Taking Rakudo Forward: What I'm Hacking On

Jonathan Worthington
My previous talk: Perl 6 from a user's perspective
This talk: Perl 6 from an implementer's perspective
Taking Rakudo Forward: What I'm Hacking On or
This talk:
A peek inside my brain
Taking Rakudo Forward: What I'm Hacking On

Perl 6 stuff

Bad puns

"I want a steak"

I/O

"Она красива!"

"I want a beer"
Taking Rakudo Forward: What I'm Hacking On

Perl 6 stuff

Bad puns

"I want a steak"

"Она красива!"

I/O

"I want a beer"

Image Attribution: WikiPedia
Rakudo Development Philosophy
Start off by achieving wide feature coverage but low feature "depth"
<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>So Crap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not so bad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Awesomeness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey nice!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better than beer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Taking Rakudo Forward: What I'm Hacking On

<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>So Crap</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Meh</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Not so bad</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Implementation</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Awesomeness</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Hey nice!</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
<tr>
<td>Better than beer</td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
<td><img src="image" alt="Rating" /></td>
</tr>
</tbody>
</table>

Before we called it Rakudo
Taking Rakudo Forward: What I'm Hacking On

<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>So Crap</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meh</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not so bad</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Awesomeness</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey nice!</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better than beer</td>
<td>Hack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>OO</td>
<td>Regexes</td>
<td>Built-ins</td>
</tr>
<tr>
<td>----------------------</td>
<td>----</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>So Crap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not so bad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Awesomeness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey nice!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better than beer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>OO</td>
<td>Regexes</td>
<td>Built-ins</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>So Crap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not so bad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Awesomeness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey nice!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better than beer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hack hack hack hack
<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>So Crap</td>
<td><strong>So Crap</strong></td>
<td><strong>So Crap</strong></td>
<td><strong>So Crap</strong></td>
</tr>
<tr>
<td>Meh</td>
<td><strong>Meh</strong></td>
<td><strong>Meh</strong></td>
<td><strong>Meh</strong></td>
</tr>
<tr>
<td>Not so bad</td>
<td><strong>Not so bad</strong></td>
<td><strong>Not so bad</strong></td>
<td><strong>Not so bad</strong></td>
</tr>
<tr>
<td>Implementation</td>
<td><strong>Implementation</strong></td>
<td><strong>Implementation</strong></td>
<td><strong>Implementation</strong></td>
</tr>
<tr>
<td>Awesomeness</td>
<td><strong>Awesomeness</strong></td>
<td><strong>Awesomeness</strong></td>
<td><strong>Awesomeness</strong></td>
</tr>
<tr>
<td>Hey nice!</td>
<td><strong>Hey nice!</strong></td>
<td><strong>Hey nice!</strong></td>
<td><strong>Hey nice!</strong></td>
</tr>
<tr>
<td>Better than beer</td>
<td><strong>Better than beer</strong></td>
<td><strong>Better than beer</strong></td>
<td><strong>Better than beer</strong></td>
</tr>
</tbody>
</table>

OMG new grammar engine!
## Taking Rakudo Forward: What I'm Hacking On

<table>
<thead>
<tr>
<th>Feature</th>
<th>OO</th>
<th>Regexes</th>
<th>Built-ins</th>
</tr>
</thead>
<tbody>
<tr>
<td>So Crap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meh</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not so bad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation Awesomeness</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hey nice!</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Better than beer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mmm....beer and hacking!
Try to get something usable into user's hands earlier rather than later
Users = Community = Feedback
Taking Rakudo Forward: What I'm Hacking On

Users = Community = Feedback
Rakudo *

Useful, usable release aimed at early adopters

Lots of nice features 😊

Various issues 😞
Taking Rakudo Forward: What I'm Hacking On

Rakudo * - a nice view point on the journey
Perl 6.0.0

Rakudo * - a nice view point on the journey

Mu
Focus of today's talk:
The work I'm doing to help us complete the next big part of the climb
Introducing Meta-models: A Story
Once upon a time, I wrote a class.

