

Rakudo Evolved

Speed, feedback and hackability

Jonathan Worthington

OH HAI

I'm Jonathan.

Or jnthn.

I like...



Living in European countries whose names start with an “s”

I like...



Hiking in the mountains

I like...



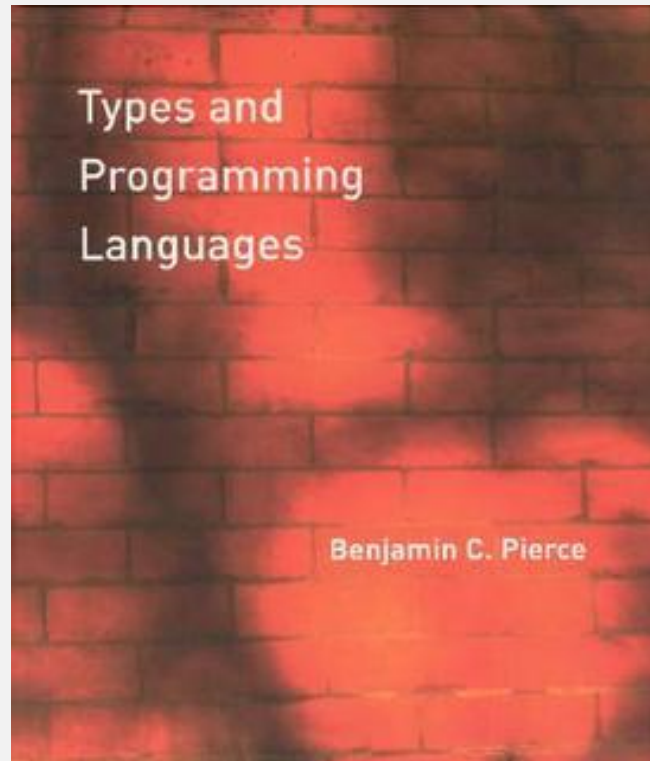
Cold weather

I like...



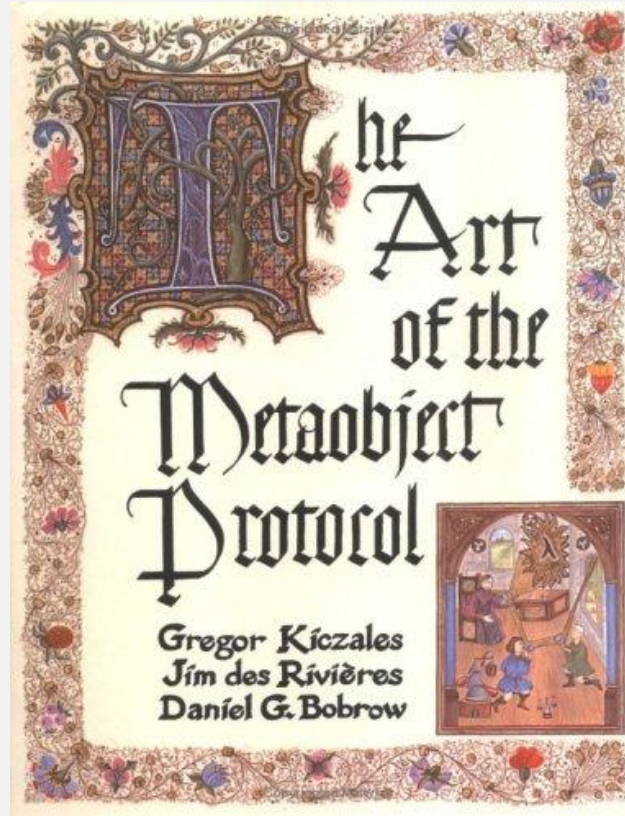
Good beer

I like...



Type systems

I like...



Meta-object protocols

I like...



Perl 6

Around a year ago...

Rakudo *

First distribution release of Rakudo Perl 6

Not just the compiler, but modules and a Perl 6 book included too

Aimed at getting more people to try Perl 6, thus giving us more feedback, contributors, modules, etc.

Rakudo * - The Good

Lots of features

**Development helped clarify many aspects
of the Perl 6 specification**

(Go to pmichaud's talk on lists for an example)

**Drew more people into the Perl 6
community and got us more feedback**

Rakudo * - The Bad

Rather Slow

Quite memory hungry

**Weak in areas of language extensibility,
such as meta-programming**

Very constrained BEGIN time

Finding A Way Forward

Some Ways Being Fast Is Hard

A literal implementation of some features is slow (e.g. operators are multi-subs)

Can be very dynamic, which limits how much we can statically know

Even when we do have plenty of type information, user-defined meta-objects may be implementing those types

