Optimizing Rakudo Perl 6

Jonathan Worthington
OH HAI
Rakudo
What is Rakudo?

Unlike with Perl 5, Perl 6 refers to just the language, not any one implementation of it.

Rakudo is a Perl 6 implementation.

Supports a wide range of language features.

Actively developed by many contributors (242 commits, 10 committers in October).
How Rakudo runs programs
How Rakudo runs programs

You give Rakudo your program

Your Program
How Rakudo runs programs

It parses it and builds an AST and a world

- **Your Program**
  - **AST** (Executable stuff)
  - **World** (Declarative stuff)
How Rakudo runs programs

The AST is turned into a VM-specific tree

Your Program

AST
(Executable stuff)

World
(Declarative stuff)

Lower Level AST
How Rakudo runs programs

Which finally becomes code for the VM
How Rakudo runs programs

We do a few fixups to the world...

Your Program

AST
(Executable stuff)

Lower Level AST

VM Code

World
(Declarative stuff)

Fixups
How Rakudo runs programs

...and we’re ready to run!

Your Program

AST
(Executable stuff)

World
(Declarative stuff)

Lower Level AST

VM Code

Run Program

Fixups
Rakudo Architecture

Rakudo
- Grammar
- Actions
- World
- Meta Objects
- CORE Setting

NQP (Bootstrapped)
- Grammar
- Actions
- World
- CORE Setting

Uses

Legend:
- NQP
- Perl 6
- VM Specific Code
- VM
Rakudo Architecture

Rakudo
- Grammar
- Actions
- World
- Meta Objects
- CORE Setting

NQP (Bootstrapped)
- Grammar
- Actions
- World
- Meta Objects
- CORE Setting

VM Abstraction
- PAST
- 6model
- nqp::ops

VM Specific
- Parrot
- CLR Backend
- LuaJit Backend

Legend:
- NQP
- Perl 6
- VM Specific Code
- VM
Where to optimize?

Optimize the Rakudo compiler

Optimize the setting (built ins)

Optimize NQP (which in turn helps Rakudo)

Optimize the lower level bits

Optimize the programs we’re compiling
Which are we doing?

All of them! 😊

The first half of the this talk will focus on ways we optimize the compiler stack and the various built-ins

The second half will focus on the Rakudo optimizer, which produces better code from the input programs
Profile, don’t guess!

Optimizations need to get targeted in order to really make a difference

Making something 2 times faster when it accounts for 0.5% of program runtime is not going to be very effective

Profilers tell us where we’re spending time
VM-Level Profiling
What is it?

Profiling that lets us understand the low level parts of our implementation

May also involve profiling the VM itself
  (we do this in the case of Parrot)

May involve profiling the layer on top of it
  (this is the case for the CLR backend)
What it can tell us

Often, we find out a lot about the cost of primitive operations...

Signature binding
Object allocation
Lexical lookups
Invocation
Time spent doing GC
Example

We take our slowest running specetest and run it through the Visual Studio C profiler.

Just from observation, we know that it spends a lot of time compiling the file.

Once we get to running it, it’s all over within 2 seconds on modern hardware.
Example

The output shows we spend 23% of the time in register allocation

This is decidedly abnormal
We can further drill down to see where time is being spent.
Example

While this does point to inefficiencies in the register allocator, it also suggests we should be **generating better code**

A quick look reveals that a couple of places generate code with an enormous number of registers used within a single block

Big pain point ➔ worth spending time on
Another story

Profiling compilation of the setting revealed that a huge amount of time (>20%) was taken by the GC to scan the system stack.

This uncovered a very inefficient algorithm for pointer analysis that had terrible CPU cache characteristics.

A 50ish line patch got a 20% improvement.
Benefits and limits

We can find out what lower level operations are taking up time.

However, at some level everything is made up of allocations, dispatches, etc.

We need to profile at a higher level in order to understand what those allocations and dispatches actually are.
Perl 6 and NQP Profiling
What is it?

Enables us to find out what NQP and Perl 6 level routines we are spending time in.

Parrot provides a sub-level profiler, which produces output that can be viewed using KCacheGrind.

Can sometimes provide a very different view of where time is being spent.
Example

This program was found to run hideously slowly for some reason

```perl
my @a = 'AA'..'ZZ';
my @b = 1..100;
.say for @a X~ @b;
```

Let’s take a look at it under the high level language profiler...
Drilling down through the results, we discover that an awful lot of time is spent calling the say method (about 35%)
Drilling down, say spends most of its time calling $*OUT.print(...)

```
sub say($) {
    my $args := pir::perl_current_args_rpa__P();
    $OUT.print(nop::shift($args).gist) while $args;
    $OUT.print("\n");
    67600.call(s) to '00000000099ABB60:DYNAMIC' (src\gen\CORE.setting)
}
```
Looking at the code, we see that every call to IO.print assumes it is dealing with a list of possible things.

```
method print(IO:D: *@list) {
  !$PIO.print(nqp::unbox_s(@list.shift.Str))
  while @list.gimme(1);
  Bool::True
}
```

That’s a lot of work when we just have a single string to output!
Things are improved greatly by adding another multi candidate for the Str case

```perl
proto method print(|$) { * }
multi method print(IO:D: Str:D $value) {
    !$PIO.print(nqp::unbox_s($value));
    Bool::True
}
multi method print(IO:D: *@list) {
    !$PIO.print(nqp::unbox_s(@list.shift.Str))
    while @list.gimme(1);
    Bool::True
}
```
Continuing on, we end up in the iterator implementation, and spot something odd.

