How does deoptimization help us go faster?

And other questions you were sensible enough not to ask!

Jonathan Worthington
An exploration of why making Perl 6 fast is hard, and some of the techniques and computer science we're throwing at the problem.
The Challenge
Let's write Perl 6 in Perl 6!
Let's write Perl 6 in Perl 6!

Ambitious language with lots of powerful abstractions and late binding
Let's write
Perl 6 in Perl 6!

Ambitious language with lots of powerful abstractions and late binding

A language we hadn't implemented yet, let alone made fast
Indexing an array (@a[\$x])
Indexing an array (@a[$x])

A multiple dispatch to the sub postcircumfix:<[ ]> (with candidates for one index, slicing, code (e.g. @a[*-1])...
Indexing an array (@a[$x])

A multiple dispatch to the sub postcircumfix:<[ ]> (with candidates for one index, slicing, code (e.g. @a[*-1])...

...which does a method call @a.AT-POS...
Indexing an array (@a[$x])

A multiple dispatch to the sub postcircumfix:<[ ]> (with candidates for one index, slicing, code (e.g. @a[*-1])...

...which does a method call @a.AT-POS...

...which gets the element and returns it if it already exists, or sets up a Scalar with an auto-vivification callback if not
Loop over the lines in a file
Loop over the lines in a file

Get an iterator and call `.pull-one` on it...
Loop over the lines in a file

Get an iterator and call .pull-one on it...

...which calls .consume-line-chars on the decoder (pluggable userspace encodings) and, if it fails, get bytes to refill the buffer...
Loop over the lines in a file

Get an iterator and call `.pull-one` on it...

...which calls `.consume-line-chars` on the decoder (pluggable userspace encodings!) and, if it fails, get bytes to refill the buffer...

...and then call the block of the loop, passing the line as an argument to it.
All these darn calls

In a language where...

Method resolution is pluggable
Type checking is pluggable
We have continuation-powerful constructs
Stack frames are first class
A mixin can change an object's type
Frames can have exit handlers (LEAVE etc.)
Rakudo Perl 6 Compiler Architecture
Perl 6 Source

Bytecode (for MoarVM, JVM, etc.)
Perl 6 Source

Compiler (written in NQP)

Bytecode (for MoarVM, JVM, etc.)
Perl 6 Source

Grammar + Actions

Bytecode (for MoarVM, JVM, etc.)
Perl 6 Source

Grammar + Actions

AST
(Abstract Syntax Tree)

Bytecode (for MoarVM, JVM, etc.)
Perl 6 Source

Grammar + Actions

AST (Abstract Syntax Tree)

Code Generation

Bytecode (for MoarVM, JVM, etc.)
Compiler is a Perl 6 program, running on the same VM instance (and thus in the same process) as the program it compiles

Scripts/one-liners: bytecode in memory

Modules: cache bytecode on disk (sounds easy; actually hard to have it Just Work)

EVAL - just a call into the compiler (also means bytecode has to be possible to GC)
Program Optimization
Actually, this wasn't the whole truth...
AST optimizer

Constant folding (calls to is PURE subs)

(Some) lexical to local lowering, plus flattening scopes where it won't matter

Inlining of native int/num/str operators

Assorted rewrites to constructs into cheaper equivalents that do the same
It has been said:

"Don't put off until runtime that which you could do at compile time"
But:

For scripts and one-liners, the language user doesn't experience compile time and runtime, just time.
And also:

When we compile a module, we know little about its usage patterns; they may vary wildly between different programs.
How many compile times?

