Rakudo Perl 6 and MoarVM Performance Advances

Jonathan Worthington


Hi. I'm Jonathan.

Hi. I'm Jonathan.

```
> "Jonathan".subst(/<[aeiou]>/, '', :g).lc
jnthn
```

My Goal:

Eliminate the implementation issues that stand in the way of greatly increased Perl 6 adoption.

Software development

More about learning than about building.

So I value...

Speed of "idea ➔ running code"

and

Ease of refactoring, to incorporate new learning

Which are Perl values

Whipuptitude

and

Manipulexity

And Perl 6 gives me these things to an even greater degree ☺

Perl 6: my learningest project

"Torment the implementers for the sake of the users" isn't a joke!

In my first couple of years, I learned rather a lot about how *not* to implement Perl 6.

But nowadays...

Vast majority of features in place

(little left that isn't "post-6.0 wish list")

Solid compiler architecture

(third time's a charm)

Lots of tests, growing ecosystem

(tells us quickly when we broke something)

Time for performance work

Optimizing the wrong design not only wastes time, it makes it harder to work towards the right one.

Now we had a design we were happy with, and performance being a real adoption blocker, *it was time*.

2013.08 - 2014.08

In this session, I'll look at the improvements made relative to YAPC::Europe last year.

There will be code. There will be graphs. There will be computer science. There will be...a reveal.

Rakudo Perl 6 Architecture

Rakudo Perl 6 Architecture

Perl 6 Source

```
say "Badger, " x 8,
  "Mushroom, " x 2,
     "SNAKE! " x 2;
```

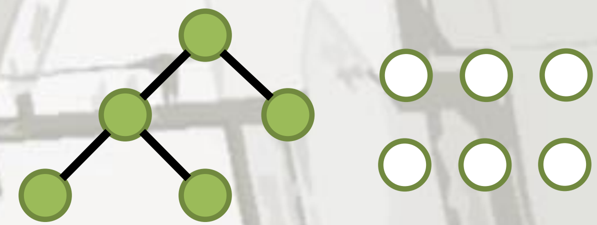
Rakudo Perl 6 Architecture

Perl 6 Source

```
say "Badger, " x 8,
  "Mushroom, " x 2,
     "SNAKE! " x 2;
```

Grammar
+
Actions

Abstract Syntax Tree + Objects



Representing classes, routines, signatures, BEGIN-time results...

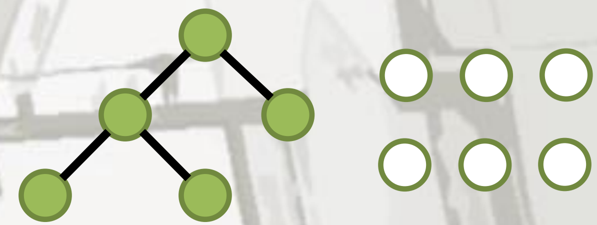
Rakudo Perl 6 Architecture

Perl 6 Source

```
say "Badger, " x 8,
  "Mushroom, " x 2,
     "SNAKE! " x 2;
```

