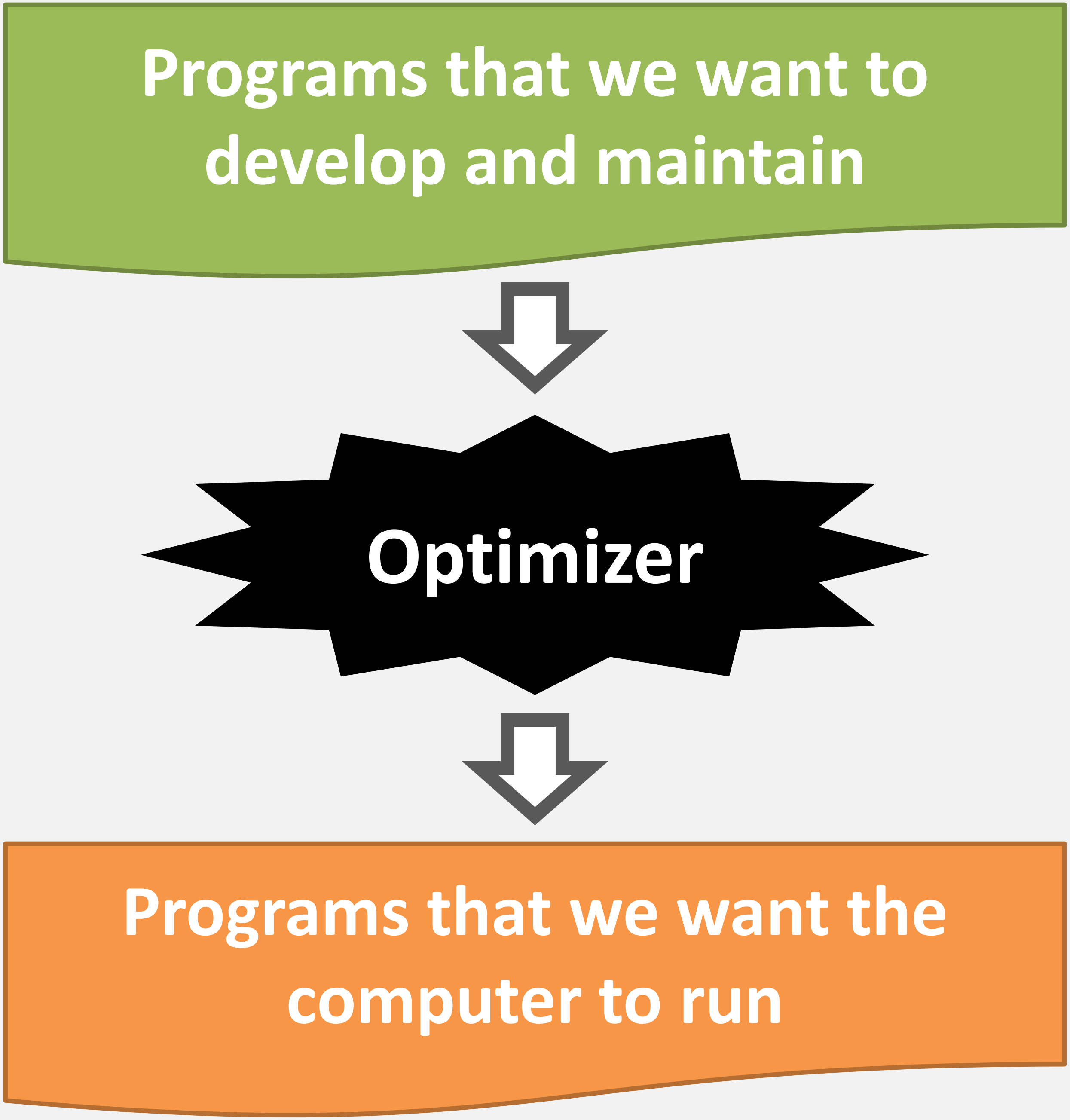
Escape analysis and related optimizations for Perl 6

Jonathan Worthington
Edument


Programs that we want to develop and maintain
Optimizer
Programs that we want the computer to run