We aren't limited to just one

Just In Time compilers give us another round of compilation
So, I'd argue:

Only do in this compile time something that a later compile time couldn't do better and/or more simply
Known
Unknowns
Even this simple module is packed with unknowns...

```perl
sub average-line-chars($handle) is export {
    my $total-chars = 0;
    my $total-lines = 0;
    for $handle.lines -> $line {
        $total-chars += $line.chars;
        $total-lines++;
    }
    return $total-chars / $total-lines;
}
```
We don’t know the types of the parameters

sub average-line-chars($handle) is export {
    my $total-chars = 0;
    my $total-lines = 0;
    for $handle.lines -> $line {
        $total-chars += $line.chars;
        $total-lines++;
    }
    return $total-chars / $total-lines;
}"
We don’t know the types of method invocants

```perl
sub average-line-chars($handle) is export {
    my $total-chars = 0;
    my $total-lines = 0;
    for $handle.lines -> $line {
        $total-chars += $line.chars;
        $total-lines++;
    }
    return $total-chars / $total-lines;
}
```
We don’t know the types of arguments to operators

sub average-line-chars($handle) is export {
   my $total-chars = 0;
   my $total-lines = 0;
   for $handle.lines -> $line {
      $total-chars += $line.chars;
      $total-lines++;
   }
   return $total-chars / $total-lines;
}
Even if we had type annotations, we could be passed a subtype (except for native types)

Anything we pass as an argument may get mixed into...
If we get passed a closure, we don't know what code is going to be invoked.

In a given use of a module, it might turn out to be the same every time.
We don't know if this loop will be hot or not

```perl
sub average-line-chars($handle) is export {
    my $totalChars = 0;
    my $totalLines = 0;
    for $handle.lines -> $line {
        $totalChars += $line.chars;
        $totalLines++;
    }
    return $totalChars / $totalLines;
}
```
In summary...

We don't know what to spend effort optimizing

We don’t know what cases to optimize it for
Dynamic problem? Dynamic solution!
Interpreter logging

Initially, run bytecode using an interpreter

Have various instructions log encountered types, code, etc.
Can logging be cheap enough?

Append 24-byte entries into a buffer until it is full.

Entries carry a call frame ID to allow stack reconstruction.
Optimization thread

Receives filled buffers

Threads place full log buffers into a concurrent queue

Optimization worker thread removes them one at a time
Aggregation

Replay the recorded events on a simulated call stack

Gradually build up statistics about types, callees, etc.
Example program

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

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

Close file handle

Calls pull-one on iterator to get each line

method pull-one() {
    # Slow path falls back to .get on the handle, which will replenish the buffer.
    $!decoder.consume-line-chars($!chomp) // ($!handle.get //IterationEnd)
}
Latest statistics for 'chars' (cuid: 4208, file: SETTING::src/core/Str.pm:2728)

Total hits: 468

Callsite 0x7f0b7089da60 (1 args, 1 pos)
Positional flags: obj

Callsite hits: 468

Maximum stack depth: 13

Type tuple 0
  Type 0: Str (Conc)
  Hits: 468
  Maximum stack depth: 13
Latest statistics for 'infix:<+>' (cuid: 3129, file: SETTING::src/core/Int.pm:245)

Total hits: 469

Callsite 0x7f0b7089da40 (2 args, 2 pos)
Positional flags: obj, obj

Callsite hits: 469

Maximum stack depth: 35

Type tuple 0
  Type 0: RW Scalar (Conc) of Int (Conc)
  Type 1: Int (Conc)
  Hits: 469
  Maximum stack depth: 35
Latest statistics for 'read-internal' (cuid: 9529, file: SETTING::src/core/IO/Handle.pm:220)

Total hits: 1  Not hot, won't optimize (yet)

Callsite 0x7f0b7089da40 (2 args, 2 pos)
Positional flags: obj, obj

Callsite hits: 1

Maximum stack depth: 16

Type tuple 0
  Type 0: IO::Handle (Conc)
  Type 1: Int (Conc)
Hits: 1
Maximum stack depth: 16
Latest statistics for 'defined' (cuid: 356, file: SETTING::src/core/Mu.pm:106)

Total hits: **475**  \textbf{Hot, but...}

Callsite 0x7f0b7089da60 (1 args, 1 pos)
Positional flags: obj

\begin{itemize}
  \item Callsite hits: 475
  \item Maximum stack depth: 32
  \item Type tuple 0
    \begin{itemize}
      \item Type 0: Scalar (Conc) of Any (TypeObj)
      \item Hits: 1  \textbf{Not on a Scalar holding Any...}
    \end{itemize}
\end{itemize}

