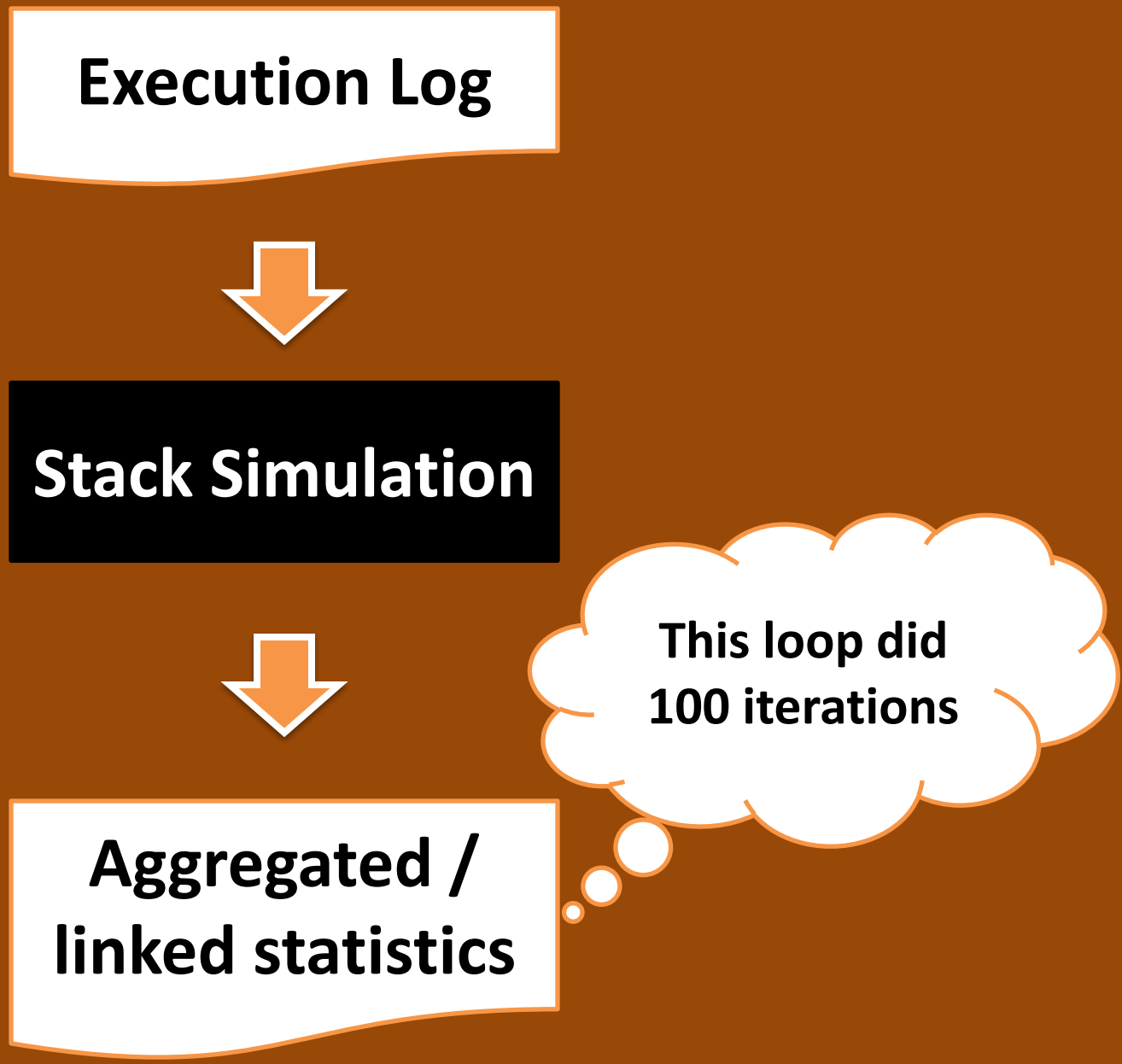

Bytecode
Intepreter
Here, we called method foo
Stuff happens!
Execution Log

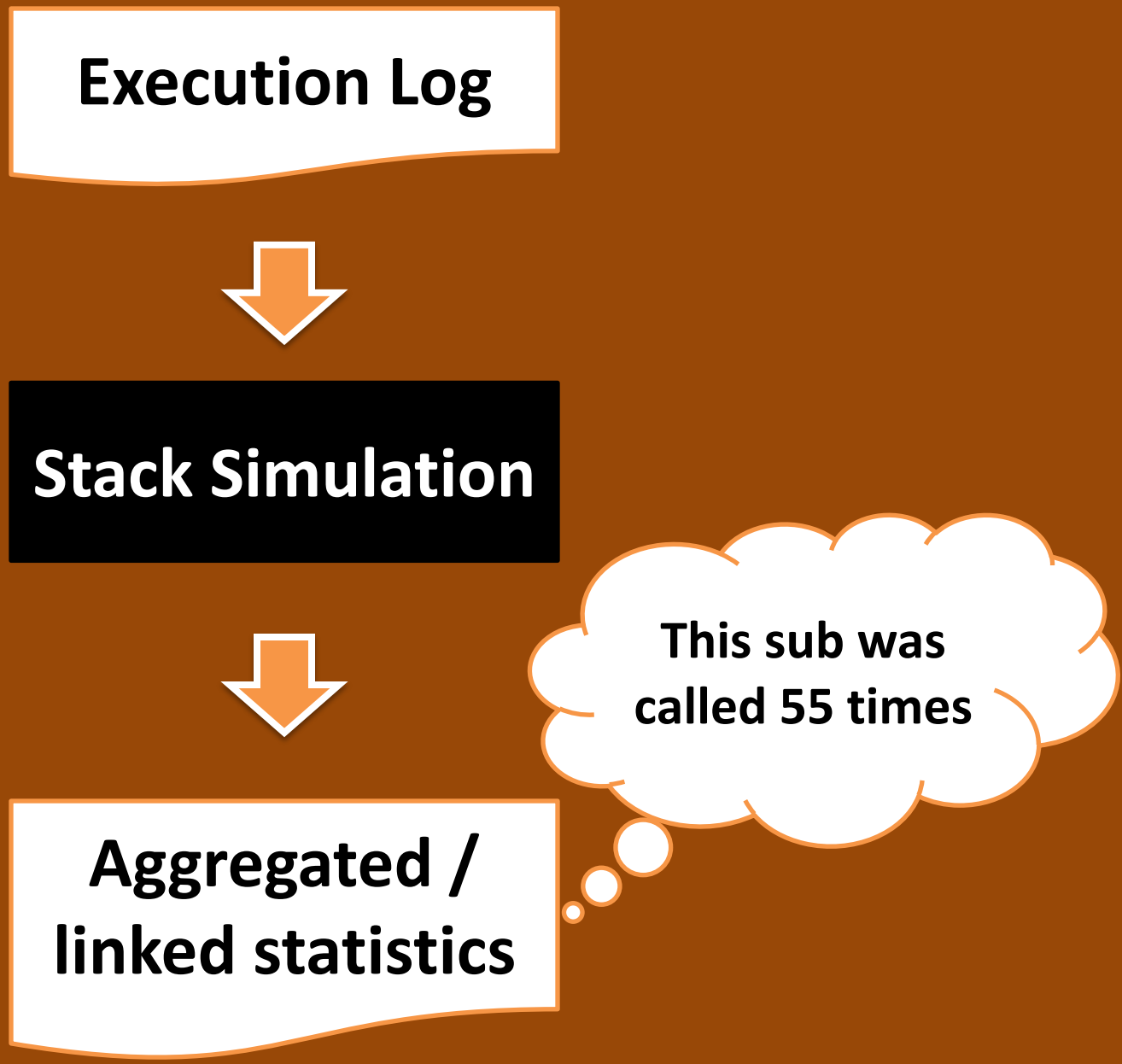

Bytecode
Intepreter
Stuff happens!