class Lolcat is Cat {
  has $.caption;
  has $!lol-factor;
  method lol() {
    say($!lol-factor < 0 ?? 'wtf' !!
          $!lol-factor < 42 ?? 'lol' !!
                        'rofl'));
  }
}
I thought my work was done, and I could go for a beer.
But then my class started asking me questions...
class Lolcat is Cat {
    has $.caption;
    has $!lol-factor;
    method lol() {
        say($!lol-factor < 0 ?? 'wtf' !!
            $!lol-factor < 42 ?? 'lol' !!
                'rofl');
    }
}
What does it mean to have methods?

class Lolcat is Cat {
    has $.caption;
    has $!lol-factor;
    method lol() {
        say($!lol-factor < 0 ?? 'wtf' !!
            $!lol-factor < 42 ?? 'lol' !!
            'rofl');
    }
}
What does it mean to inherit?

class Lolcat is Cat {
    has $.caption;
    has !$lol-factor;
    method lol() {
        say($!lol-factor < 0 ?? 'wtf' !!
            $!lol-factor < 42 ?? 'lol' !! 'rofl');
    }
}
class Lolcat is Cat {
    has $.caption;
    has $!lol-factor;
    method lol() {
        say($!lol-factor < 0  ?? 'wtf'  !!
            $!lol-factor < 42  ?? 'lol'  !!
                'rofl');
    }
}

Do other classes all behave like me?
class Lolcat is Cat {
    has $.caption;
    has $!lol-factor;
    method lol() {
        say($!lol-factor < 0   ?? 'wtf' !!
             $!lol-factor < 42  ?? 'lol' !!
             'rofl');
    }
}
Classes in Perl 6 are just one type of package.

We also have grammars and roles.
... token package_declarator: class {
    :my $*PKGDECL := 'class';
    <sym> <package_def>
}

token package_declarator: grammar {
    :my $*PKGDECL := 'grammar';
    <sym> <package_def>
}

token package_declarator: role {
    :my $*PKGDECL := 'role';
    <sym> <package_def>
}

STD.pm...
token package_declarator: class {
    :my $*PKGDECL := 'class';
    <sym> <package_def>
}
token package_declarator: grammar {
    :my $*PKGDECL := 'grammar';
    <sym> <package_def>
}
token package_declarator: role {
    :my $*PKGDECL := 'role';
    <sym> <package_def>
}
All have methods, attributes, semantics for inheritance and composition, etc.

Many more commonalities than differences.
Could bake the details deep in the implementation.

Not hackable, not extensible...and thus not Perl 6-like.
Idea!

Define an API and implement it for each type of package.
Taking Rakudo Forward: What I'm Hacking On

**OO API**

Make the API actually be a set of methods on an object

Different type of package = different type of object

Tweak an existing package type by subclassing
Implement the object model in terms of objects.

Extend the object model in terms of objects.
Meta-object

An object that specifies how some other object works
Meta-object Protocol

The set of methods that we implement in a meta-object
::LolCat := ClassHOW.new_type(name => 'LolCat');

LolCat.^add_parent(Cat);

LolCat.^add_attribute(Attribute.new(
    name => '$!caption', has_accessor => True
));
LolCat.^add_attribute(Attribute.new(
    name => '$!lol-factor'
));

LolCat.^add_method('lol', method () {
    ...
});

LolCat.^compose();
The 6model Project
Today's object implementation in Rakudo builds a layer on top of the Parrot built-in object model.
Allowed us to get to something that works well enough quickly

but

We've hit limits of this approach
Taking Rakudo Forward: What I'm Hacking On

- Semantic gap hurts
- Hard to hack on or change
- Hard to reason about
- Tricky to port to other VMs
- Performance issues
- No easy path to implement type-driven optimizations
- No easy path to implement representation polymorphism
Let's look at this with fresh eyes.
Small object model core designed with serving Perl 6's needs at its heart
Taking Rakudo Forward: What I'm Hacking On

Learn from...

- Moose
- SMOP
- Smalltalk
- Current Model
- Academic Work
- Static OO Languages
- CLOS

Small object model core designed with serving Perl 6's needs at its heart
So what do I want out of this process?
Small Low-Level Core

Write the rest in Perl 6 (or a subset of it)
Tension between "low-level and fast at runtime" and "high level, hackable, extensible and maintainable"
"What are the core primitives to try and get really fast?"
Method dispatch in the common, optimizable cases

Attribute access

Type checks

Object instantiation
Don't need to worry quite so much over...
Type construction (happens at compile time)

Role composition

Introspection

The uncommon cases
Conclusions

Primitives will be:
- Method dispatch
- Attribute storage and lookup
- Object allocation

Build everything else (classes, inheritance, roles, introspection) out of them
Taking Rakudo Forward: What I'm Hacking On

Representation
Polymorphism
How do we represent an object in memory?