...
Statistics for defined (2)

... Type tuple 4
  - Type 0: Str (TypeObj)
  - Hits: 2  **Nor on a Str type object...**
    Maximum stack depth: 14

Type tuple 5
  - Type 0: Int (Conc)
  - Hits: 1  **Nor on an Int**
    Maximum stack depth: 32

Type tuple 6
  - Type 0: Str (Conc)
  - Hits: 468  **But LOADS of calls on a Str!**
    Maximum stack depth: 13
Latest statistics for ' ' (cuid: 1, file: -e:3)

Total hits: 468

Callsite 0x7f0b7089da60 (1 args, 1 pos)
Positional flags: obj

Callsite hits: 468

Maximum stack depth: 12

Type tuple 0
  Type 0: Str (Conc)
  Hits: 468  
  Always given a Str

Maximum stack depth: 12

...
Statistics for loop body (2)

Logged at offset:

68:
  468 x type Scalar (Conc)
76:
  468 x type Str (Conc)
110:
  468 x type Int (Conc)
    468 x static frame 'chars' (4208)
    468 x type tuple:
      Type 0: Str (Conc)
144:
  468 x type Int (Conc)
  468 x static frame 'infix:<+>' (3129)
  468 x type tuple:
    Type 0: RW Scalar (Conc) of Int (Conc)
    Type 1: Int (Conc)

Always same chars method, always Str \(\rightarrow\) Int
Planning

The statistics are used to plan what code to optimize, and what cases to optimize it for.
The total number of calls to a given block or routine provides an indication of whether to consider it further; it's weighed up against bytecode size.
We can classify a callsite, or the overall use of a routine, as monomorphic, polymorphic, and megamorphic.
Monomorphic

Only a single type (or tuple of types) is observed (or the outliers are so few we might as well consider it so)
Polymorphic

A few different types (or tuples of types) are observed (again, we're willing to overlook the odd outlier)
Megamorphic

Many different types show up without any being notably more common
Observed type specialization of 'infix::<+>' (cuid: 3129, file: SETTING::src/core/Int.pm:245)

The specialization is for the callsite: 
Callsite 0x7f0b7089da40 (2 args, 2 pos) 
Positional flags: obj, obj

It was planned for the type tuple: 
  Type 0: RW Scalar (Conc) of Int (Conc) 
  Type 1: Int (Conc) 
Which received 469 hits (100% of the 469 callsite hits).

The maximum stack depth is 35. **Totally monomorphic**
Observed type specialization of 'defined' (cuid: 356, file: SETTING::src/core/Mu.pm:106)

The specialization is for the callsite:
Callsite 0x7f0b7089da60 (1 args, 1 pos)
Positional flags: obj

It was planned for the type tuple:
  Type 0: Str (Conc)
Which received 468 hits (98% of the 475 callsite hits).

The maximum stack depth is 13. **Monomorphic-ish**
Monomorphomorphic/polymorphic

Can generate versions of the code specialized by input type

Will be one or just a few of them; worth the work/RAM
Megamorphic

Not worth producing type specializations

But can still do some other optimizations
In the future...

We'll analyze when a megamorphic sub/method is monomorphic/polymorphic in some arguments (this shows up in array/hash assignments)
Specialization

Graph
So, we've decided what we're going to optimize and, typically, what types we'll produce specializations for.

What next?
We need to turn the bytecode into a form that's ideal for analysis and transformation.
Basic blocks

Sequences of instructions that do not involve flow control (such as a branch or an exception throw) or invocation (calling things)
Basic blocks and Perl 6

A lot of operations are what we've called invokish - they may lead to a function call

(For example, decont of a Scalar won't, but of a Proxy will)
checkarity liti16(1), liti16(1)
param_rp_o r1, liti16(0)
decont r8, r1
wval r9, liti16(1), liti16(35) (P6opaque: Str)
istype r10, r8, r9
assertparamcheck r10
decont r9, r1
set r0, r9
param_sn r2
takedispatcher r3
wval r4, liti16(1), liti16(35) (P6opaque: Str)
getattr_s r5, r0, r4, lits($!value), liti16(0)
chars r6, r5
p6box_i r4, r6
wval r7, liti16(1), liti16(37) (P6opaque: Int)
...
May invoke (Proxy?)
May invoke (Proxy?)