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

Cost of method resolution: largely a solved problem

Most code is not polymorphic

Produce specialized versions for the precise type(s) that are really used

Resolve at optimization time, and inline smaller methods

Objects ☹

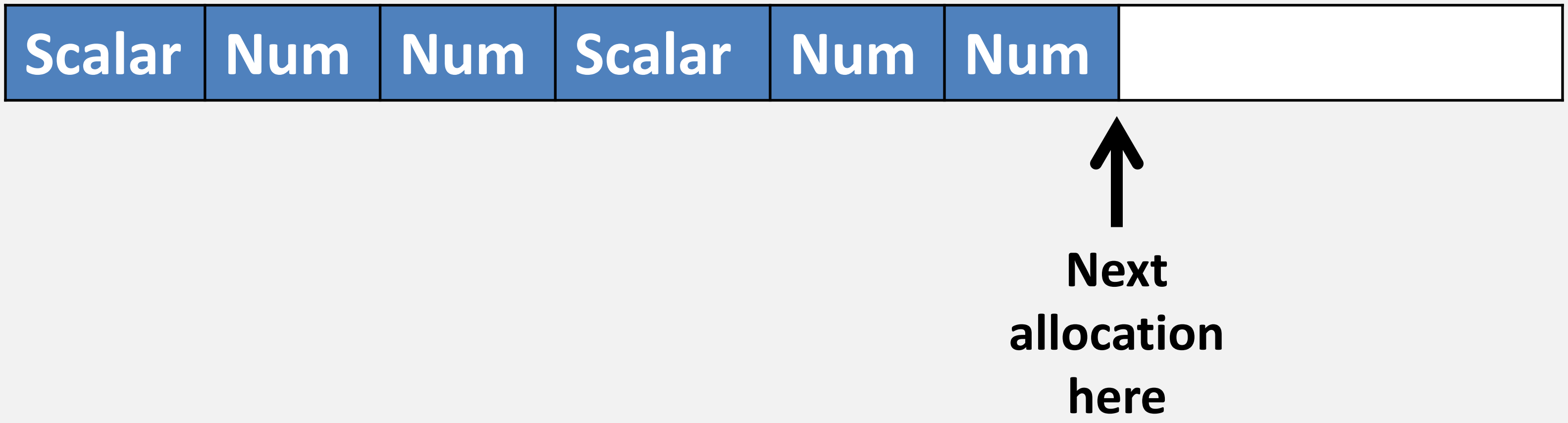
- ~~Cost of method resolution~~ Solved
- Allocations mean more memory pressure and more time doing garbage collection EA?
- Harder to analyze/optimize the program EA?

The memory challenge

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

The memory challenge

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



When it's full, we garbage collect

The memory challenge

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

The analysis challenge

```
# Assign a value to a property
$obj.x = 21;
# Call some method on the object.
$obj.do-stuff();
# Do we know what $obj.x is?
say 2 * $obj.x;
```

The analysis challenge

Objects may be referenced from many places

Anything holding the reference might modify it

Might even be done by code running in another thread

Speculative optimization

Partly thanks to objects, we often can't prove properties of programs in order to produce optimizations

However, we can speculatively optimize, so long as we can fall back to unoptimized code if we're wrong

Guards + deopt ☺

Keep statistics about what types tend to show up

If the type is stable, insert a guard: a quick check we got what we wanted

If the guard fails, deoptimize (fall back to the interpreter)

Guards + deopt ☹

Runtime cost to evaluate guards

Retention of state to enable deopt

Take up space in the instruction stream, hitting the instruction cache, and perhaps pushing code over inline limits

We can't reason about the scope and lifetimes of *all* objects.

But surely we can reason about *some* of them?

Yes!

And this is precisely what escape analysis does!

Take each object allocation in the code under consideration

Consider each instruction that involves that object

If an instruction causes the object to gain a reference that we can't track, we consider it to have <u>escaped</u>

But…

```
for @values -> $v {
    # $sv escapes to `+` below,
    # thus the resulting Num of
    # $v.sin also escapes ☹
    my $sv = $v.sin;
    do-something(1e0 + $sv);
}
```

Inlining! ☺

```
for @values -> $v {
    # $sv is only used in decont, so does not
    # escape; nor does the Num assigned into it
    my $sv = nqp::box_n(
        nqp::sin_n(nqp::unbox_n($v)),
        Num);
    # do-something not inlined, so Num escapes
    do-something(nqp::box_n(
        nqp::add_n(
            1e0,
            nqp::unbox_n(nqp::decont($sv))),
        Num));
}
```

Great, but what can we do with this information?

Scalar Replacement!