Grammar
+
Actions

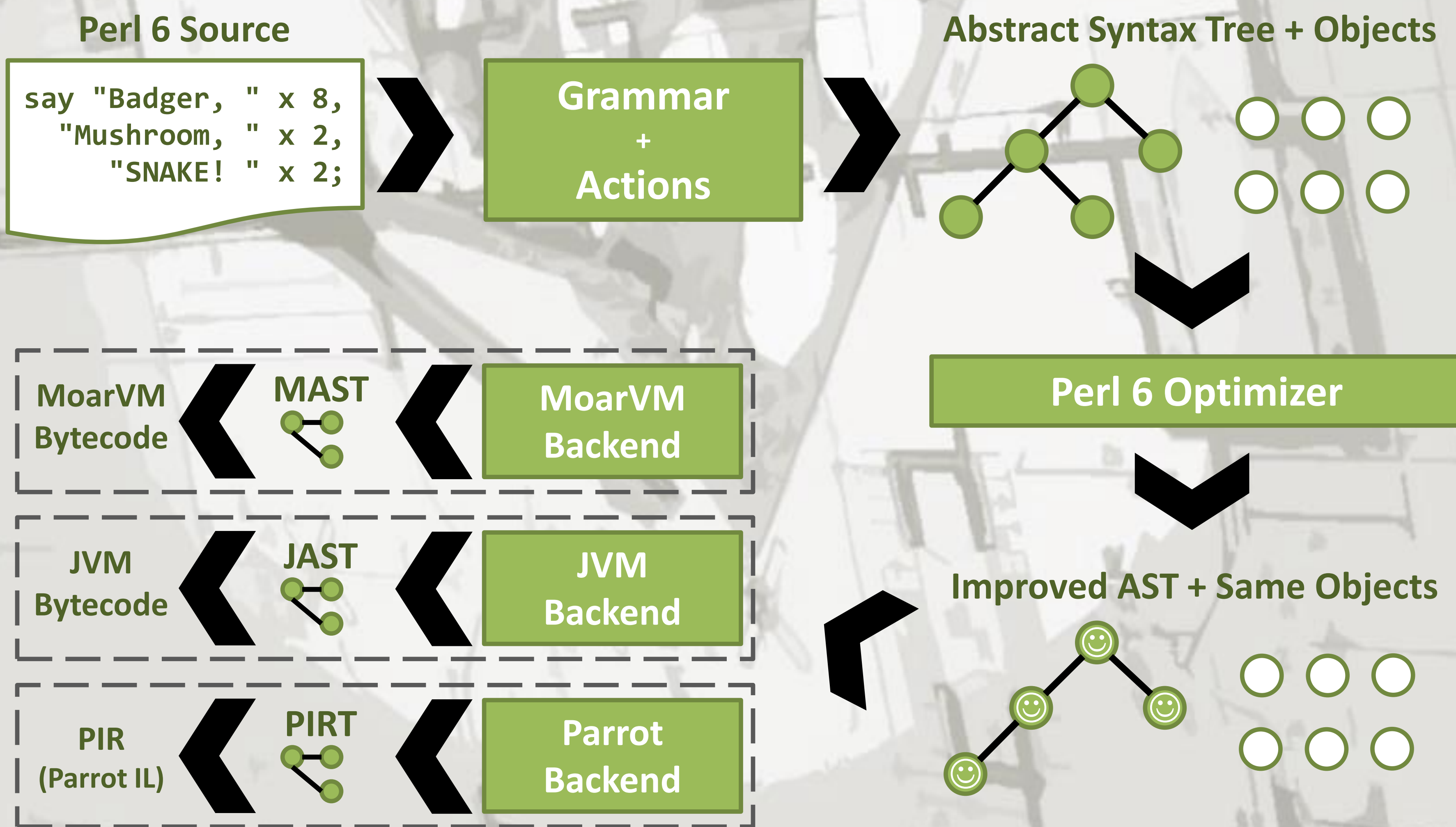
Abstract Syntax Tree + Objects



Perl 6 Optimizer

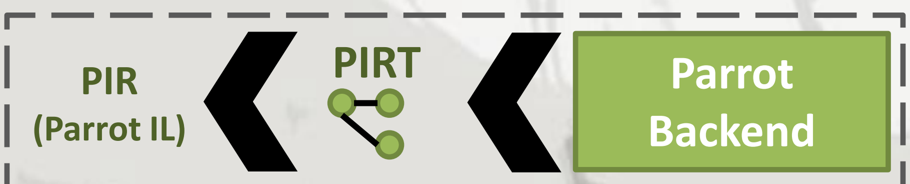
Improved AST + Same Objects

Rakudo Perl 6 Architecture



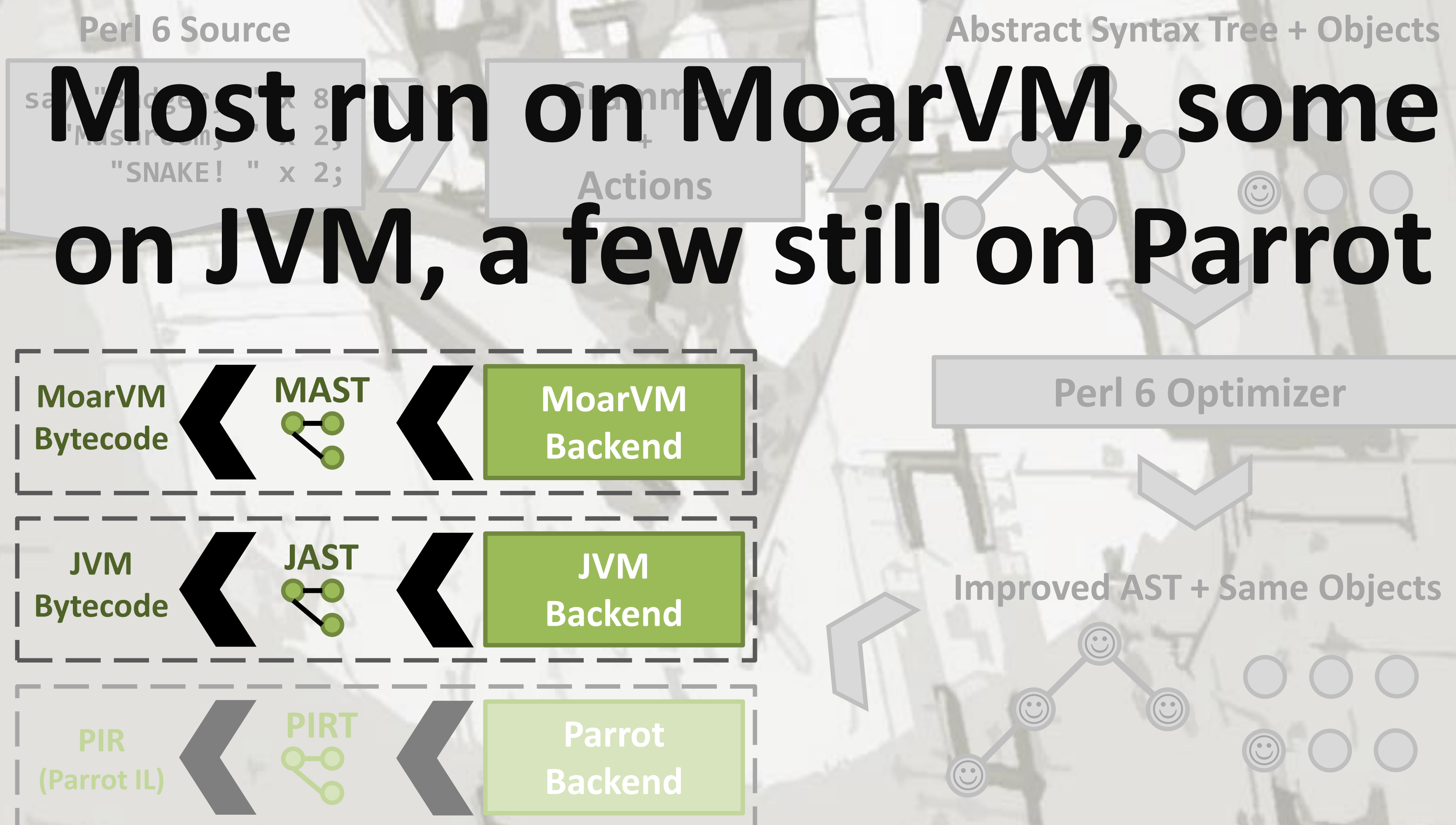
A year ago...

Just about everybody using Rakudo Perl 6 was using the Parrot backend; the others were in their infancy



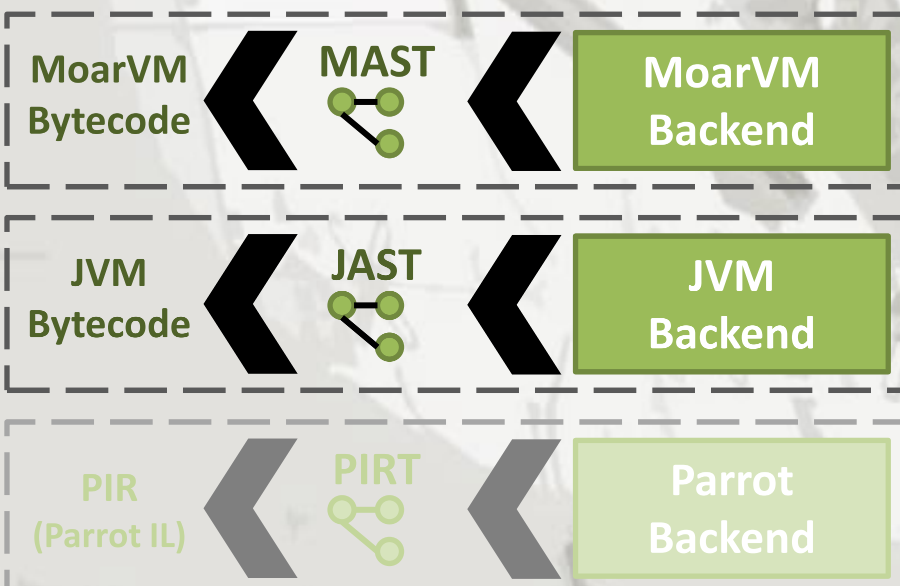
Today

Most run on MoarVM, some on JVM, a few still on Parrot



Today

Most run on MoarVM, some on JVM, a few still on Parrot



We couldn't have reached the point we're at today without Parrot

but

We couldn't have made the advances of the last couple of years without moving beyond it

CORE.setting

The Perl 6 built-ins are mostly written in Perl 6, with some calls down to (VM) primitives