May invoke (subset?)
May invoke (Proxy?)

May invoke (subset?)

May call error generator
May invoke (Proxy?)

May invoke (subset?)

May call error generator

May invoke (Proxy?)
Control Flow Graph

Basic blocks are nodes

Put an edge when control may flow from one basic block to another
Successors and predecessors

The successors of a basic block are those we may go to.

The predecessors of a basic block are those we may come from.
Control exceptions

All basic blocks in the region covered by a control exception (next, last, etc.) are given the basic block of the handler as a successor.
Non-control exceptions

For now, their handlers are all linked from an empty "entry point" basic block.

This is imprecise, but safe; we'll see why shortly...
Dominance

Basic block $A$ dominates basic block $B$ if every possible path through the CFG from the entry to $B$ goes through $A$. 
<table>
<thead>
<tr>
<th>Block</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>BB1, BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>Block</td>
<td>Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB1, BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>Block</td>
<td>Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB1, BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td>BB3</td>
</tr>
<tr>
<td>BB4</td>
<td>BB4</td>
</tr>
</tbody>
</table>

Diagram: BB1 → BB2 → BB3 → BB4 → BB5 → BB6
<table>
<thead>
<tr>
<th>Block</th>
<th>Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>BB1, BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td>BB3</td>
</tr>
<tr>
<td>BB4</td>
<td>BB4</td>
</tr>
<tr>
<td>BB5</td>
<td>BB5, BB6</td>
</tr>
<tr>
<td>Block</td>
<td>Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB1, BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td>BB3</td>
</tr>
<tr>
<td>BB4</td>
<td>BB4</td>
</tr>
<tr>
<td>BB5</td>
<td>BB5, BB6</td>
</tr>
<tr>
<td>BB6</td>
<td>BB6</td>
</tr>
</tbody>
</table>
Strict dominance

Just means excluding block's dominance of themselves
<table>
<thead>
<tr>
<th>Block</th>
<th>Strictly Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td></td>
</tr>
<tr>
<td>BB4</td>
<td></td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
</tbody>
</table>
Immediate dominance

Basic block A immediately dominates Basic Block B if it strictly dominates it, but does not strictly dominate another BB that strictly dominates it.
<table>
<thead>
<tr>
<th>Block</th>
<th>Strictly Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td></td>
</tr>
<tr>
<td>BB4</td>
<td></td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Strictly Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB2, BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5, BB6</td>
</tr>
<tr>
<td>BB3</td>
<td></td>
</tr>
<tr>
<td>BB4</td>
<td></td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Immediately Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB2</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5</td>
</tr>
<tr>
<td>BB3</td>
<td>BB4</td>
</tr>
<tr>
<td>BB4</td>
<td>BB5</td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
</tbody>
</table>
Dominance tree

The immediate dominator of each basic block is unique

Thus they form a tree, aka the dominance tree
<table>
<thead>
<tr>
<th>Block</th>
<th>Immediately Dominates</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>BB2</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5</td>
</tr>
<tr>
<td>BB3</td>
<td>BB4</td>
</tr>
<tr>
<td>BB4</td>
<td>BB6</td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Immediately Dominates</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
</tr>
<tr>
<td>BB1</td>
<td>BB2</td>
</tr>
<tr>
<td>BB2</td>
<td>BB3, BB4, BB5</td>
</tr>
<tr>
<td>BB3</td>
<td>BB3</td>
</tr>
<tr>
<td>BB4</td>
<td></td>
</tr>
<tr>
<td>BB5</td>
<td>BB6</td>
</tr>
<tr>
<td>BB6</td>
<td></td>
</tr>
</tbody>
</table>
Dominance children

Successor and predecessor are refer to the CFG

Parent and children refer to the dominance tree
Why bother?