How do we store attributes?

How do we box/unbox native types?
These are all issues related to representation.

Perl 6 offers representation polymorphism, to allow classes to choose (or let the class user choose) a representation strategy.
Possible to leave a class open to being instantiated with different representations

```
class Color::RGB is repr(*) {
    has uint8 $.red;
    has uint8 $.green;
    has uint8 $.blue;
}
```

“I want to store lots of these in an array” => bit-packed representation

“Just one, fast access” => typical word-aligned representation
Conclusions

We shall have two core APIs.

**HOW API** = control over dispatch, declarations, introspection

**REPR API** = control over object allocation, attribute storage (and if applicable, GC interaction)
Gradual Typing
How much type information is there in this code?

```pascal
sub get_cat_from_rescue_center($type, $owner) {
    my Cat $rescued = cat_search($type);
    $rescued.owner = $owner;
    return $rescued;
}

my $kitteh = get_cat_from_rescue_center('tabby', 'Anna');
```
How much type information is there in this code?

```perl
sub get_cat_from_rescue_center($type, $owner) {
    my Cat $rescued = cat_search($type);
    $rescued.owner = $owner;
    return $rescued;
}

my $kitteh = get_cat_from_rescue_center( 'tabby', 'Anna');
```
How much type information is there in this code?

```
sub get_cat_from_rescue_center($type, $owner) {
    my Cat $rescued = cat_search($type);
    $rescued.owner = $owner;
    return $rescued;
}

my $kitteh = get_cat_from_rescue_center('tabby', 'Anna');
```
How much type information is there in this code?

```
sub get_cat_from_rescue_center($type, $owner) {
    my Cat $rescued = cat_search($type);
    $rescued.owner = $owner;
    return $rescued;
}

my $kitteh = get_cat_from_rescue_center(
    'tabby', 'Anna');
```

Cat       Any       Mu
No extra type information provided

The compiler lets you choose how much type information to provide

and

tries to give you more benefits as you give it more information to work with
A key place we can take advantage of type information is to optimize method dispatches.

Normally, we look up methods in a hash table.

Faster is to index into a v-table.
class Shape {
    has $.name;
    method area() { ... }
}
class Square is Shape {
    method area($side) { $side ** 2 }
}
class Shape {
    has $.name;
    method area() { ... }
}

class Square is Shape {
    method area($side) { $side ** 2 }
}

V-table for Shape

...  
Copied v-table from Any
...  
area
name
class Shape {
    has $.name;
    method area() { ... }
}
class Square is Shape {
    method area($side) { $side ** 2 }
}
Conclusions

Compiling method dispatches to v-table lookups means we need the meta-objects built and available at compile time.

Single unified compile time and runtime MOP...

...and a place to hang a v-table
The Model
So Far
<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Taking Rakudo Forward: What I'm Hacking On

<table>
<thead>
<tr>
<th><strong>Object</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Meta-object</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>new_type</td>
</tr>
<tr>
<td>add_method</td>
</tr>
<tr>
<td>add_parent</td>
</tr>
<tr>
<td>add_attribute</td>
</tr>
<tr>
<td>compose</td>
</tr>
<tr>
<td>methods</td>
</tr>
<tr>
<td>parents</td>
</tr>
<tr>
<td>attributes</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Taking Rakudo Forward: What I'm Hacking On

Object

...
Taking Rakudo Forward: What I'm Hacking On

Object
- Meta-object
- REPR
  - Under control of REPR

Meta-object
- new_type
- add_method
- add_parent
- add_attribute
- compose
- methods
- parents
- attributes
  ...

REPR
- type_object_for
- instance_of
- get_attr
- bind_attr
  ...

?
Taking Rakudo Forward: What I'm Hacking On

Objects are getting a little fat...