The method call returned a Bool
Execution Log

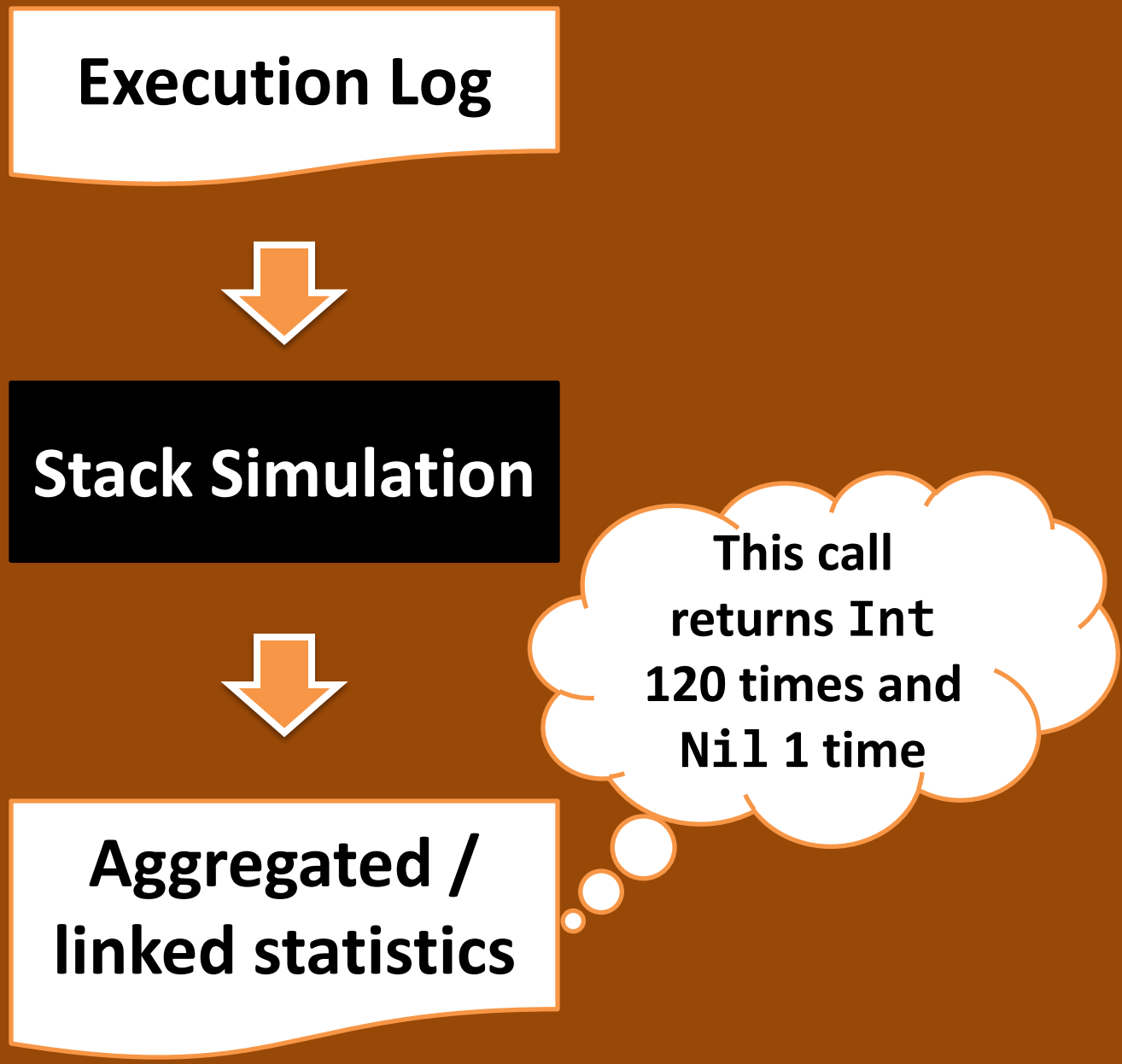
Meanwhile, on another thread…

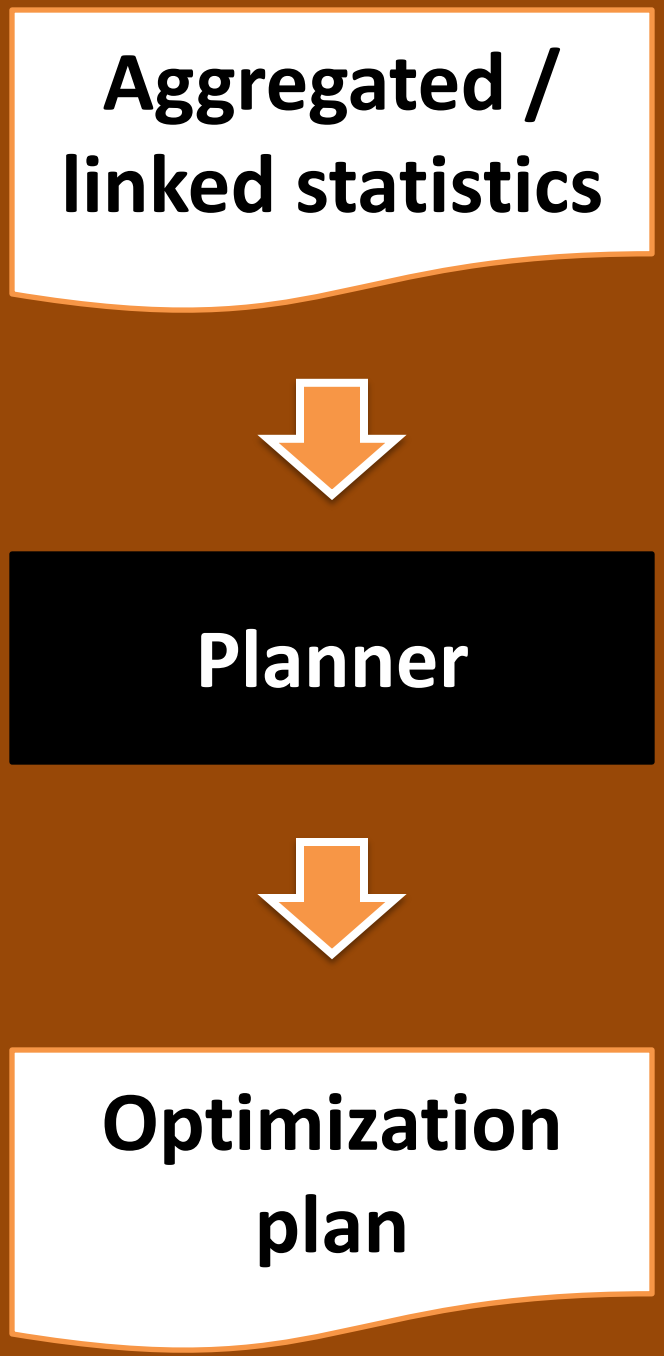

Oooh, a log packed full
of statistics!