The dominance tree is a good order to visit basic blocks to propagate type information.

But there's another reason...
Static Single Assignment

Form where each variable only has one (textual) assignment in the program

Can form it by renaming
SSA in linear code: easy

Bump version per assign

```
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 in linear code: easy

Bump version per assign

```
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 in linear code: easy

Bump version per assign

```plaintext
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 in linear code: easy

Bump version per assign

```plaintext
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 in linear code: easy

Bump version per assign

```
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 in linear code: easy

Bump version per assign

```c
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)
```
What about flow control?

```
gt_i r2, r1, r0
if r2 goto BB(3)
set r0, r1
return_i r0
```
What about flow control?

```plaintext
gt_i r2(1), r1(1), r0(1)
if r2(1) goto BB(3)

set r0, r1

return_i r0
```
What about flow control?

```plaintext
gt_i r2(1), r1(1), r0(1)
if r2(1) goto BB(3)
```

```plaintext
set r0(2), r1(1)
```

```plaintext
return_i r0
```
gt_i r2(1), r1(1), r0(1)
if r2(1) goto BB(3)

set r0(2), r1(1)

return_i r0(????)
PHI functions

At such "join points" in the graph, we insert PHI functions

These "merge" the incoming values
What about flow control?

gt_i r2(1), r1(1), r0(1)
if r2(1) goto BB(3)

set r0(2), r1(1)

PHI r0(3), r0(1), r0(2)
return_i r0(3)
Placing PHI functions is also driven by dominance (of note, dominance frontiers - the places that a basic block's strict dominance ends)
Why SSA?

Associate facts with each SSA variable (known type, known concrete, known value), and then can easily look them up and rely on them.
And at PHI functions?

Merge what we know

But how to do it safely?

Use a lattice for each fact type
Known type lattice

Easy rule: only move up

\[
\begin{array}{c}
\text{Unknown (T)} \\
\text{Int} & \text{Str} & \text{Product} & \ldots \\
\bot
\end{array}
\]
Known type lattice

\[ \text{join}(\text{Int}, \text{Int}) \rightarrow \text{Int} \]
Known type lattice

\[ \text{join}(\text{Int, Str}) \rightarrow T \]
Known type lattice

$$\text{join}(\text{Int}, T) \rightarrow T$$

![Diagram of a lattice with nodes for Int, Str, Product, Unknown (T), and a bottom element.]}
Where do facts come from?

Sometimes we refer to a static value (constants, types)

Others come from the statistics
But...

The statistics only mean we tend to have a certain type or code object; they aren't a proof that we always will!
Enter guards

Thus, we insert guard instructions, which quickly check that the actual type etc. encountered is the one the statistics suggest is typical.
Deoptimization enables speculation
When a guard fails...

This is when we are forced to perform deoptimization

Fall back to the interpreted code that can handle all cases
Consequence

Must make sure that we preserve enough data so that we can fall back to the interpreter and have it continue
Example: dead code elimination

Some dead writes can't actually be removed, because they'll be needed if we are forced to deoptimize.
But generally, we win

Guards are far cheaper than the indirections they replace (and they "hoist" the checks)

Deoptimizations are rare
What about mixins?

Mixins change the types of objects "at a distance"

Force global deoptimization of the whole call stack
Some Optimizations
With a bunch of facts, and guards ensuring they are true, we can now proceed to transform the graph
Resolving method calls

Knowing the exact type lets us resolve method calls directly.

Saves a hash lookup in the method cache.
Avoiding multi-dispatch

Use the type facts to determine which multi candidate would be called, thus avoiding the overhead of the multi-dispatch cache
Specialization linking

Use argument types to identify which specialization of a callee should be used, avoiding argument type checks in the called code.
Inlining