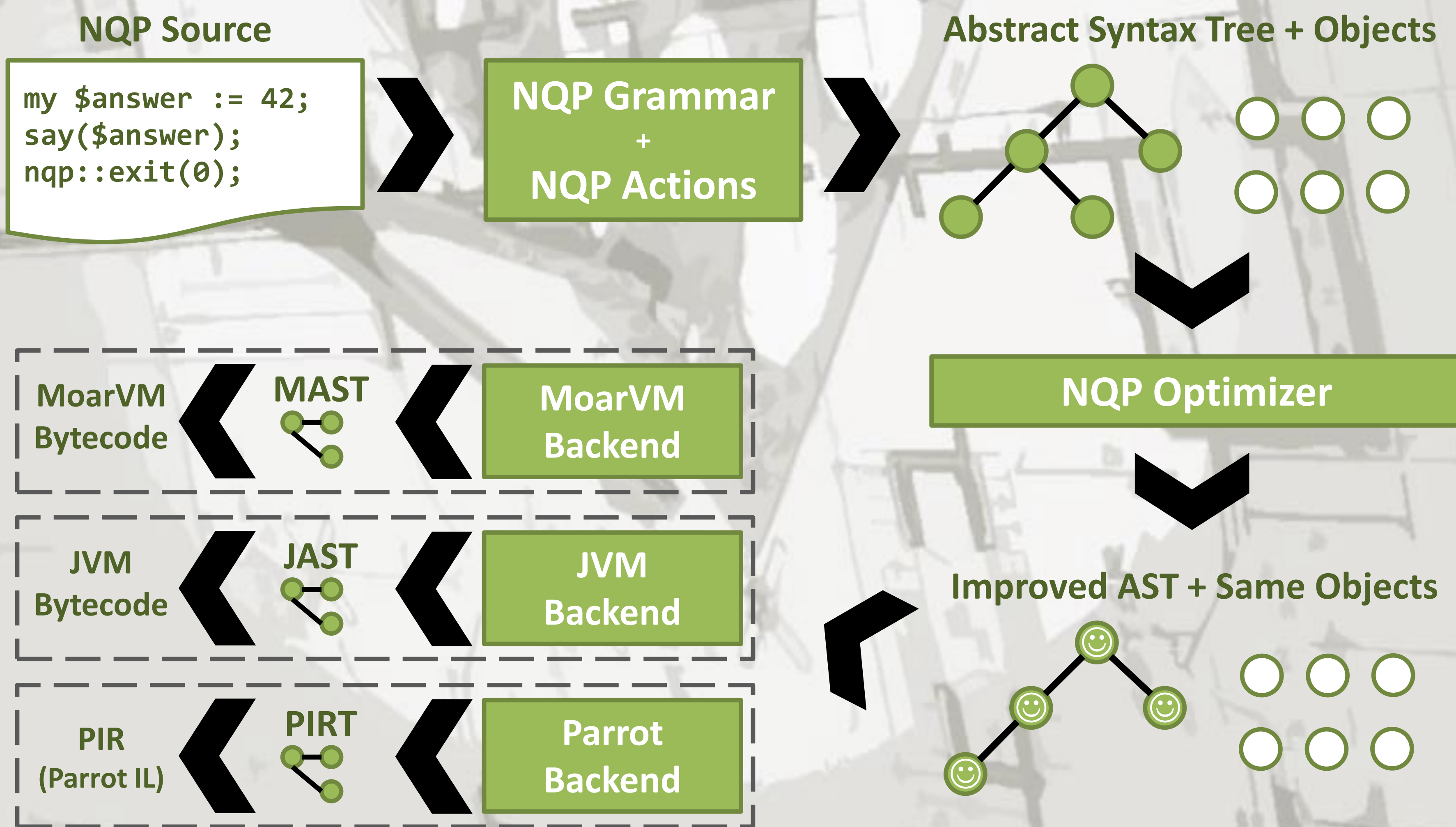
```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

NQP (Not Quite Perl 6)

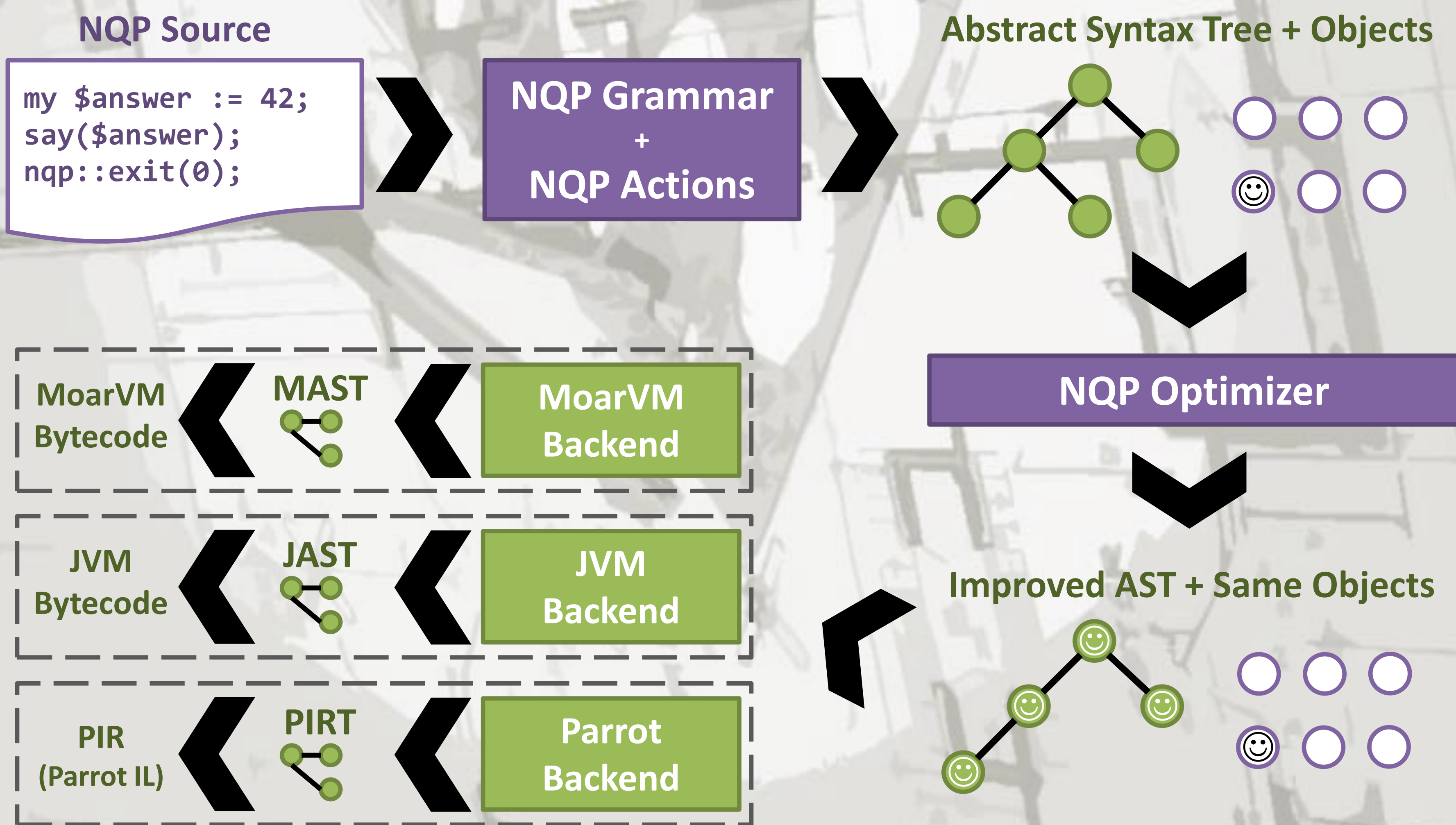
A small, easier-to-optimize, Perl 6 subset

Nearly all of Rakudo is NQP code (except CORE.setting)

NQP Architecture



NQP compiler + same toolchain



Where to improve things?

Took a holistic view of the whole pipeline, from source code through to runtime

Earlier stages can help later ones do their job better

We improved all the things!

Rakudo Optimizer	• More transformations, so Perl 6 programs run faster
NQP Optimizer	• More transformations, so NQP (and thus the Perl 6 compiler) run faster
Grammar Engine	• Improved it, so we parse NQP, Perl 6, and user's grammars more cheaply
Toolchain (AST and code-gen)	• Made AST nodes lighter and faster; improved quality of generated code
CORE.setting	• Improved many built-ins, so programs using them will run faster
MoarVM	• Made everything running on it - all the above - run faster

Today we'll focus in on...

Rakudo Optimizer	• More transformations, so Perl 6 programs run faster
NQP Optimizer	• More transformations, so NQP (and thus the Perl 6 compiler) run faster
Grammar Engine	• Improved it, so we parse NQP, Perl 6, and user's grammars more cheaply
Toolchain (AST and code-gen)	• Made AST nodes lighter and faster; improved quality of generated code
CORE.setting	• Improved many built-ins, so programs using them will run faster
MoarVM	• Made everything running on it - all the above - run faster

Rakudo optimizer examples

Turn range iterations into native integer loops

Before

```
for 1..100000 {
    do_it()
}
```

After

```
my int $i = 0;
my $body = { do_it() };
while $i < 100000 {
    $body($i);
    $i = $i + 1;
}
```

Rakudo optimizer examples

Devirtualize private method calls, resolving them at compile time (and whining about missing ones!)

Before

```
self!guts_thingy(42);
```

After

```
<A CONSTANT>(self, 42);
```

Rakudo optimizer examples

Routines contain more symbols than meets the eye...

You write...

```
method done() {
    $!winner || $!draw
}
```

But really it's...

```
method done(*%_) {
    my $_; # Topic
    my $!; # Error
    my $/; # Match
    $!winner || $!draw
}
```

Rakudo optimizer examples

We can statically see we'll never use the magicals…

Before

```
method done(*%_) {
    my $_; # Topic
    my $!; # Error
    my $/; # Match
    $!winner || $!draw
}
```

After

```
method done(*%_) {
    ~~my $_; # Topic~~
    ~~my $!; # Error~~
    ~~my $/; # Match~~
    $!winner || $!draw
}
```

Rakudo optimizer examples

...and %_ will never be used, so we can make it anonymous

Before

```
method done(*%_) {
    my $_; # Topic
    my $!; # Error
    my $/; # Match
    $!winner || $!draw
}
```

After

```
method done(*%_) {
    my $_; # Topic
    my $!; # Error
    my $/; # Match
    $!winner || $!draw
}
```

Rakudo optimizer examples

Furthermore, the `self` lexical holding the invocant is lowered to a normal local variable

This is a little faster to access, and easier on VM optimizers

Rakudo optimizer examples

As a final example, we also desugar simple junctions

Before

```
if $a < $lim1 & $lim2 {
    ...
}
```

After

```
TMP = $a;
if TMP < $lim1
&& TMP < $lim2 {
    ...
}
```

Note: these are tree transforms

I've used the program text to illustrate transformations

But in reality, we do them at the AST level, which is far more robust and straightforward

Note: th s

il n a
e filter!

"BOOM!!1!1'

I've u d to
illu ns

But in m at
the AS ar more
rob d straightforward

MoarVM

Started out as a naive interpreter of bytecode

MoarVM Bytecode (from NQP or Perl 6)

❯

Interpreter (Huge Switch Statement / Computed Goto)

❯

Stuff Happens

Interpretation: pretty easy

Validate bytecode to make sure ops and operands are valid on first call - and then just run!

```
OP(add_i):
    GET_REG(cur_op, 0).i64 = GET_REG(cur_op, 2).i64 +
                             GET_REG(cur_op, 4).i64;
    cur_op += 6;
    goto NEXT;
```

A bytecode deep dive

Let's start out by considering a simple Perl 6 builtin, the prefix ++ operation on an `Int`:

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

Compiling this code produces 23 instructions. Let's take it apart...

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// Ensure we've 1..1 args
checkarity 1, 1

// Grab the first arg into r1
param_rp_o r1, 0

// Coerce any NQP/other language
// values into Perl 6 types
hllize r4, r1
set r1, r4
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// Grab value out of Scalar
decont r4, r1

// Grab type; de-Scalar if needed
wval r5, 1, 34
decont r3, r5

// Ensure arg is an Int
istype r6, r4, r3
assertparamcheck r6
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// Also ensure it's not a type
// object (Perl 6 equivalent of
// undef, but typed)
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// Put arg into r0. Rakudo's
// optimizer lowered $a to a
// local, or we'd see a bindlex.
set r0, r1

// Ensure there's no named args.
paramnamesused

// Swallow any dispatch iterator.
takedispatcher r2
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// nqp::decont($a)
decont r3, r0

// 1 and Int objects, taken
// from constant table
wval r4, 0, 79
wval r5, 1, 34

// Actually do the addition
add_I r5, r3, r4, r5
```

A bytecode deep dive

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
multi prefix:<++>(Int:D $a is rw) {
    $a = nqp::add_I(nqp::decont($a), 1, Int);
}
```

```
// Assign result to $a, with a
// superstitious decont.
decont r4, r5
assign r0, r4

// Decont return value (legit)
p6decontrv r0, r0

// Return it
return_o r0
```

"No wonder it's slow!"

We could find a few ways to improve the generated code

However, they'd mostly kill off (cheap) data shuffling, not (more costly) checks

Enter Spesh

Spesh is the name for MoarVM's "type <u>speci</u>alizer"

(Why? If we called it "spec" everyone would say it wrong, or try to Google "Perl 6 spec")

Spesh seeks out hot code, sees what kinds of arguments it is given, and makes a specialized version.

Single Static Assignment

The first thing spesh does is get the code in SSA form, by giving registers "versions"

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
```

❯

```
checkarity 1, 1
param_rp_o r1(1), 0
hllize r4(1), r1(1)
set r1(2), r4(1)
```

Specialization walkthrough

Let's consider the case where `prefix:<++>` is called with a single argument: a `Scalar` container holding an `Int`

How will the code change?