<table>
<thead>
<tr>
<th>Object</th>
<th>Meta-object</th>
<th>REPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>REPR</td>
<td>WHAT</td>
</tr>
<tr>
<td></td>
<td>V-table</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Under control of REPR</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meta-object</th>
<th>new_type</th>
<th>add_method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add_parent</td>
<td>add_attribute</td>
</tr>
<tr>
<td></td>
<td>compose</td>
<td>methods</td>
</tr>
<tr>
<td></td>
<td>parents</td>
<td>attributes</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>REPR</th>
<th>type_object_for instance_of</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>get_attr</td>
</tr>
<tr>
<td></td>
<td>bind_attr</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
Taking Rakudo Forward: What I'm Hacking On

- **Object**
  - S-Table
  - Under control of REPR

- **S-Table**
  - Meta-object
  - REPR
  - WHAT
  - V-table

- **Meta-object**
  - new_type
  - add_method
  - add_parent
  - add_attribute
  - compose
  - methods
  - parents
  - attributes
  - ...

- **REPR**
  - type_object_for
  - instance_of
  - get_attr
  - bind_attr
  - ...

Diagram showing the relationships between different components and data structures.
Bounded Serialization
We build the meta-objects and S-tables at compile time

but

We need them at runtime

Serialize (freeze) them at the end of the compile, and deserialize (thaw) them at program startup
One of the main reasons that Rakudo’s **startup time** is so bad today is that we have to construct all of the built-in types at startup.

Want to just serialize them all once and be able to quickly deserialize them each startup.
Taking Rakudo Forward: What I'm Hacking On

TRICKY PROBLEM IS TRICKY.

😊
class Food {
  has $.hot;
  has $.vegetarian;
}

class Pizza {
  has $.diameter;
  has @.toppings;
}

use Food;
use Food;
class Pizza {
    has $.diameter;
    has @.toppings;
}

class Food {
    has $.hot;
    has $.vegetarian;
}
Taking Rakudo Forward: What I'm Hacking On

**Pizza.pm**

```perl
use Food;
class Pizza {
    has $.diameter;
    has @.toppings;
}
```

**Food.pm**

```perl
class Food {
    has $.hot;
    has $.vegetarian;
}
```

**Diagram:**

- **Pizza.pm**
  - `use Food;`
  - `class Pizza {`
    - `has $.diameter;`
    - `has @.toppings;`
  - `}`

- **Food.pm**
  - `class Food {`
    - `has $.hot;`
    - `has $.vegetarian;`
  - `}`
Taking Rakudo Forward: What I'm Hacking On

use Food;
class Pizza {
  has $.hot;
  has $.vegetarian;
  has $.diameter;
  has @.toppings;
}

class Food {
  has $.hot;
  has $.vegetarian;
}
use Food;
class Pizza {
  has $.hot;
  has $.diameter;
  has @.toppings;
}

class Food {
  has $.hot;
  has $.vegetarian;
}
Give every object and every S-table a pointer to a Serialization Context.
Object

S-Table

SC

Under control of REPR

S-Table

Meta-object

REPR

WHAT

V-table

SC

Meta-object

new_type

add_method

add_parent

add_attribute

compose

methods

parents

attributes

...

REPR

type_object_for instance_of

get_attr

bind_attr

...

WHAT
When serializing, we visit objects added to our SC.

If it’s **not in an SC**, serialize it and visit its children.

If it **already has an SC**, serialize a fix-up (reference) so we can link it.
Taking Rakudo Forward: What I'm Hacking On

VM

Portability
Today Rakudo only runs on and targets the Parrot VM.

Just as Perl 5 supports many platforms, in Perl 6 we want to support many runtimes.

“Perl 6 should be available everywhere.”
Small meta-model core

= 

Small amount to port

Design is quite naturally portable. \o/
Current Status
The core of the model has been implemented.

Working representation polymorphism.

First, working cut of an implementation of classes.
Today, the core so far is implemented on:

Parrot
.Net CLR
JVM
In the future there will likely be more

but

don’t want to spread limited development resources too thin.
What now?
Taking Rakudo Forward: What I'm Hacking On

**NQP**

Finish filling out ClassHOW

Push it into the bootstrapped NQP on Parrot

Implement serialization contexts and serialization

Update NQP to use them
Rakudo

Get the grammar and actions to run on the updated NQP

Re-write the meta-objects to work with the new object model

Use serialization contexts

Debug until it works 😊
.Net/JVM

Get Class HOW to run

Get NQP tests to pass

Bootstrapped, self-hosting NQP

Get Rakudo to run there
Merci beaucoup!
Questions?