For small callees, replace the call with the code in the callee, avoiding the overhead of creating and tearing down the call frame and arg passing.
Aside: uninlining

In order that we can inline, we also have to be able to undo it in deoptimization. This is "uninlining". A bit tricky, but we manage it.
Unchecked attribute accesses

Just read the memory location holding an attribute, rather than having to do a lookup by name (also applies to the value slot of a Scalar!)
Checks to constants

Type checks already answered by the established facts can be turned into constants. Same with "is it a container", "is it concrete", etc.
Constant conditional removal

These "new constants" may resolve some conditionals, allowing for removal of the check and branch instructions.
Let's see how the chars method was before optimization...
checkarity  liti16(1), liti16(1)
param_rp_o  r1(2), liti16(0)
hlize       r8(2), r1(2)
set         r1(3), r8(2)
decont       r8(3), r1(3)
wwal       r9(2), liti16(1), liti16(35) (P6opaque: Str)
istype   r10(1), r8(3), r9(2)
assertparamcheck  r10(1)
decont     r9(3), r1(3)
isconcrete  r10(2), r9(3)
assertparamcheck r10(2)
decont     r9(4), r1(3)
set         r0(2), r9(4)
param_sn    r2(2)
wwal       r4(2), liti16(1), liti16(35) (P6opaque: Str)
getattr_s  r5(1), r0(2), r4(2), lits($!value), liti16(0)
chars      r6(1), r5(1)
p6box_i    r4(3), r6(1)
wwal       r7(2), liti16(1), liti16(37) (P6opaque: Int)
decont     r9(5), r4(3)
istype   r6(2), r9(5), r7(2)
unless_i   r6(2), BB(12)
isconcrete  r10(3), r9(5)
if_i        r10(3), BB(15)
wwal       r8(4), liti16(1), liti16(21) (P6opaque: Nil)
istype   r6(3), r9(5), r8(4)
if_i        r6(3), BB(15)
wwal       r8(5), liti16(4), liti16(8) (not deserialized)
prepargs  callsite(0x7f0b7089da40, 2 arg, 2 pos, nonflattening, interned)
arg_o       liti16(0), r4(3)
arg_o       liti16(1), r7(2)
invoke_v    r8(5)
return_o    r4(3)
Argument handling, type and definedness checks

The work

Return value type check, including letting Nil pass by
Now here's the chars method after specialization and optimizations...
sp_getarg_o r8(2), liti16(0)
set r1(3), r8(2)
set r9(3), r1(3)
const_i64_16 r10(2), liti16(1)
set r9(4), r1(3)
set r0(2), r9(4)
sp_p6oget_s r5(1), r0(2), liti16(8)
chars r6(1), r5(1)
p6box_i r4(3), r6(1)
wval r7(2), liti16(1), liti16(37)
set r9(5), r4(3)
return_o r4(3)
One basic block, so all the possible invokish things have been devirtualized
All type checks removed
And, yes, a bunch of (cheap) set instructions that we'd like to get rid of in the future (mostly from overzealous deopt safety)
Producing Machine Code
No time for details, but as a next step, we can then compile this into x64 machine code, eliminating the overhead of interpretation.

(See video of brrt's TPCiA talk)
Specialization
Entry (and
Reentry)
So, how do we transition from slow-path interpreted code into specialized code?
Entry on invoke

See if the callsite and argument types match any specialization ("guard tree")

Use that which matches
On Stack Replacement

At the end of a loop body, check if there's an optimized version of the loop code; replace the running code "on stack" with it if there is
Reentry

What if a hot loop deopts one time in a hundred or so?

OSR can put us back into the optimized version again later
Future Plans
Box/unbox elimination, native reference elimination

To avoid allocating temporary box and reference objects, thus saving work immediately and causing less GC overhead
Escape analysis

Work out when an allocation doesn't escape a call, and replace it with a "stack" allocation rather than a "heap" (GC) allocation
More precise deopt handling

Current approach is safe, but decidedly coarse; it can't account for effects of guards that were added, but in the end weren't used.
More aggression on inlines

We don't yet propagate facts into the inlines; we could get further improvements to the code if we were able to do so
More tooling

Today, you can set the MVM_SPESH_LOG=a_file environment variable and read the (giant) output; a nicer tool would be good
That's all, folks!
Questions?