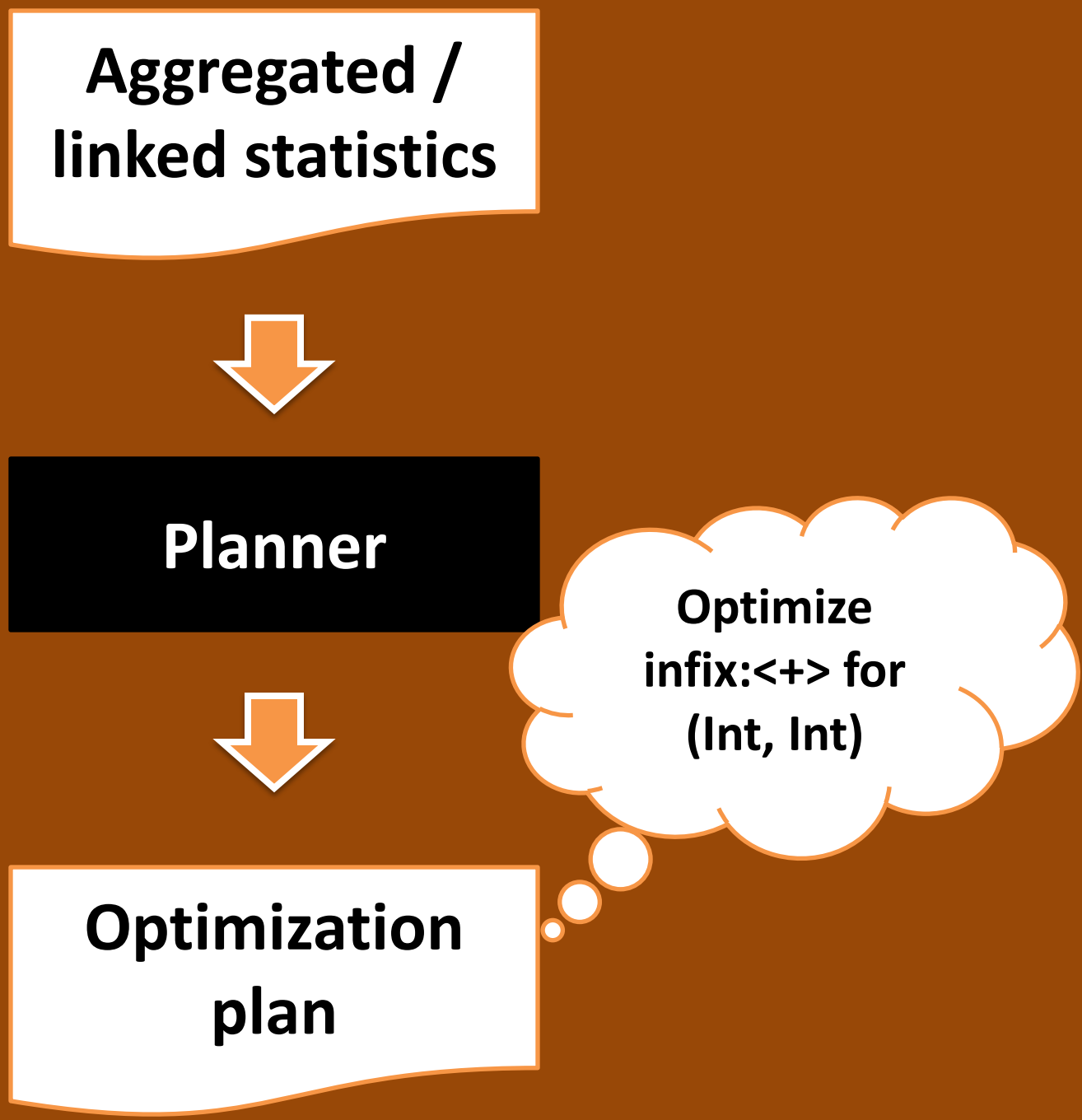

Execution Log
Stack Simulation
Aggregated / linked statistics

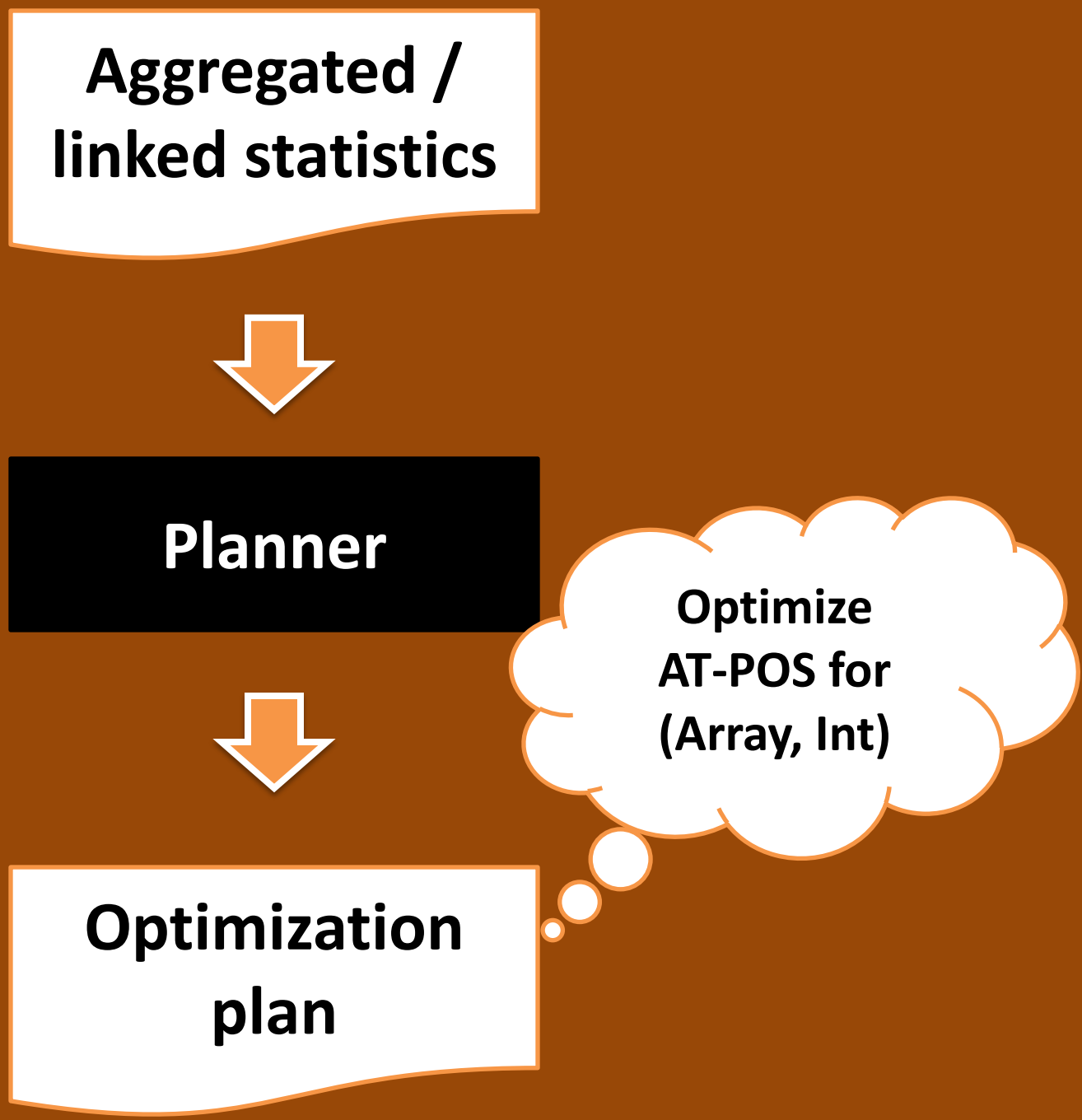

Execution Log
Stack Simulation
This loop did
100 iterations
Aggregated /
linked statistics