We call two variants of the `not` operator – one more expensive than the other.
Continuing on, we end up in the iterator implementation, and spot something odd.

We call two variants of the not operator – one more expensive than the other.

Fix by initializing $end to False!
Example

Going deeper, we find a real hot spot in the implementation of the X op - one line that accounts for over 50% of runtime
Example

Cross with an arity-2 op should be easy!

```perl
my $rop = METAOP_REDUCE($op);
# ...
gather {
    while $i >= 0 {
        if @l[$i].gimme(1) {
            @v[$i] = @l[$i].shift;
            if $i >= $n { my @x = @v; take $rop(|@x); }    
        } else {
            $i = $i + 1;
            @l[$i] = (@lol[$i].flat,).list;
        }
    } else { $i = $i - 1; }
}
```

This is overkill for the common case.
Cross with an arity-2 op should be easy!

```perl
my $rop = @lol.elems == 2 ?? $op !! METAOP_REDUCE($op);
# ...
gather {
  while $i >= 0 {
    if @l[$i].gimme(1) {
      @v[$i] = @l[$i].shift;
      if $i >= $n { my @x = @v; take $rop(|@x); } else {
        $i = $i + 1;
        @l[$i] = (@lol[$i].flat,).list;
      } else {
        $i = $i - 1; }
    } else { $i = $i - 1; }
  }
}
```

This call took 30%; now it takes just 3%
Example

This was not a made up example; rather, it was a walk through of a process that resulted in commits to Rakudo

Just from these changes, the example program now ran in half the time

There’s plenty more improvements waiting to be discovered and implemented
Profiler win!

The profiler shows time spent across...

User code (Perl 6)
Built-ins in the setting (Perl 6)
The compiler implementation (NQP)

We can drill down between them (even seeing where the compiler calls a BEGIN block written in Perl 6!)
What is an optimizer?
Where does an optimizer fit in?

The optimization phase comes after we have fully built the AST and the world.
Where does an optimizer fit in?

It considers both, and then tries to improve them (mostly, it does changes “in place”)

Your Program

[Diagram]

AST

World

Optimizer

Better AST

Better World
Overall approach

An optimizer does everything in two steps

Analysis
What optimizations can I perform here? Is it really safe to do so?

Transformation
Given the analysis says “yes”, actually do the optimization
What’s the hard part?

The transformations tend to be relatively straightforward

All of the hard work takes place in the analysis phase

Doing a transformation where it’s not safe results in an “improved” program that is faster...and wronger!
The Rakudo Perl 6 Optimizer
Optimizations we do today

So far, the optimizer can do...

- Inlining simple, declaration-free blocks
- Compile-time sub call binding checks
- Inlining of simple subs
- Compile-time multi-dispatch resolution
- Inlining of compile-time resolved multi candidates

It can also detect some cases where code could never possibly work – and alert you at compile time
For this example, we’ll consider a short program with a tight loop; the programmer gave us a little type information too

```plaintext
my int $i = 0;
while $i < 10000000 {
    $i = $i + 1;
}
say $i;
```
Without optimization (1)

store_lex "$i", 0
loop1019_test:
  find_lex $I1013, "$i"
  perl6_box_int $P101, $I1013
  nqp_get_sc_object $P102, "1320268783", 10
  $P1012 = "&infix:<<>"($P101, $P102)
chain_end_15:
  unless $P1012, loop1019_done
.const 'Sub' $P1015 = "11_1320268784.057"
capture_lex $P1015
  $P1015()
goto loop1019_test
loop1019_done:
  find_lex $I1020, "$i"
  perl6_box_int $P101, $I1020
  $P102 = "&say"($P101)
Without optimization (2)

```perl
.sub "_block1014" :anon
:subid("11_1320268784.057")
  .param pmc param_1017 :call_sig
  .lex "$\_", $P1016
  .lex "call_sig", param_1017
  bind_signature
  find_lex $I1018, "$i"
  perl6_box_int $P103, $I1018
  nqp_get_sc_object $P104,
  "1320268783.804", 11
  $I100 = "&infix:<+>"($P103, $P104)
  store_lex "$i", $I100
  perl6_box_int $P105, $I100
  .return ($P105)
.end
```