Specialize by callsite

```
checkarity 1, 1
param_rp_o r1(1), 0
hllize r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
decont r3(1), r5(1)
istype r6(1), r4(2), r3(1)
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
assertparamcheck r6(2)
set r0(1), r1(2)
paramnamesused
```

We're doing a specialization for a callsite with a single object arg; toss checks!

Specialize by callsite

```
~~param_rp_o r1(1), 0~~
sp_getarg_o r1(1), 0
hllize r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
decont r3(1), r5(1)
istype r6(1), r4(2), r3(1)
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
assertparamcheck r6(2)
set r0(1), r1(2)
```

We know it's an object coming in, so use a cheaper op that skips boxing check.

Specialize by HLL

```
sp_getarg_o r1(1), 0
hllize r4(1), r1(1)
set r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
decont r3(1), r5(1)
istype r6(1), r4(2), r3(1)
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
assertparamcheck r6(2)
set r0(1), r1(2)
```

We know the incoming arg is a Perl 6 type, so we can avoid the HLL coercion.

Specialize by constant

```
sp_getarg_o r1(1), 0
set r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
~~decont r3(1), r5(1)~~
set r3(1), r5(1)
istype r6(1), r4(2), r3(1)
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
assertparamcheck r6(2)
set r0(1), r1(2)
```

This wval (a constant) is known not to be in a Scalar, so we can toss the decont.

Specialize by type

```
sp_getarg_o r1(1), 0
set r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
set r3(1), r5(1)
~~istype r6(1), r4(2), r3(1)~~
iconst_64 r6(1), 1
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
assertparamcheck r6(2)
set r0(1), r1(2)
```

We know r4(2) is an Int (from the arg), and r3(1) is Int, so the istype must be true.

Specialize by definedness

```
sp_getarg_o r1(1), 0
set r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
set r3(1), r5(1)
iconst_64 r6(1), 1
assertparamcheck r6(1)
decont r3(2), r1(2)
isconcrete r6(2), r3(2)
iconst_64 r6(2), 1
assertparamcheck r6(2)
set r0(1), r1(2)
```

We're also specializing for a defined Int, therefore isconcrete must be true.

Passed assertions redundant

```
sp_getarg_o r1(1), 0
set r4(1), r1(1)
set r1(2), r4(1)
decont r4(2), r1(2)
wval r5(1), 1, 34
set r3(1), r5(1)
iconst_64 r6(1), 1
assertparamcheck r6(1)
decont r3(2), r1(2)
iconst_64 r6(2), 1
assertparamcheck r6(2)
set r0(1), r1(2)
```

The assertion operations do nothing if given a true value - so they can go.

Dead code elimination

```
sp_getarg_o r1(1), 0
set r4(1), r1(1)
set r1(2), r4(1)
~~decont r4(2), r1(2)~~
~~wval r5(1), 1, 34~~
~~set r3(1), r5(1)~~
~~iconst_64 r6(1), 1~~
~~decont r3(2), r1(2)~~
~~iconst_64 r6(2), 1~~
set r0(1), r1(2)
```

We now have a number of unused values, and so can delete the ops that set them.

After some more opts...

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

›

```
sp_getarg_o r1, 0
set r4, r1
set r1, r4
set r0, r1
takedispatcher r2
sp_p6oget_o r3, r0, 16
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
set r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

Aside: if spesh were perfect…

```
checkarity 1, 1
param_rp_o r1, 0
hllize r4, r1
set r1, r4
decont r4, r1
wval r5, 1, 34
decont r3, r5
istype r6, r4, r3
assertparamcheck r6
decont r3, r1
isconcrete r6, r3
assertparamcheck r6
set r0, r1
paramnamesused
takedispatcher r2
decont r3, r0
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
decont r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
sp_getarg_o r1, 0
set r4, r1
set r1, r4
set r0, r1
takedispatcher r2
sp_p6oget_o r3, r0, 16
wval r4, 0, 79
wval r5, 1, 34
add_I r5, r3, r4, r5
set r4, r5
assign r0, r4
p6decontrv r0, r0
return_o r0
```

```
sp_getarg_o r0, 0
takedispatcher r2
sp_p6oget_o r3, r0, 16
wval r4, 0, 79
add_I r4, r3, r4, r4
sp_p6bind_o r0, 16, r4
return_o r4
```

Hand-optimized, but spesh should get close soon ☺

A real world...benchmark!

Consider the use of ++ in:

```
my $i = 0; while ++$i <= 1000000 { }
```

Naively, we must check that the specialized version we made is valid per call. ☹

Specializing the call

```
// Look up prefix:<++>
const_s r1(6), lits(&prefix:<++>)
getlexstatic_o r5(6), r1(6)
decont r8(2), r5(6)

// Fetch $x
getlex r7(2), lex(idx=0,outers=0)

// Call it
prepargs callsite(...)
arg_o liti16(0), r7(2)
invoke_o r7(3), r8(2)
```

Here's the bytecode we start off with.

Specializing the call

```
// Look up prefix:<++>
~~const_s r1(6), lits(&prefix:<++>)~~
~~getlexstatic_o r5(6), r1(6)~~
~~decont r8(2), r5(6)~~
sp_getspeshslot r5(6), sslot(7)

// Fetch $x
getlex r7(2), lex(idx=0,outers=0)

// Call it
prepargs callsite(...)
arg_o liti16(0), r7(2)
invoke_o r7(3), r8(2)
```

The callee never changes, so we can cache it.

Specializing the call

<pre>
// Look up prefix:<++>
sp_getspeshslot r5(6), sslot(7)

// Fetch $x
getlex r7(2), lex(idx=0,outers=0)

// Call it
prepargs callsite(...)
arg_o liti16(0), r7(2)
<s>invoke_o r7(3), r8(2)</s>
sp_fastinvoke_o r7(3), r8(2), 0
</pre>

Then, we invoke our special prefix ++ directly!

Inlining

In reality, we go a step further.

Since the prefix:<++> code is quite small, we simply inline it into the calling code - meaning we avoid making call frames!

But wait…

If the whole program is this:

```
my $i = 0; while ++$i <= 1000000 { }
```

And spesh looks for hot code by counting calls, how do we ever optimize the loop?

On Stack Replacement

If we detect a loop is hot, we:

Pause it
Build optimized code
Resize frame for any inlines
Resume in the optimized code

But we're still interpreting!

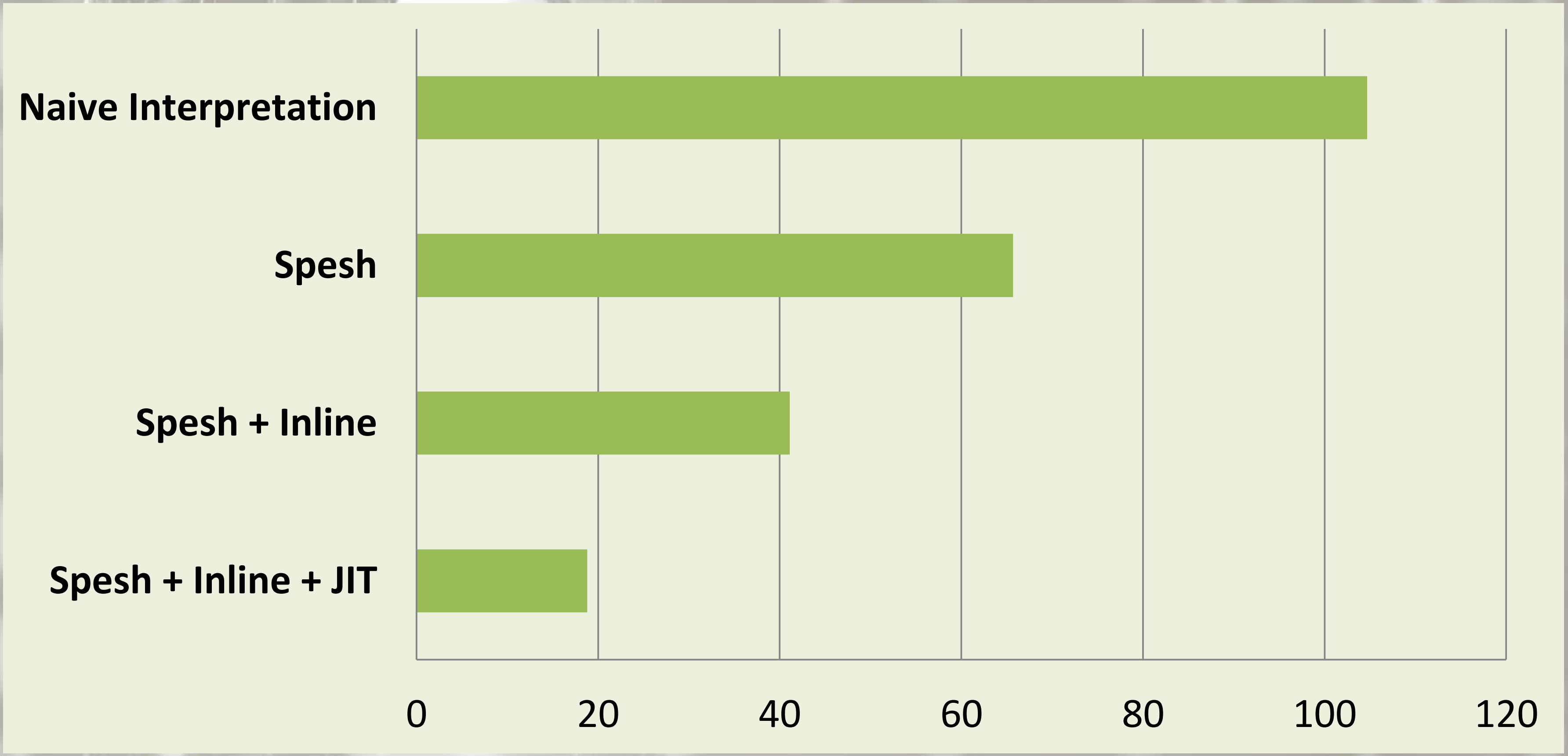
By now, we've got much better code for the interpreter to zip through. However, we're still interpreting it - which comes with a good bit of overhead!

Enter JIT compilation!

Thanks to an outstanding Google Summer of Code project, we can now turn much output of spesh into x64 machine code! ☺

Some timings