Execution Log
Stack Simulation
This sub was called 55 times
Aggregated / linked statistics


Execution Log
Stack Simulation
Aggregated / linked statistics
This call returns Int 120 times and Nil 1 time


Aggregated / linked statistics
Planner
Optimization plan

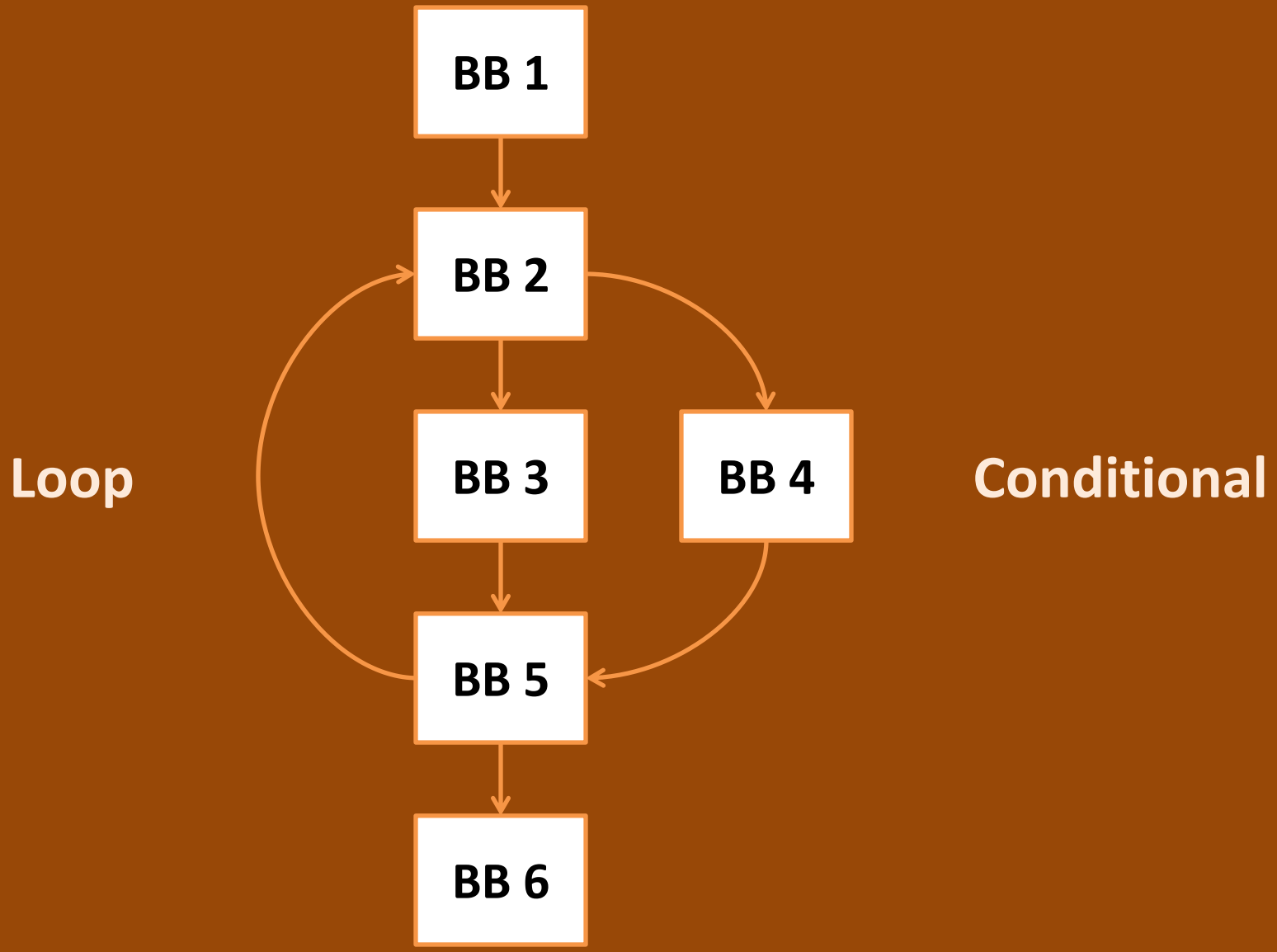

Aggregated / linked statistics
Planner
Optimize infix:<+> for (Int, Int)
Optimization plan


Aggregated / linked statistics
Planner
Optimize AT-POS for (Array, Int)
Optimization plan

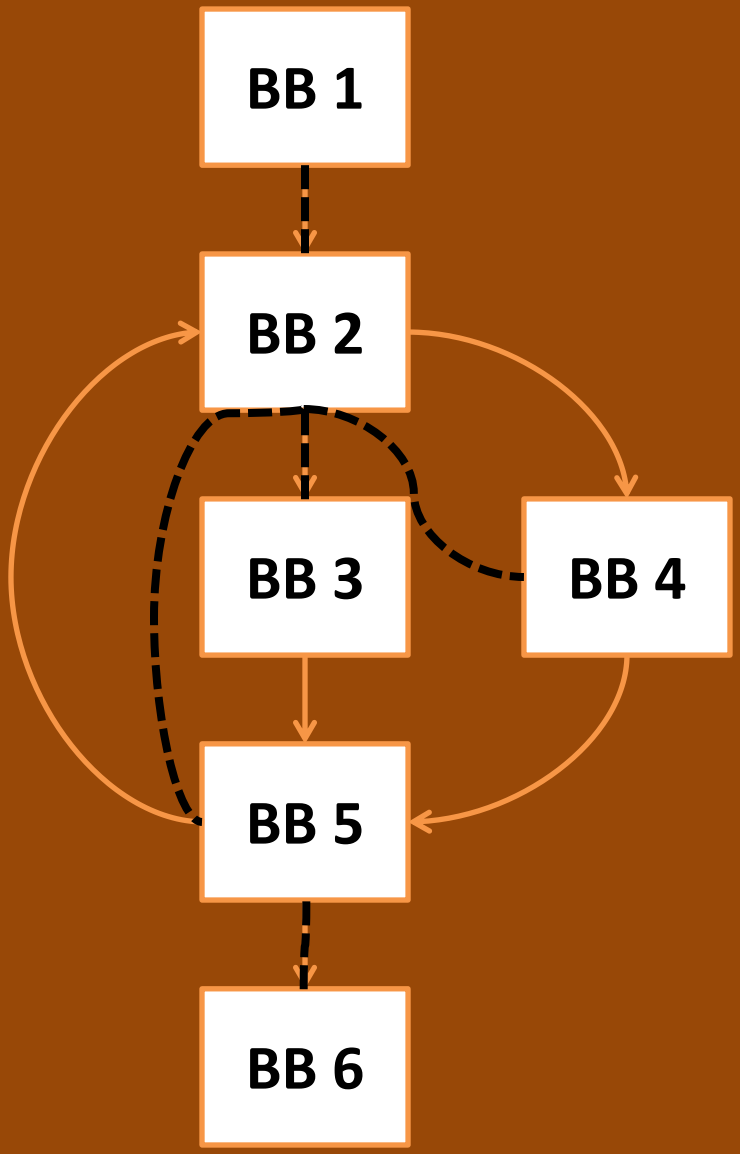
Bytecode isn't suitable for efficient program analysis