Not actually anything to do with Perl 6 `Scalar`, although it works on them

Create a local variable to hold each object attribute

Delete allocation, rewrite all attribute reads and writes into locals

Before Scalar Replacement

```
for @values -> $v {
    # $sv is only used in decont, so does not
    # escape; nor does the Num assigned into it
    my $sv = nqp::box_n(
        nqp::sin_n(nqp::unbox_n($v)),
        Num);
    # do-something not inlined, so Num escapes
    do-something(nqp::box_n(
        nqp::add_n(
            1e0,
            nqp::unbox_n(nqp::decont($sv))),
        Num));
}
```

Scalar Replacement: Step 1

```
# Approximation; this is done at bytecode level
for @values -> $v {
    # Scalar has $!value and $!descriptor
    my ($sv_value, $sv_descriptor);
    # Attribute write binds to a variable
    $sv_value := nqp::box_n(
        nqp::sin_n(nqp::unbox_n($v)),
        Num);
    # Attribute read uses the variable
    do-something(nqp::box_n(
        nqp::add_n(
            1e0, nqp::unbox_n($sv_value)),
        Num));
}
```

Scalar Replacement: Step 2

```
# Approximation; this is done at bytecode level
for @values -> $v {
    # Variables for Scalar attributes (unused!)
    my ($sv_value, $sv_descriptor);
    # Variable for the num inside the Num box
    my num64 $temp_value =
        nqp::sin_n(nqp::unbox_n($v));
    # Attribute read (unbox) uses the variable
    do-something(nqp::box_n(
        nqp::add_n(1e0, $temp_value),
        Num));
}
```

Scalar Replacement Result

```
# Approximation; this is done at bytecode level
for @values -> $v {
    do-something(nqp::box_n(
        nqp::add_n(
            1e0,
            nqp::sin_n(nqp::unbox_n($v))),
        Num));
}
```

2 less memory allocations per iteration

Got rid of the guard on the read of `$!value` from `Scalar`

In fact, the entire `Scalar` container simply went away

Got rid of some box/unbox

So, how is this actually done?

Unfortunately, it's a bit harder than the Perl 6 example made it look!

Hard enough that the full thing is still several months/headaches away

Let's start with the "basics", which are in the latest Rakudo/MoarVM releases

Two steps

1. Perform an abstract interpretation of the program, looking for object allocations, and preparing a set of transforms that, if applied, would result in scalar replacement of the allocated objects.
2. For the allocations that didn't escape, perform the transforms.

Abstract Interpretation

A program analysis technique

Simulate running the program, but without having real values

Pay attention to the instructions that are interesting for the analysis that is being performed

AI: allocations

```
fastcreate r10(2), Scalar
```

Allocate hypothetical replacement registers for each attribute, and record a transform to delete the allocation instruction

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2)	h1: $!value h2: $!descriptor	No	Delete allocation

AI: aliases

```
set r5(3), r10(2)
```

Add the target register to the set of those aliasing the allocation, and add a transform to delete the set instruction

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value h2: $!descriptor	No	Delete allocation Delete set

AI: write attribute

```
p6obind r5(3), offset(16), r2(1)
```

Add a transform that turns the attribute bind instruction into a set instruction into the replacement register; stash facts

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind ➔ set h1, r2(1)

AI: read attribute

```
p6oget r4(2), r5(3), offset(16)
```

Add a transform that will turn the attribute get instruction into a set instruction that reads the replacement register; track facts

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind ➔ set h1, r2(1) p6oget ➔ set r4(2), h1

+ facts(r4(2)) = facts(r2(1))

AI: guard

```
guardtype r4(2), Num
```

Check if the facts we propagated can be used to prove the type the guard asserts; add a transform to delete it if so

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind → set h1, r2(1) p6oget → set r4(2), h1 Delete guard

+ facts(r4(2)) = facts(r2(1))

AI: allocations, again

```
fastcreate r14(1), Num
```

This is just another allocation; make a new entry into the tracked allocations table

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind → set h1, r2(1) p6oget → set r4(2), h1 Delete guard
Num	r14(1)	h3: $!value (num64)	No	Delete Allocation

+ facts(r4(2)) = facts(r2(1))

AI: the return instruction

```
return_o r14(1)
```

The allocated value escapes by being returned

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind → set h1, r2(1) p6oget → set r4(2), h1 Delete guard
Num	r14(1)	h3: $!value (num64)	Yes	Delete Allocation

+ facts(r4(2)) = facts(r2(1))

Transform application

The transforms for the Num are discarded because it escapes. The Scalar ones are applied to the program.