```
my $i = 0; while ++$i <= 100000000 { }
```

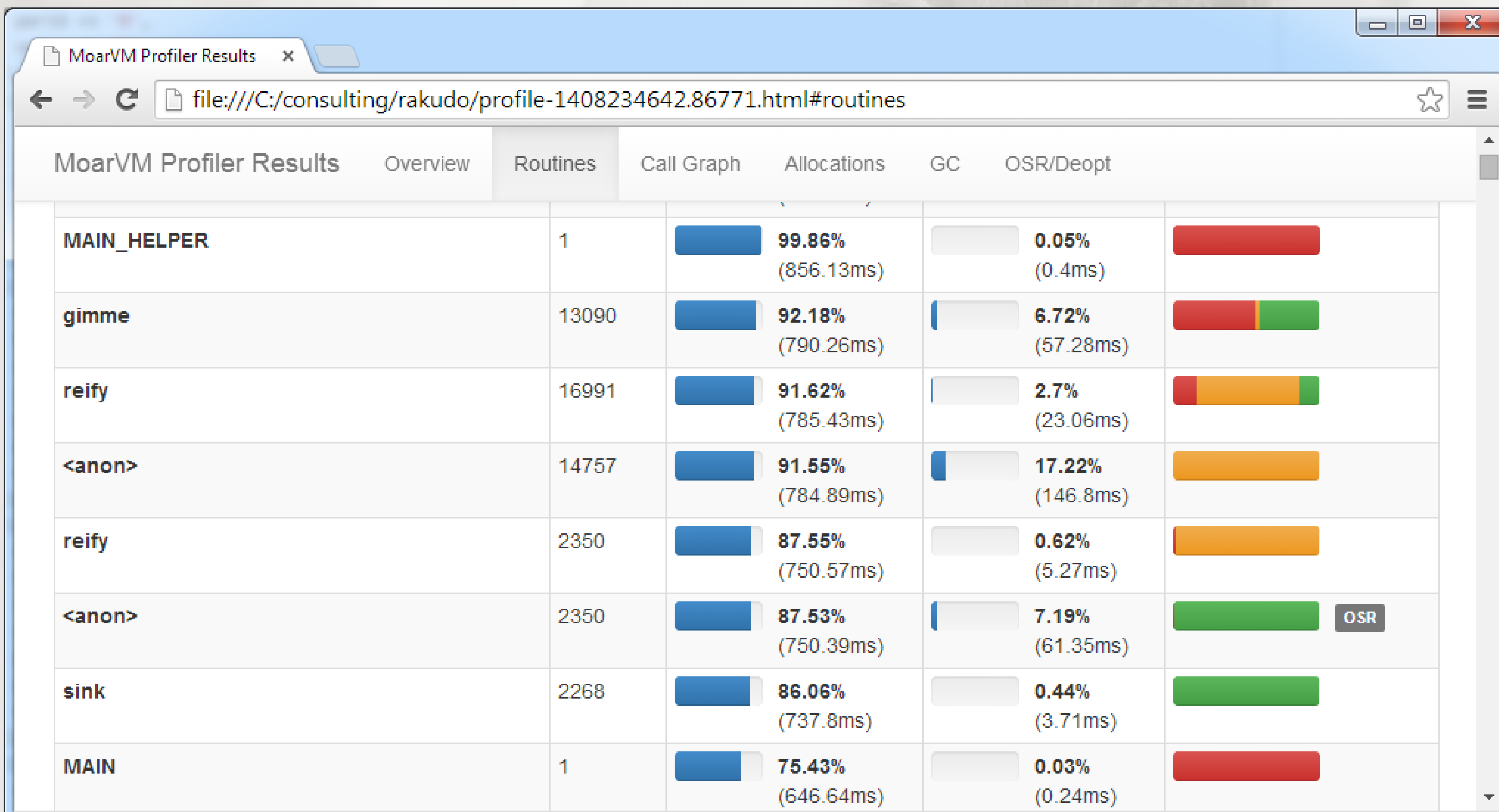


All this seems so magical!

Perl 6 *needs* a smart runtime.
The design relies on inlining to get acceptable performance.
But how can we know what is happening with our code?

Today, I'm happy to reveal…

...a MoarVM spesh-aware, JIT-aware, profiler!



Using the profiler

To profile runtime (normal):

```
perl6 --profile script.p6
```

Or compile time (for NQP and Rakudo developers, mostly):