So, we parse it into a more suitable data structure

Control Flow Graph

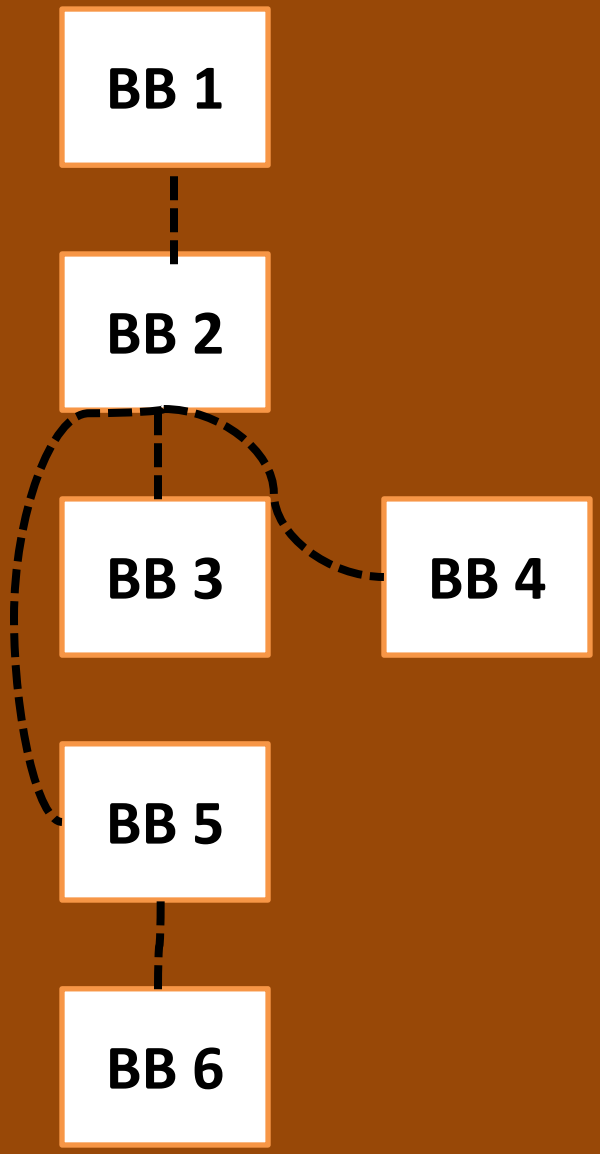


Dominance tree



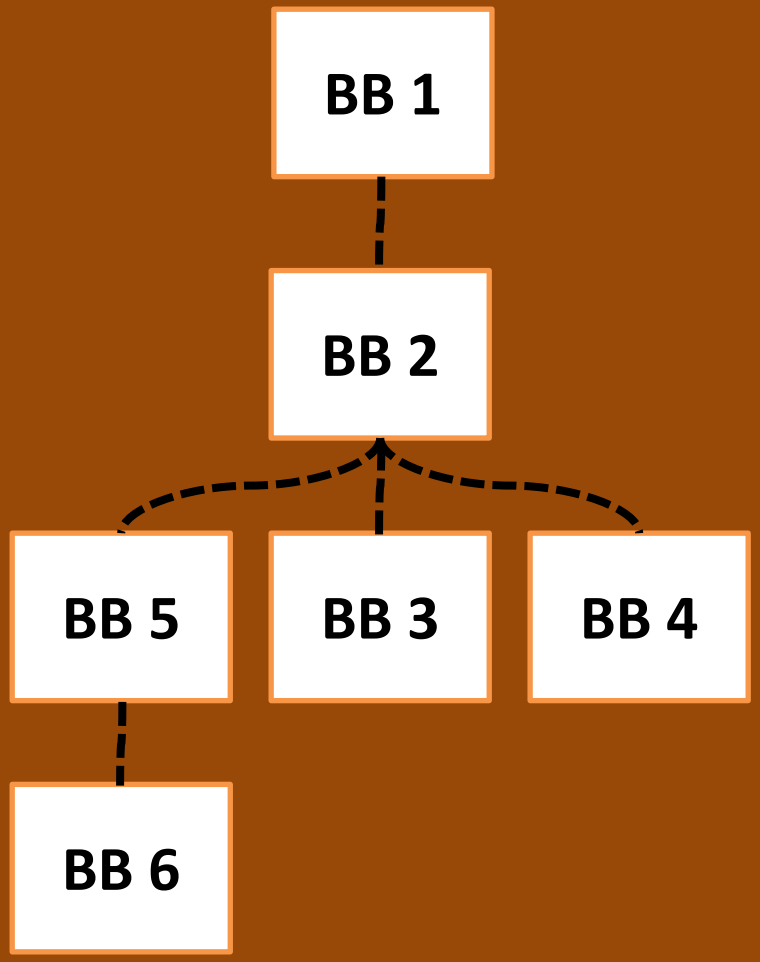
Block	Immediately Dominates
BB1	BB2
BB2	BB3, BB4, BB5
BB3	
BB4	
BB5	BB6
BB6	

Dominance tree



Block	Immediately Dominates
BB1	BB2
BB2	BB3, BB4, BB5
BB3	
BB4	
BB5	BB6
BB6	

Dominance tree



Block	Immediately Dominates
BB1	BB2
BB2	BB3, BB4, BB5
BB3	
BB4	
BB5	BB6
BB6	

SSA Form

```
param_rp_i r0, liti16(0)
param_rp_i r1, liti16(1)
mul_i r0, r0, r0
add_i r0, r0, r1
return_i r0
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1, liti16(1)
mul_i r0, r0, r0
add_i r0, r0, r1
return_i r0
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1(1), liti16(1)
mul_i r0, r0, r0
add_i r0, r0, r1
return_i r0
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1(1), liti16(1)
mul_i r0(2), r0(1), r0(1)
add_i r0, r0, r1
return_i r0
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1(1), liti16(1)
mul_i r0(2), r0(1), r0(1)
add_i r0(3), r0(2), r1(1)
return_i r0
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1(1), liti16(1)
mul_i r0(2), r0(1), r0(1)
add_i r0(3), r0(2), r1(1)
return_i r0(3)
```

SSA Form

```
param_rp_i r0(1), liti16(0)
param_rp_i r1(1), liti16(1)
mul_i r0(2), r0(1), r0(1)
add_i r0(3), r0(2), r1(1)
return_i r0(3)
```

(Plus some mechanism to deal with branches.
The dominance calculation helps there.)

We associate

facts

with each SSA variable

But statistics aren't facts, they're just statistics!

So, we insert

guards

that deoptimize if the type isn't what was predicted

Finally, we're ready to go ahead and apply lots of **optimizations!**

Rewrite a method lookup into a constant, because we know the precise type

Rewrite a multi-dispatch into a direct call to the correct candidate

Rewrite a call to the general code into a call to the applicable *specialization*