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind ➔ set h1, r2(1) p6oget ➔ set r4(2), h1 Delete guard
~~Num~~	~~r14(1)~~	~~h3: $!value (num64)~~	~~Yes~~	~~Delete Allocation~~

But what if we deopt?

The code we performed scalar replacement on may have guards

The unoptimized code expects the real objects to be available

Therefore, we must *materialize* the required replaced objects on deopt

AI: deopt instructions (1)

```
guardconc r9(2), Int # deopt 12
```

Check if r10(2) and r5(3) are needed if we deopt at this point; if so, add a transform to add a materialization table entry

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind → set h1, r2(1) p6oget → set r4(2), h1 Deopt@12: h1,h2→r5

AI: deopt instructions (2)

```
guardconc r9(2), Int // deopt 12
```

Also need to make sure that replacement registers aren't optimized away

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2) r5(3)	h1: $!value *+ facts of r2(1)* h2: $!descriptor	No	Delete allocation Delete set p6obind ➔ set h1, r2(1) p6oget ➔ set r4(2), h1 Deopt@12: h1,h2→r5 Deopt usage of h1 @ 12 Deopt usage of h2 @ 12

The algorithm defined so far is implemented and enabled by default as of MoarVM 2019.02 ☺

Doesn't handle transitive references

Objects in a SSA version merge escape

Can't analyze code in loops

It's limited.

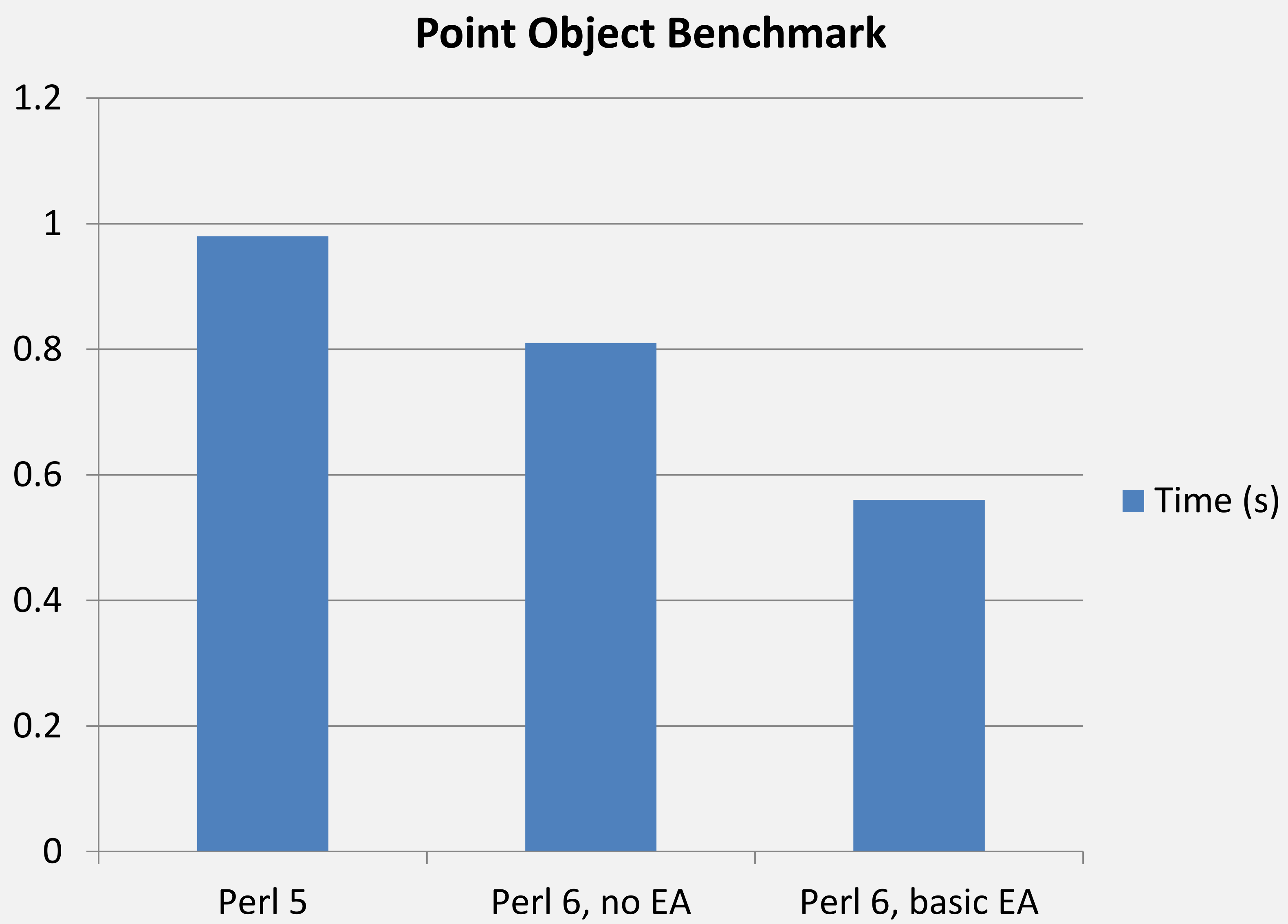
But on some benchmarks, it's still measurably effective.

```
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;
```

```perl
# Perl 5 version, for comparison
use v5.10;

package Point;
sub new {
    my ($class, %args) = @_;
    bless \%args, $class;
}
sub x {
    my $self = shift;
    $self->{x}
}
sub y {
    my $self = shift;
    $self->{y}
}

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


Point Object Benchmark
1.2
1
0.8
0.6
0.4
0.2
0
Perl 5
Perl 6, no EA
Perl 6, basic EA
Time (s)

And that's just from eliminating...

The `Scalar $p`

A `Hash` inside of construction

Various guards

The current algorithm still misses...

The `Point` object itself

The `Scalar`s of `Point`'s attributes

The `$total` and various `Int`s

And all of their associated guards

All of which will be possible in the future!


WOOAHHHH
WE'RE HALFWAY THERE!

In progress: transitive references

What if one allocation we might replace is bound into the attribute of another allocation we might replace?