```
perl6 --profile-compile script.p6
```

Results

Let's finish up with a look at some graphs, comparing:

Perl 5 v20

Rakudo on Parrot 2013.08

Rakudo on MoarVM 2014.08

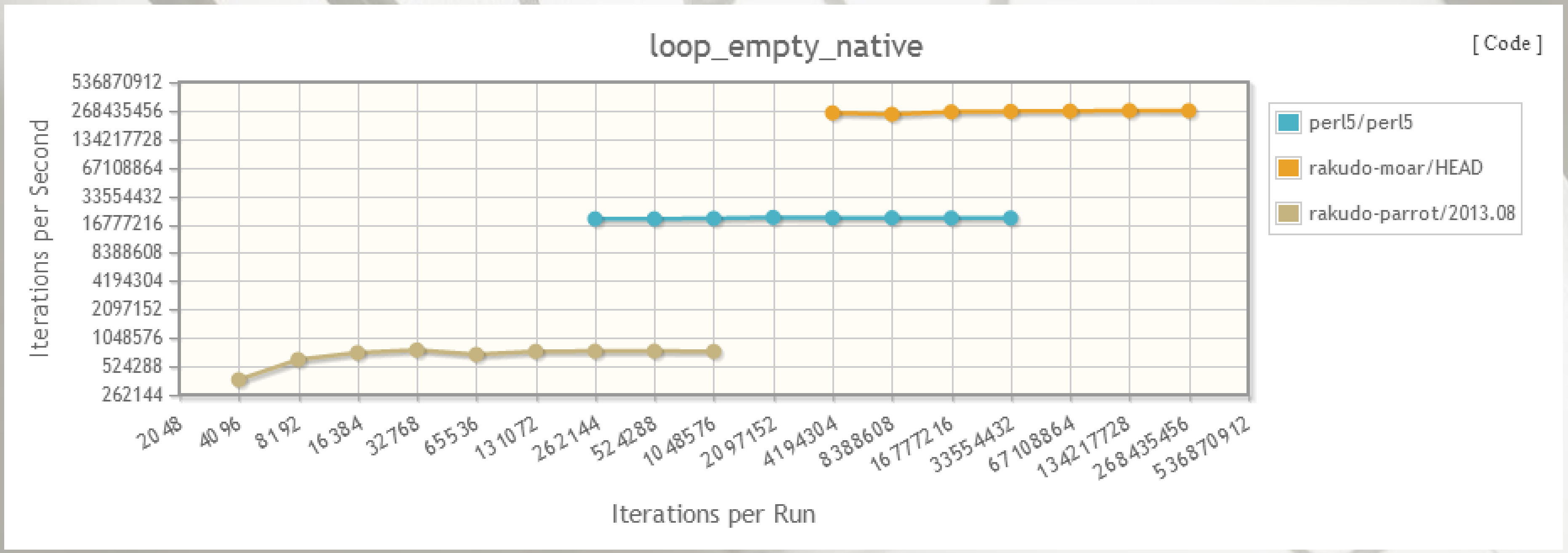
perl6-bench

The following graphs are produced by the excellent perl6-bench tool and suite

Not something I've worked on (so somewhat impartial ☺)

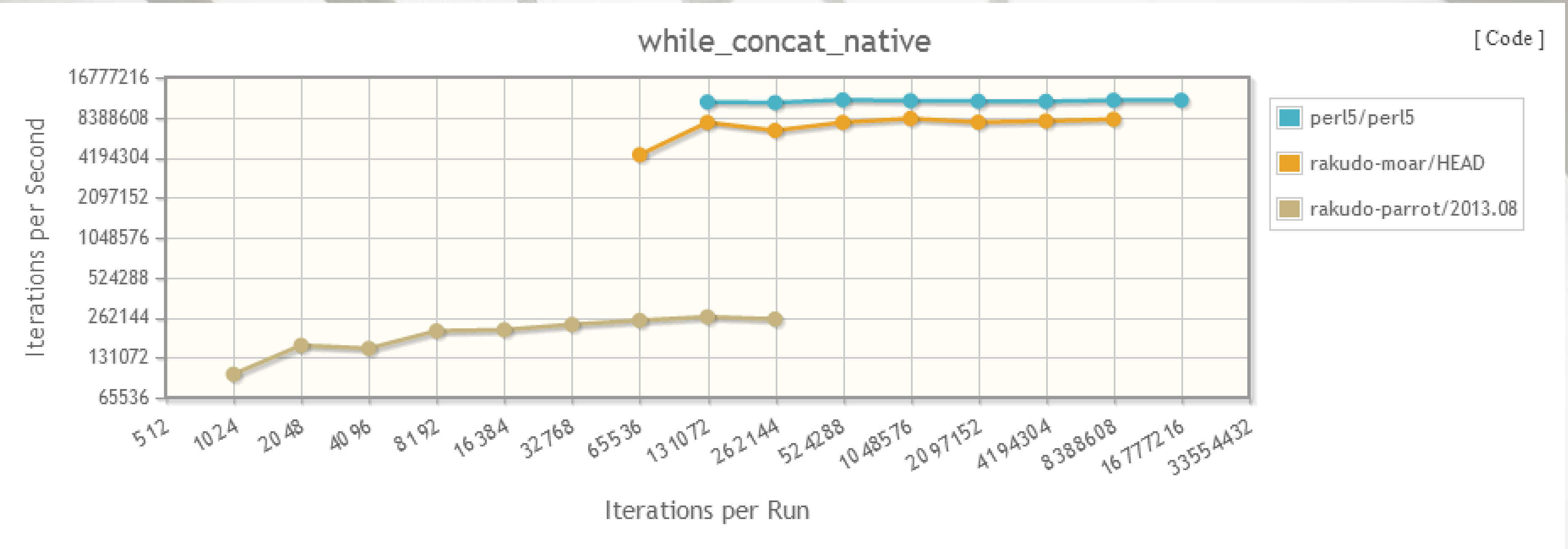
Great news: natives

Awesome, thanks to JIT. Here, we're 14x faster than Perl 5, and 355x faster than 2013.08!



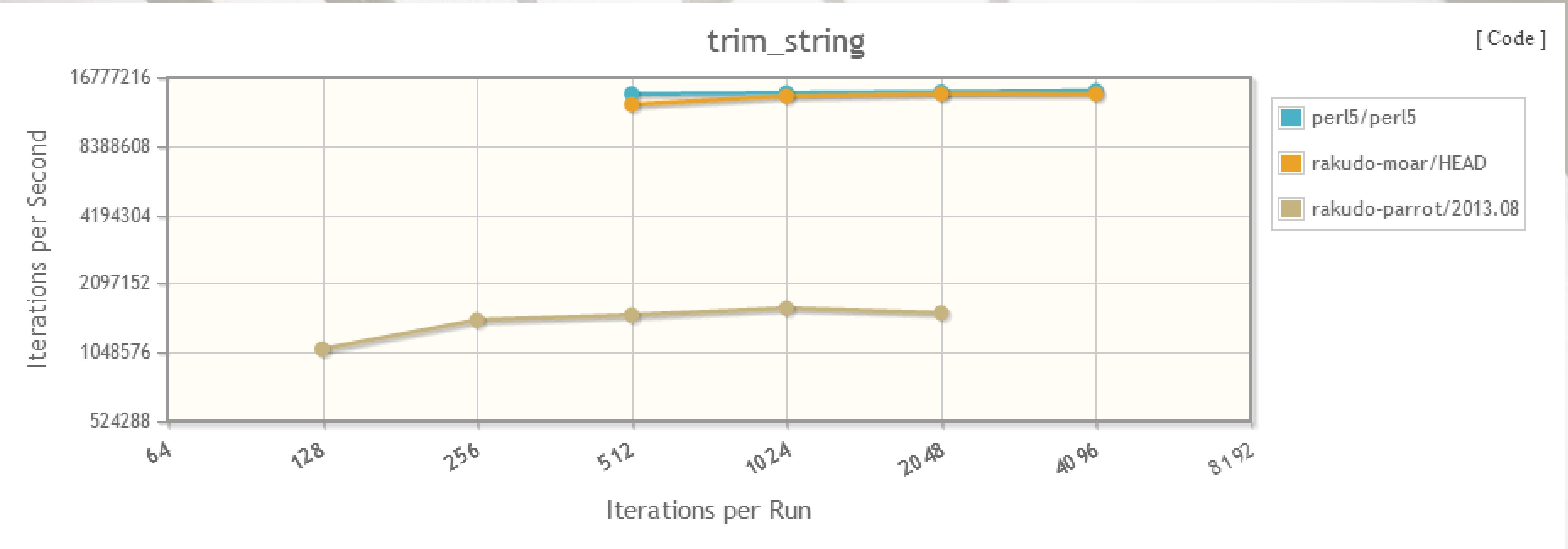
Good news: natives