For small callees, *inline* the callee's code into that of the caller

Eliminate duplicate type checks that are already proven by existing facts

Eliminate guards when we can do a proof that its condition will be met

Eliminate conditionals when we can prove which way they will go

Rewrite attribute access into simple, unchecked, pointer dereferences

We also recently got

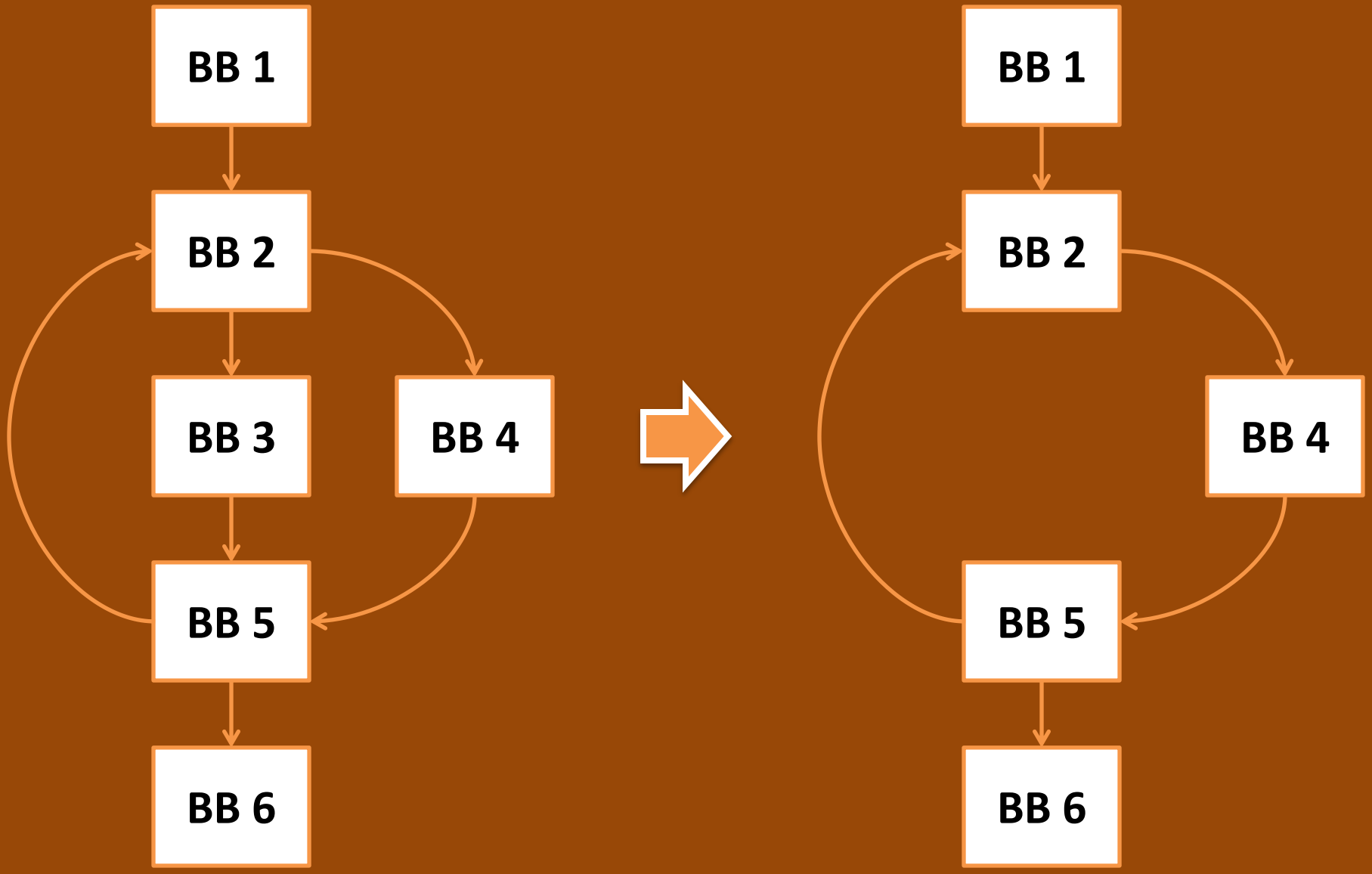
escape analysis

Replace object allocations with a register per attribute

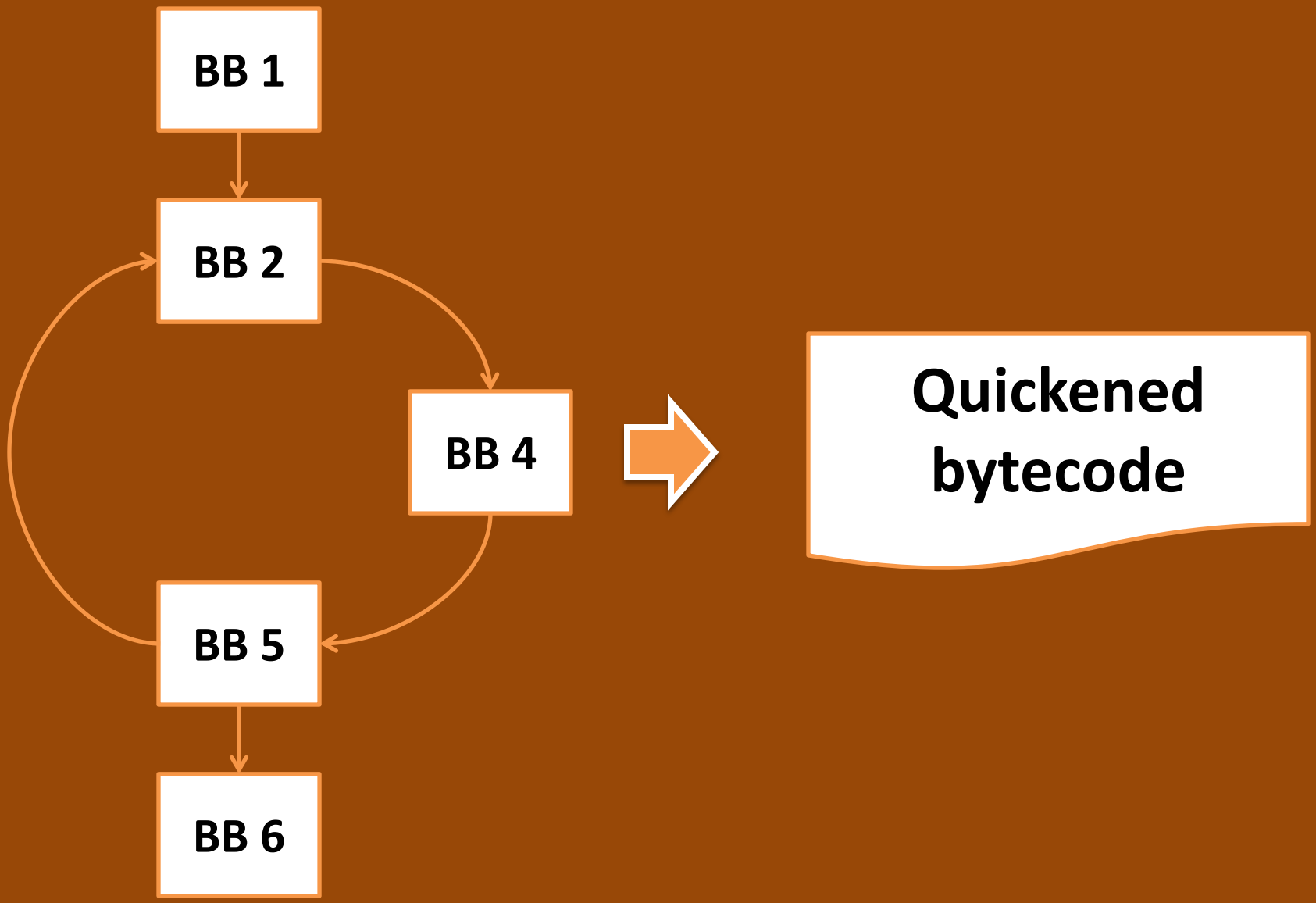
Eliminate, sink, or defer the object allocation

Do type proofs that look into objects ➔ eliminate more guards!