```
fastcreate r10(2), Scalar
fastcreate r14(1), Num
p6obind_n r14(1), offset(16), r2(1)
p6obind r10(2), offset(16), r14(1)
```

In 2019.02: the Num is considered to escape

AI: transitive references

```
p6obind r10(2), offset(16), r14(1)
```

Add a transform to totally delete the bind, and add the replacement register as an alias

Allocated Type	Aliases	Replacements	Escapes	Transforms
Scalar	r10(2)	h1: $!value + facts(r14(1)) h2: $!descriptor	No	Delete allocation Delete p6obind
Num	r14(1) h1	h3: $!value (num64)	No	Delete Allocation

Transitive references: deopt?

Will need to materialize the "inner" object into the replacement register

To handle circular references, will also have to do two passes: allocate all objects, then populate attributes

Not implemented but...nothing seems broken. Need more tests!

And what next?

Partial Escape Analysis

Some objects only escape along some - perhaps rare - code paths

Or perhaps they escape near the end of a body of code that uses them

Do replacement up to the escape point

Need heuristics for when *not* to do it

Handle P6bigint

A Perl 6 `Int` isn't a straight boxing; it may be a native `int` or a big integer

In the big integer case, it's a pointer to a `malloc`'d bit of memory

We must not leak this!
We must not double-free this!

Handle SSA merges

An object register is assigned on both sides of a branch, and used after it

Naively: just materialize

Might be aliases to the same replacement

If they're the same type and both scalar replaced, can we avoid materializing?

Handle loops

Don't know what escapes on the back edge, because we didn't analyze that far

Take what we know as a first estimate

Once all back-edges are processed, do the abstract interpretation on the loop again

Iterate to a fixed point

Handle loops: OSR!

We use On Stack Replacement to replace the code running in a hot loop with the optimized version

It's like a reverse deopt

Consequence: we'll need to take scalar replaced objects apart during OSR!

Other representations?

For now, only considering P6opaque

Could we apply EA to a hash where all keys used are constants?

A small fixed-size array's slots?

A `CPointer` wrapper in native bindings?

In summary...

Perl 6 involves lots of objects

(Partial) Escape Analysis allows us to reason about their scope and lifetime

We can use this to deconstruct objects, eliminating or deferring their allocation

This "scalar replacement" allows for many further optimizations

Thank you!

Questions?