Native loop and concatenation is about even with Perl 5, and 45x faster than 2013.08.



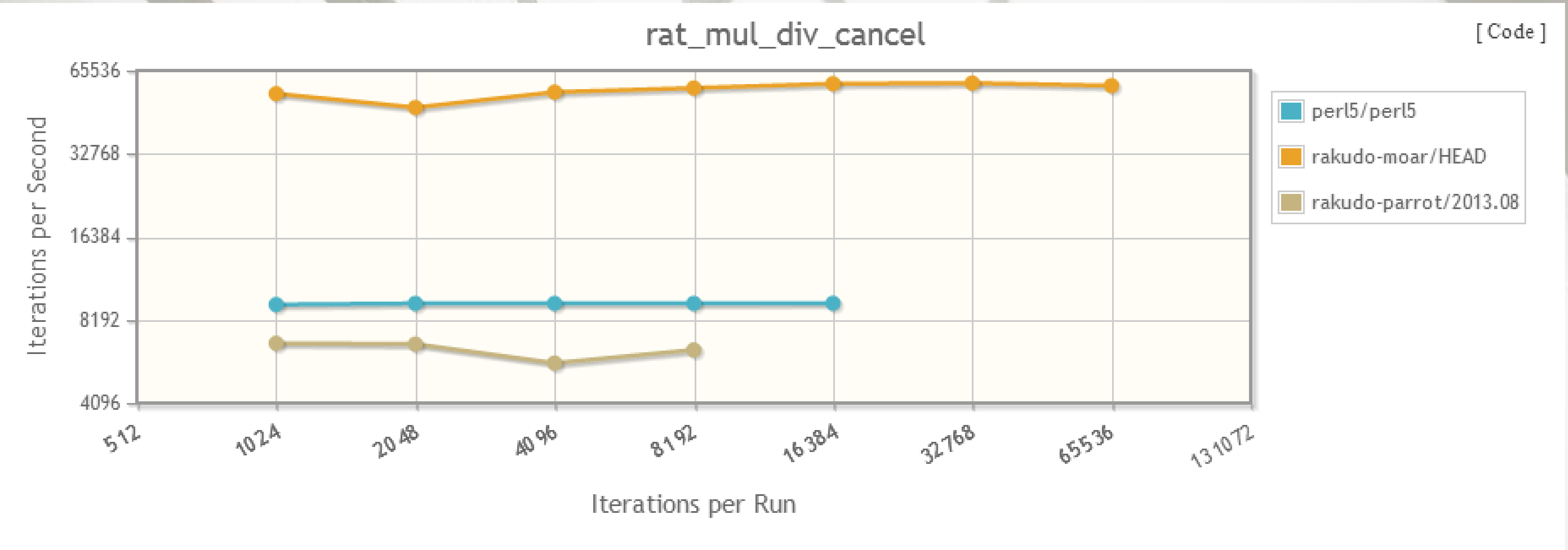
Good news: trim

Our trim built-in matches the usual Perl 5 idiom, and is 10x faster than 2013.08.



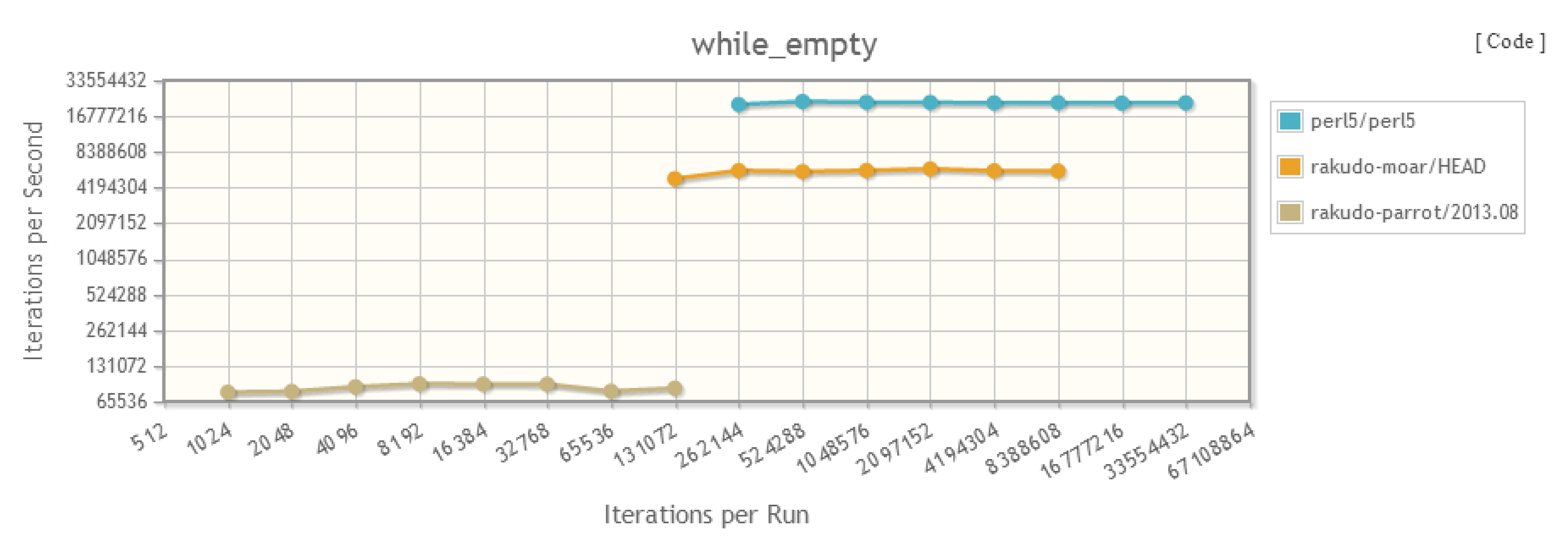
Great news: rationals

Our Rat (rational number) support is 6x faster than Perl 5, and 9x faster than 2013.08.



OK news: non-native loops

Perl 5 is 4x faster here. That's a big step forward; Perl 5 is 263x faster than 2013.08!



Aside: why is this hard?

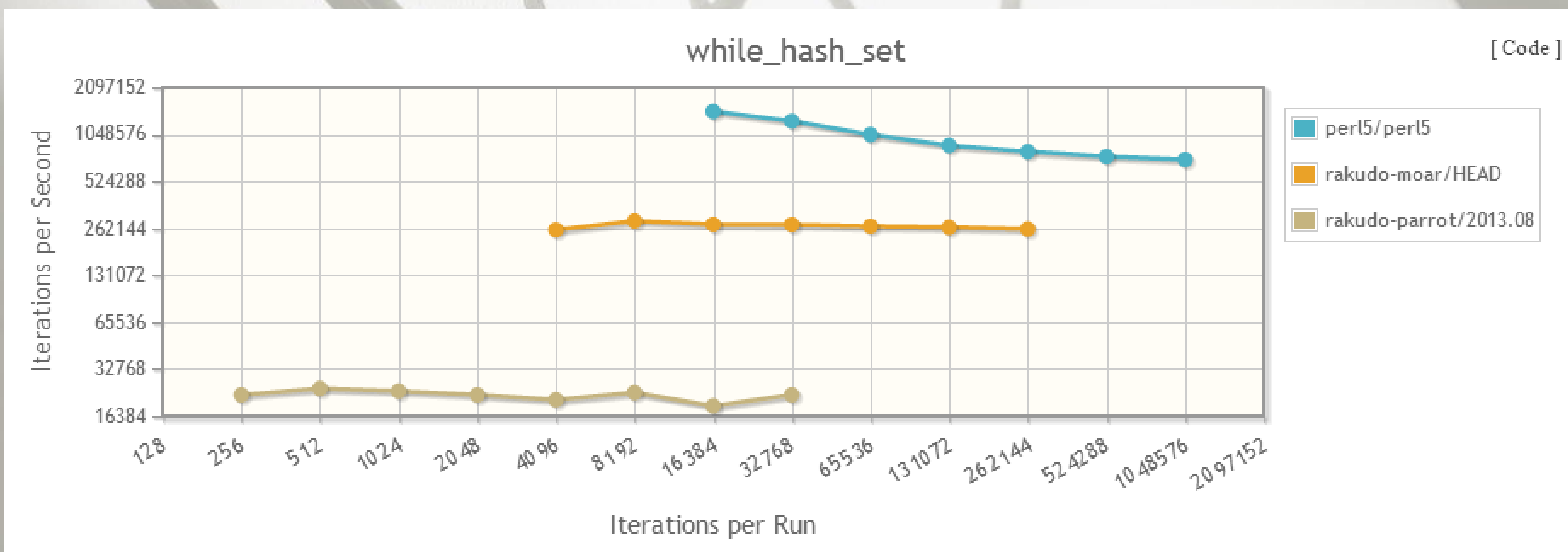
Why is this hard to make fast in Perl 6?