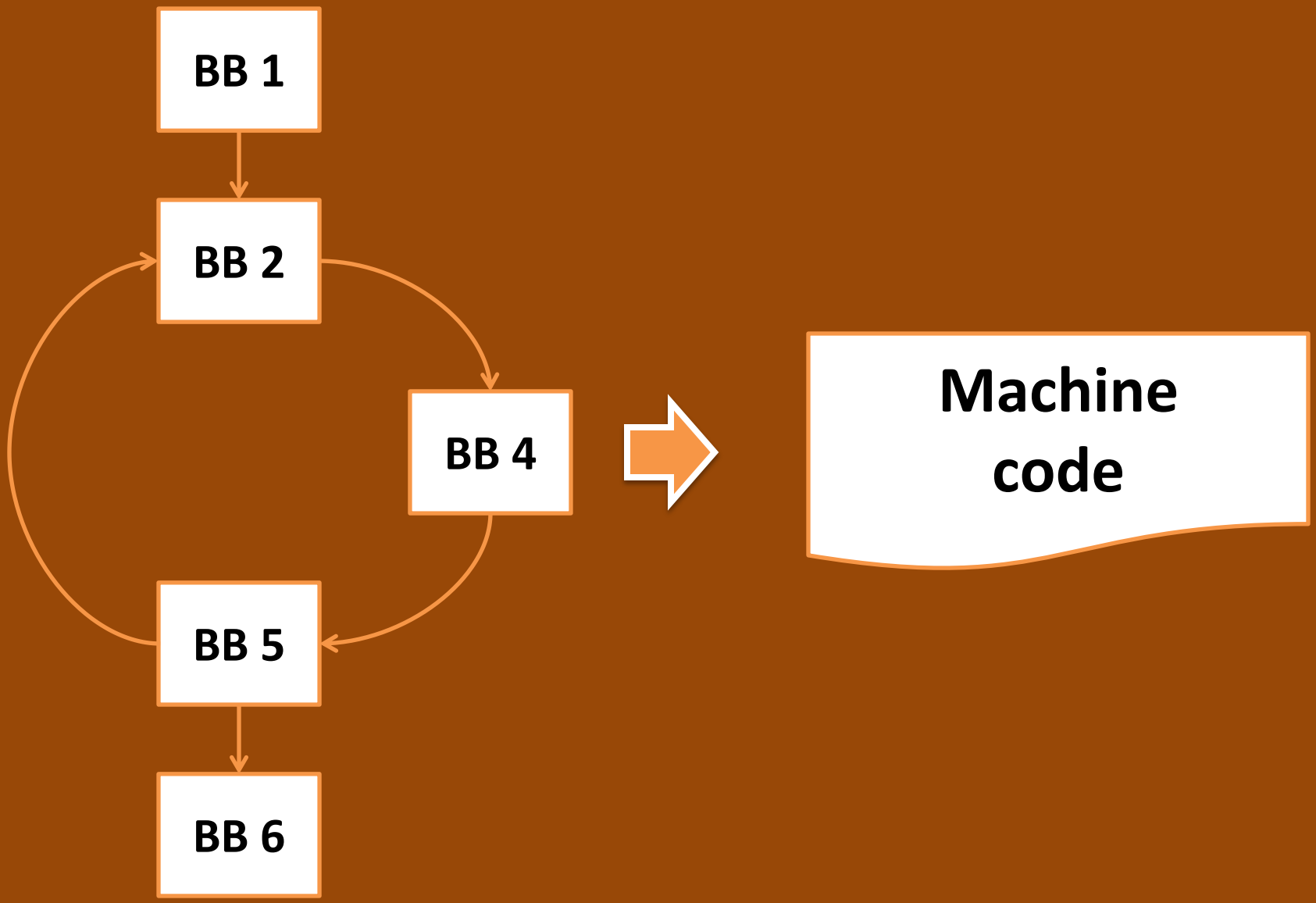
Get an optimized graph…



…and generate optimized code



…and, on x64, machine code



The results

of various benchmarks

Disclaimer

There's lies, statistics, and benchmarks ☺

Some of these numbers rely on EA-based optimizations not yet available in a default build

No tricks

I tried to write the kind of code a typical programmer would write, *not* use every Perl 6 trick I know to squeeze out more speed.

Compared to the Christmas release, today's Rakudo and MoarVM are

much faster!

Benchmark	Xmas	Today	Improvement
Read a million lines (UTF-8)	3.217	0.508	**6.33**
Array reading and addition	10.214	0.875	**11.67**
Hash reading	17.357	0.862	**20.14**
Hash store	40.134	2.247	**17.86**
Complex	11.092	0.695	**15.96**
Short-lived point object	21.174	0.369	**57.38**
Parse 10,000 docker files	23.964	6.145	**3.9**
Million native calls	4.727	0.898	**5.26**

But what about compared to Perl 5, Python, or Ruby?

Not a competition to see which is fastest, but rather to see if Perl 6 is competitive.

Some results are already looking fairly decent…

Basic object operations on a short-lived object

```
class Point {
    has $.x;
    has $.y;
}
my $total = 0;
for ^1_000_000 {
    my $p = Point.new(x => 2, y => 3);
    $total = $total + $p.x + $p.y;
}
say $total;
```

Basic object operations on a short-lived object

Language	Time	Perl 6 is...
Perl 6	0.385	-
Perl 5	0.955	2.48x faster
Python	0.351	1.10x slower
Ruby	0.191	2.02x slower

Basic object operations on a short-lived object

Language	Time	Perl 6 is...
Perl 6	0.385	-
Perl 5	0.955	2.48x faster
Python	0.351	1.10x slower
Ruby	0.191	2.02x slower

Non-Perls use positional parameters in the constructor…

Basic object operations on a short-lived object

Language	Time	Perl 6 is...
Perl 6	0.385	-
Perl 5	0.955	2.48x faster
Python	0.351	1.10x slower
Ruby	0.191	2.02x slower

...so we need to EA away the temporary hash to compete with them

Read a million lines of UTF-8 (checking it) and count the chars

```
my $fh = open "longfile";
my $chars = 0;
for $fh.lines {
    $chars = $chars + .chars
}
$fh.close;
say $chars
```

Read a million lines of UTF-8 (checking it) and count the chars

Language	Time	Perl 6 is...
Perl 6	0.509	-
Perl 5	0.977	**1.92x faster**
Python	2.207	**4.34x faster**
Ruby	0.412	**1.24x slower**

Read a million lines of UTF-8 (checking it) and count the chars

Language	Time	Perl 6 is...
Perl 6	0.509	-
Perl 5	0.977	1.92x faster
Python	2.207	4.34x faster
Ruby	0.412	1.24x slower

Perl 6, unlike the others, has grapheme-level strings.

Integer math (allowing use of Perl 6 native `int`)

```
sub gcd(int $a is copy, int $b is copy) {
    while $b ≠ 0 {
        my int $t = $b;
        $b = $a % $b;
        $a = $t;
    }
    $a
}
for ^2_000_000 {
     die "oops" unless gcd(40, 30) == 10;
}
```

Integer math (allowing use of Perl 6 native `int`)

Language	Time	Perl 6 is...
Perl 6	0.664	-
Perl 5	0.884	**1.33x faster**
Python	0.406	**1.63x slower**
Ruby	2.69	**4.05x faster**

Integer math (allowing use of Perl 6 native `int`)

Language	Time	Perl 6 is...
Perl 6	0.664	-
Perl 5	0.884	1.33x faster
Python	0.406	1.63x slower
Ruby	2.69	4.05x faster

With JIT, we should really sweep the floor with this one. Alas, not yet.