Unrequired boxing

Multi-dispatch to the + operator

Boxing again!
In Perl 6, every block is conceptually a new lexical scope and a closure

Analysis
Our block declares no lexical symbols, so it serves no operational purpose

Transformation
Flatten it into the enclosing scope
store_lex "$i", 0
loop1019_test:
  find_lex $I1013, "$i"
  perl6_box_int $P101, $I1013
  nqp_get_sc_object $P102, "1320268783.804", 10
  $P1012 = "&infix:<<>>>(P101, $P102)
chain_end_15:
  unless $P1012, loop1019_done
  find_lex $P103, "$_"
  set pres_topic_1, $P103
  find_lex $I1016, "$i"
  perl6_box_int $P104, $I1016
  nqp_get_sc_object $P105, "1320271436.881", 11
  $I100 = "&infix:++>>(P104, $P105)
  store_lex "$i", $I100
  perl6_box_int $P106, $I100
  store_lex "$_", pres_topic_1
  goto loop1019_test
loop1019_done:
  find_lex $I1020, "$i"
  perl6_box_int $P101, $I1020
  $P102 = "&say"($P101)
Operators and multiple dispatch

In Perl 6, all operators are multiple dispatch lexical subroutines

This means that operator overloading just means declaring extra multi candidates

Changes are lexically scoped – that is, your operator changes are not global, and thus do not affect unrelated code
Performance consequences

The Perl 6 multiple dispatch algorithm can be implemented efficiently; Rakudo does rather well here.

However, invocation overhead is still very high compared to just adding two numbers!

The good news: in a given scope we know all of the multi candidates that are possible.
Compile time resolution

Knowing all possible candidates
+
Knowing the types of all arguments
=
Can (sometimes) decide which candidate is going to be called at compile time

\(\smile\)
Compile time resolution

At compile time we know that...

\[ i + 1 \]

Will make a call &infix<+>\( (\text{int}, \text{int}) \)

Can we safely decide which multi candidate will be called based upon this information?
Compile time resolution

We cannot go for a simple match – we have incomplete information in some cases.

Just because we statically have Any does not mean we won’t have types that pick a narrower candidate at runtime.

However, always safe to pick from natives group or the one immediately above it.
Inlining

Deciding which multi candidate to invoke helps a bit – but the decision making is actually dominated by the invocation.

However, we have another option: some very simple subs can be inlined.

This means their bodies are just copied to the place where the call would be.
After inlining operator multis

store_lex "$i", 0
loop1019_test:
  find_lex $I1014, "$i"
  islt $I100, $I1014, 10000000
  perl6_booleanize $P101, $I100
  perl6_decontainerize_return_value $P102, $P101
  unless $P1012, loop1019_done
  find_lex $P103, "$_"
  set pres_topic_1, $P103
  find_lex $I1015, "$i"
  add $I100, $I1015, 1
  store_lex "$i", $I100
  perl6_box_int $P103, $I100
  store_lex "$_", pres_topic_1
  goto loop1019_test
loop1019_done:
  find_lex $I1020, "$i"
  perl6_box_int $P101, $I1020
  $P102 = "&say"($P101)

Here, the < operator has been inlined

Here, the + operator has been inlined

Additionally, much boxing is now gone!
The optimizations result in code that is much more “to the point”, and that isn’t paying invocation overhead all the time.

Compared to the original program, this optimized version runs 23 times faster!

The code still isn’t all that great – we can do somewhat better yet.
That could never work!

While trying to resolve some dispatches at compile time, the optimizer may also be able to prove that it will never work

```perl
sub greet($name, $greeting) {
    say "$greeting, $name";
}
greet("Elena");

===SORRY===
CHECK FAILED:
Calling 'greet' will never work with argument types (str)
(line 4)
   Expected: :(Any $name, Any $greeting)
Future Optimizer
Work
Currently, the optimizer does not analyse how variables are used in a program.

Knowing when variables are read and/or written would allow for a range of optimizations and detection of program errors at compile time.

This is the “next big task” for the optimizer.
Methods calls are late bound, so we don’t tend to really know what to inline

However, we can make a good guess, then include both an inline and a guard type check, which falls back to a normal dispatch

Best when the call is in a hot loop, but the guard check can be moved outside of it
Type inference

Many variables keep the same type they start out with for their entire lifetime.

May be able to infer this initial type, and then try to “prove” that it is preserved throughout the variable’s lifetime.

A way to make untyped programs faster.
Looking Ahead
The Rakudo of today tends to be faster – and is sometimes considerably faster – than the Rakudo of a year ago.

Performance is one of the biggest adoption blockers, and is a big focus for us.

Much more work to come – stay tuned, or come and join in the fun! 😊
all<Ďakujem Danke>
Questions?

Blog: http://6guts.wordpress.com/
Twitter: jnthonwrthngtn
Email: jnthon@jnthon.net