```
my $i = 0; while ++$i <= 100000000 { }
```

Firstly, because Int has big integer semantics. Secondly, because Int is an immutable, heap-allocated object by spec - and we do it that way. The silver lining: in the time Perl 5 does 4 ++s, we can allocate and GC an object!

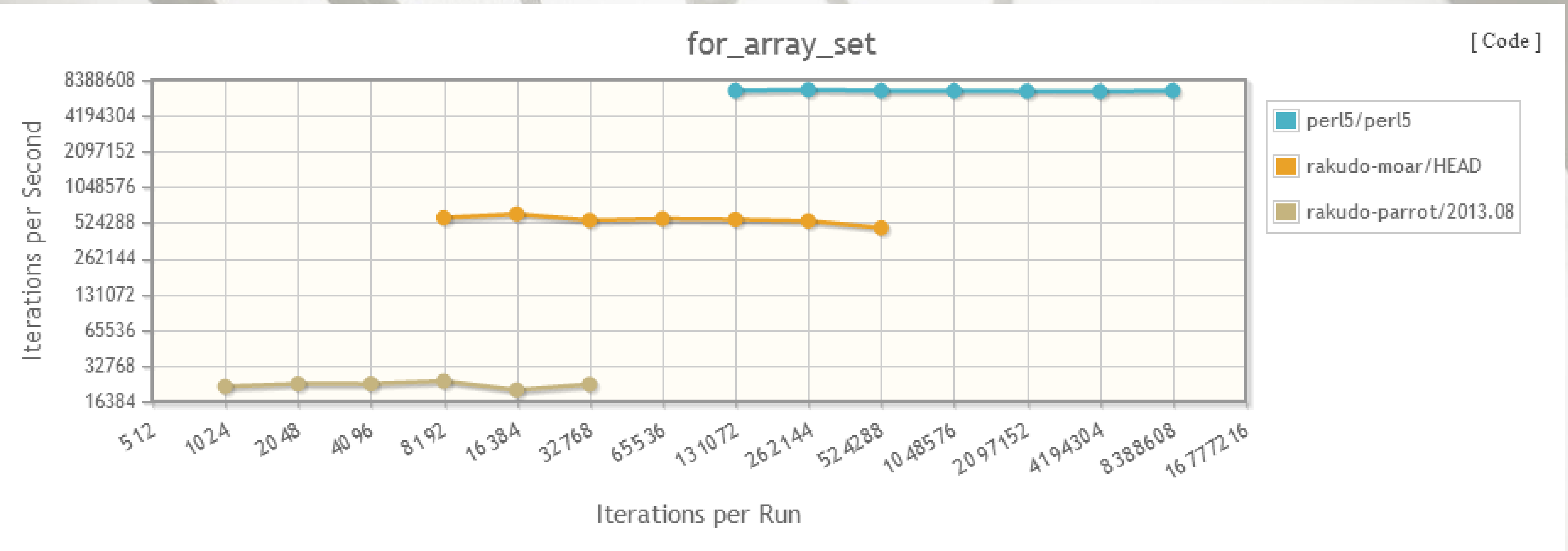
OK news: hashes

Perl 5 is 3x faster for this one. But again, we improved: it was 57x faster than 2013.08.



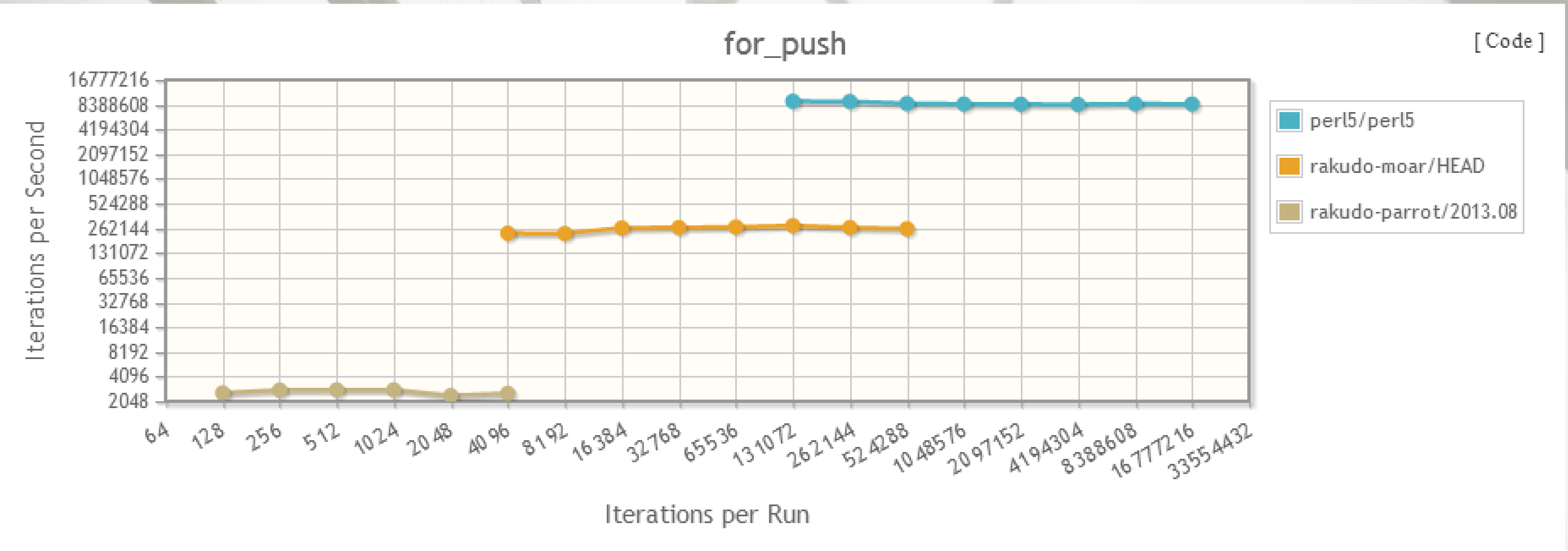
Bad news: arrays

2013.08 was about 300x slower than Perl 5.
2014.08 is still about 13x slower.



More bad array news: push

2013.08 was about 3,600x slower than Perl 5.
2014.08 is 34x slower. Better. But still sucks.



What's so hard about arrays?

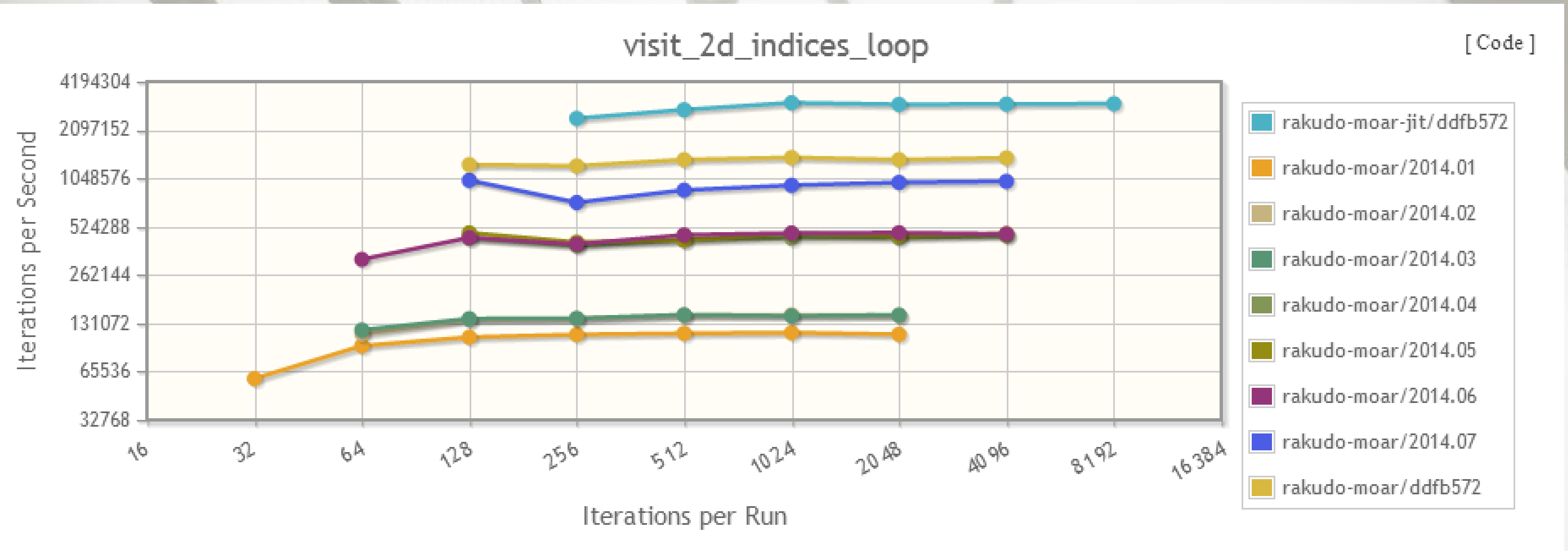
Perl 6 supports lazy lists.

That's great in that we can use normal for loops to do I/O.

However, we're still bad at pushing eager context down into list processing logic. Thankfully, this is now receiving attention.

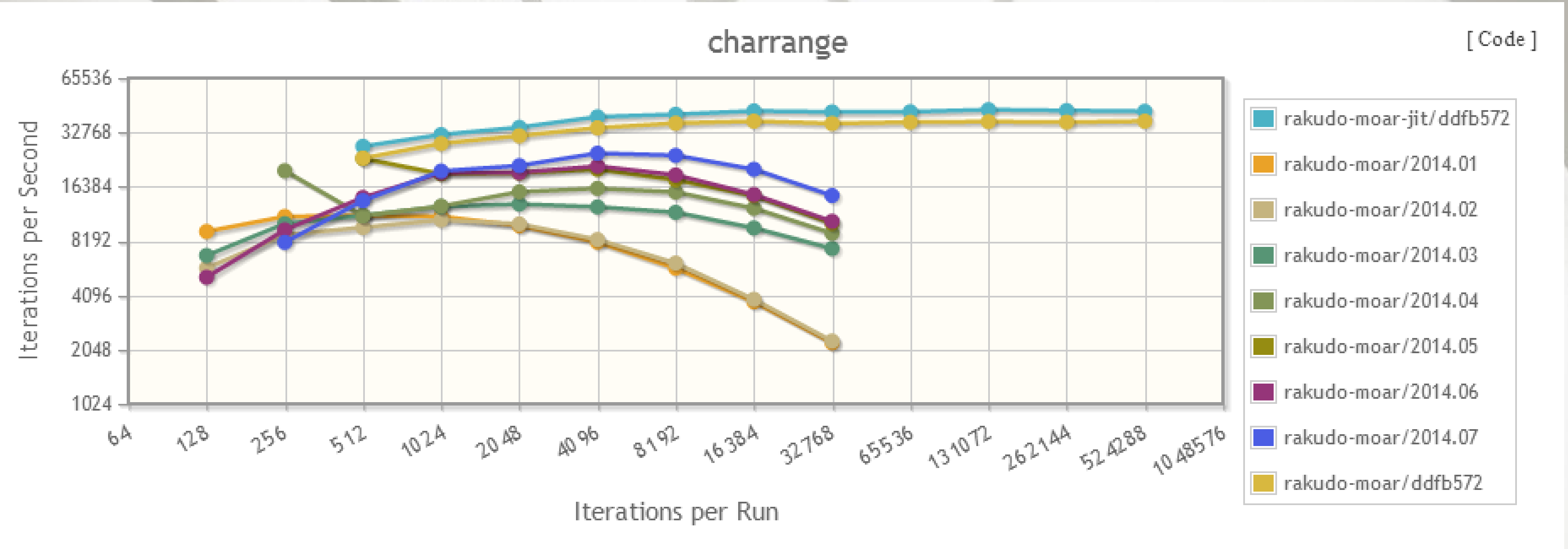
Steady improvement

This shows all releases since January on a 2D array indexing benchmark; we got 20x faster.



Algorithmic improvement

Sometimes, the improvement is algorithmic, as shown by the shapes here.



Overall...

We've made vast steps forward with Perl 6 performance in the last year

Much less likely to be an adoption barrier than a year ago; it depends how performance-sensitive the work you're doing is

Some strong areas, some weak ones

The future looks good

MoarVM, with its spesh and JIT, are enabling us to perform increasingly sophisticated dynamic optimization of code

perl6-becnh provides essential feedback

Now, we have a whole new treasure trove of information to open from the profiler!

Not *just* performance

It's been a year of advances on many other fronts for the Perl 6 project too:

Modules, the built-ins, documentation, JVM support, Pod, dozens of bugs fixed...

Come to my Sunday session to see what we've been doing with asynchrony and parallelism!


Questions?