Some simple operations using complex numbers

```
my $total-re = 0e0;
for ^2_000_000 {
    my $x = 5 + 2i;
    my $y = 10 + 3i;
    my $z = $x * $x + $y;
    $total-re = $total-re + $z.re
}
say $total-re;
```

Some simple operations using complex numbers

Language	Time	Perl 6 is...
Perl 6	0.175	-
Perl 5	40.1	**229x faster**
Python	1.16	**6.61x faster**
Ruby	1.52	**8.68x faster**

Some simple operations using complex numbers

Language	Time	Perl 6 is...
Perl 6	0.175	-
Perl 5	40.1	229x faster
Python	1.16	6.61x faster
Ruby	1.52	8.68x faster

Hmmm. Obtained using Math::Complex. It's built-in for other languages.

Some simple operations using complex numbers

Language	Time	Perl 6 is...
Perl 6	0.175	-
Perl 5	40.1	229x faster
Python	1.16	6.61x faster
Ruby	1.52	8.68x faster

EA allows us to totally eliminate the temporary `Complex` objects

Really need to do better at arrays and hashes…

Reading from an array, plus basic integer math

```
my @arr = 1,2,3;
my $total = 0;
for ^10_000_000 {
    $total += @arr[1] + @arr[2];
}
say $total;
```

Reading from an array, plus basic integer math

Language	Time	Perl 6 is...
Perl 6	0.886	-
Perl 5	0.514	**1.72x slower**
Python	1.00	**1.13x faster**
Ruby	0.509	**1.74x slower**

Lots of assignments into a dynamically allocated array

```
for ^10_000 {
    my @arr;
    for ^1_000 {
        @arr[$_] = 42;
    }
}
```

Lots of assignments into a dynamically allocated array

Language	Time	Perl 6 is...
Perl 6	0.734	-
Perl 5	0.527	**1.40x slower**
Python	0.624	**1.18x slower**
Ruby	0.505	**1.46x slower**

Lots of assignments into a dynamically allocated array

Language	Time	Perl 6 is...
Perl 6	0.734	-
Perl 5	0.527	**1.40x slower**
Python	0.624	**1.18x slower**
Ruby	0.505	**1.46x slower**

Every array slot is a `Scalar`, which we have to allocate.

Lots of assignments into a dynamically allocated array

Language	Time	Perl 6 is...
Perl 6	0.734	-
Perl 5	0.527	**1.40x slower**
Python	0.624	**1.18x slower**
Ruby	0.505	**1.46x slower**

Plus, arrays may be lazy, which creates a little extra overhead too (for now).

Reading values from a hash and basic integer math

```
my %h = a => 10, b => 12;
my $total = 0;
for ^10_000_000 {
    $total = $total + %h<a> + %h<b>;
}
```

Reading values from a hash and basic integer math

Language	Time	Perl 6 is...
Perl 6	0.886	-
Perl 5	0.787	**1.12x slower**
Python	1.15	**1.30x faster**
Ruby	0.597	**1.48x slower**

Set up lots of hashes with keys obtained from an array

```
my @keys = 'a'..'z';
for ^500_000 {
    my %h;
    for @keys {
        %h{$_} = 42;
    }
}
```

Set up lots of hashes with keys obtained from an array

Language	Time	Perl 6 is...
Perl 6	2.30	-
Perl 5	1.65	1.35x slower
Python	0.837	2.66x slower
Ruby	2.64	1.18x faster

Set up lots of hashes with keys obtained from an array

Language	Time	Perl 6 is...
Perl 6	2.30	-
Perl 5	1.65	1.35x slower
Python	0.837	2.66x slower
Ruby	2.64	1.18x faster

The Perls certainly are doing hash randomization - but who else is?

Set up lots of hashes with keys obtained from an array

Language	Time	Perl 6 is...
Perl 6	2.30	-
Perl 5	1.65	1.35x slower
Python	0.837	2.66x slower
Ruby	2.64	1.18x faster

Perl 6 is, as with arrays, also doing a `Scalar` allocation per element

And then some things really need work…

Startup time - important for scripting - is still unimpressive

Language	Time	Perl 6 is...
Perl 6	0.093	-
Perl 5	0.0047	**19.9x slower**
Python	0.011	**8.40x slower**
Ruby	0.038	**2.47x slower**

And please, let's not talk about regex performance…

…oh well, OK, if we must…

Perl 5

```
my $i = 0;
for (1..10_000_000) {
    $i++ if "boo" =~ /^b/
}
say $i;
```

Perl 6

```
my $i = 0;
for ^10_000_000 {
    $i++ if "boo" ~~ /^b/
}
say $i;
```

Perl 5

```
my $i = 0;
for (1..10_000_000) {
    $i++ if "boo" =~ /^b/
}
say $i;
```

1.60s

Perl 6

```
my $i = 0;
for ^10_000_000 {
    $i++ if "boo" ~~ /^b/
}
say $i;
```

38.7s

(24x slower)

Rakudo doesn't yet know how to avoid using the regex engine for simple things - but Perl 5 seems to be really rather good at that.

So what if we *manually* avoid it in Perl 6, to see what we might be able to achieve?

Perl 5

```
my $i = 0;
for (1..10_000_000) {
    $i++ if "boo" =~ /^b/
}
say $i;
```

Perl 6, using `starts-with`

```
my $i = 0;
for ^10_000_000 {
    $i++ if "boo".starts-with('b')
}
say $i;
```

Perl 5

```
my $i = 0;
for (1..10_000_000) {
    $i++ if "boo" =~ /^b/
}
say $i;
```

1.60s

Perl 6, using `starts-with`

```
my $i = 0;
for ^10_000_000 {
    $i++ if "boo".starts-with('b')
}
say $i;
```

0.700

(2.2x faster)

But still…even the case where we *do* hit the regex engine (or use grammars) needs to be faster.

The consequences

for those writing Perl 6 programs today

Inlining means that calling an accessor is about as cheap as accessing an attribute

And both of those are cheaper than using a hash instead of an object

Similarly, small subs and methods (and private methods) can be inlined too, so don't worry much over using those

Avoid regexes when a simple method - like `starts-with` or `contains` - will do the job

Some constructs are not yet well optimized. There's usually more than one way to do things, so - on hot path code - experiment with some other ways.

Slow things today include…

Destructuring (and signature unpacks)
Multi-dispatch with `where` clauses
Flattening into argument lists
Multi-dimensional arrays

(But if you're reading this in 2020 or later, check these are still true, because things improve regularly. ☺)

Assignment into an array or hash *copies* into the target

Binding, carefully used, can turn O(n) into O(1)

Some modules are notably faster than others, so consider those too

Recently, got a roughly 5x speedup by switching YAML module

And, of course, Perl 6 parallelism support can be a great "get out of jail free" card

The plans

for further performance improvements

Well, obviously…

Optimize away use of regexes where they aren't needed

And make the regex and grammar implementation fast anyway

More EA

Current focus is getting the latest round of work into user's hands

Beyond that, make EA understand loops, and able to scalar replace arrays and hashes

Speed up array/hash

Performance parity is within reach, largely by squeezing more waste out of the generated code

Speed up array/hash

To be notably faster than Perl 5 and friends, we need to do more

Can delay or even avoid `Scalar` allocation - if we can better convey when we only need an r-value. That's a tricky problem.

Region JIT?

Currently, MoarVM is a method JIT with aggressive inlining

But our statistics model means we could do region JIT, and it'd probably be a win for us

Keep working at it

There's no shortcut to maturity

Need to continue analyzing things that are slow, understanding why, and finding solutions

The ultimate goal here, is that performance joins with the many other reasons that one might *choose* to use Perl 6

It's hard work.
It's challenging.

It's hard work.
It's challenging.

But it's in our grasp.

Questions?

@ jonathan@edument.cz

W jnthn.net

 jnthnwrthngtn

 jnthn