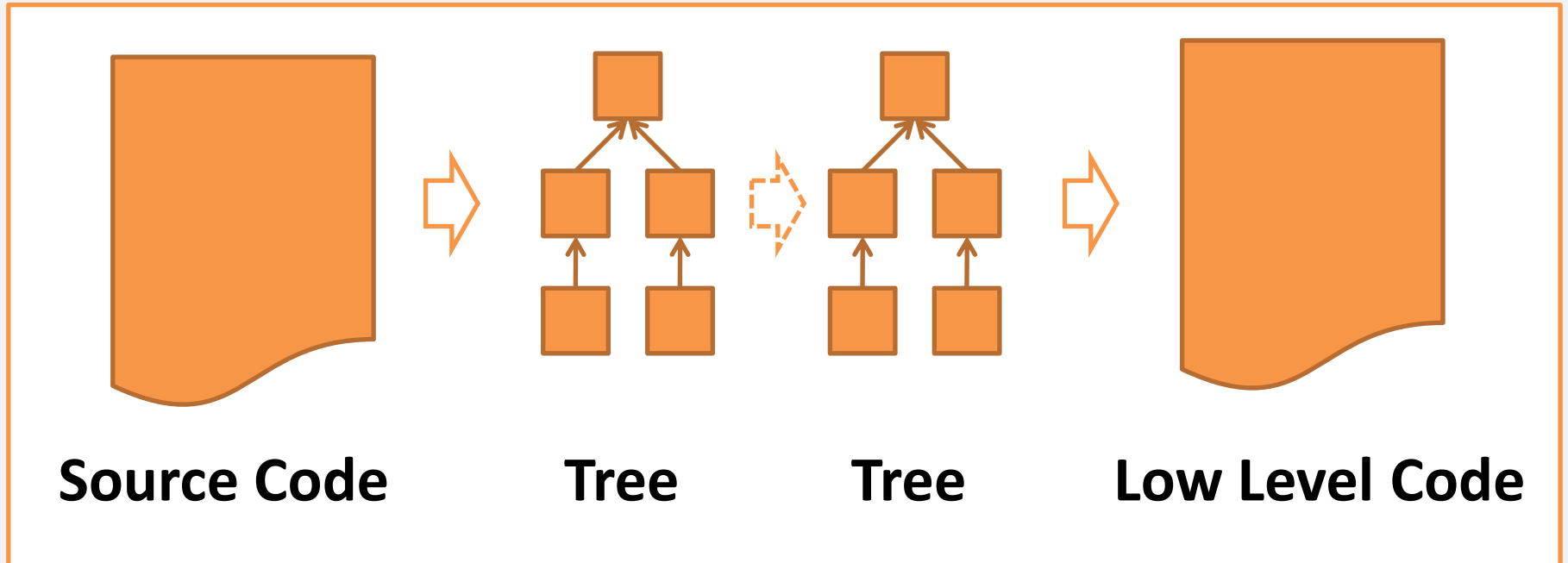
Some Ways Being Fast Is Possible

**Always know what multis are in scope, so
can often decide and inline things at
compile time**

**Sometimes a guess is enough, provided we
can handle having guessed wrong too**

**Require/enable user-defined meta-objects
to provide what the optimizer needs**

Typical Compilation Process



A clean, well understood approach – but for Perl 6, we'd hit its limits

Looking Deeper

A compiler always needs to build a **model of the things that it is compiling, from classes all the way down to individual variables**

Used for semantic analysis and optimization

In Perl 6, the programmer has a huge degree of control over the elements we're building the model out of

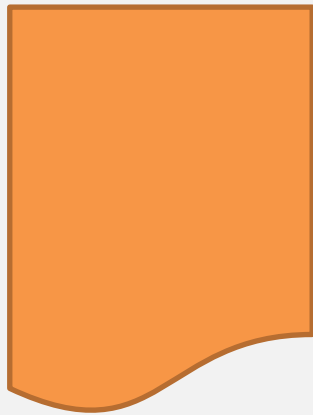
Blurring Boundaries

**Needed to (mostly) let go of the boundary
between runtime and compile time**

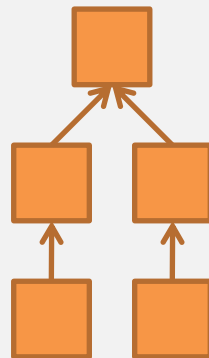
**The model we'd build at compile time
would thus contain the **very same objects**
that will be used at runtime**

Those objects may well be user defined

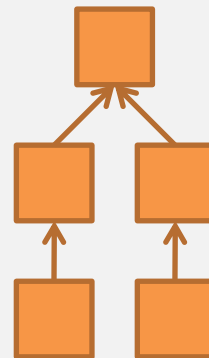
Something More Like...



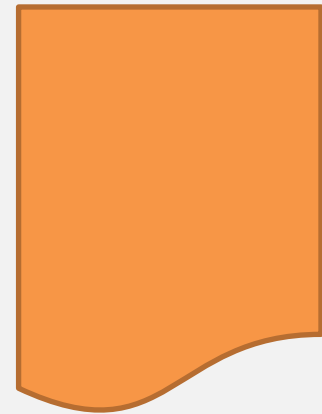
Source Code



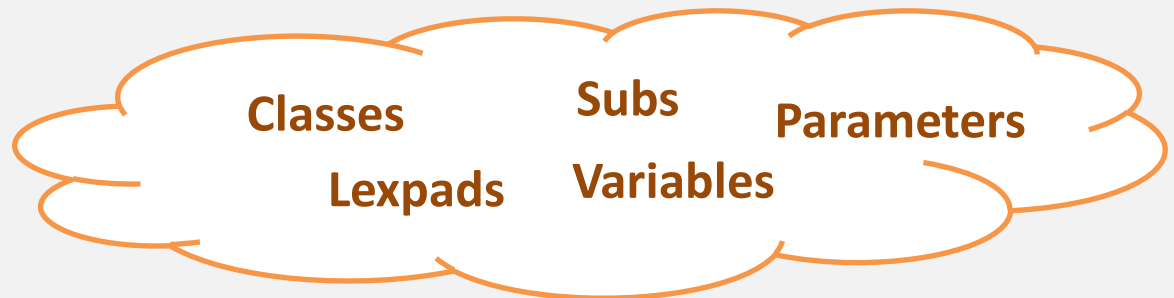
Tree
+
Model



Tree
+
Model



Low Level Code
+
Model



Consequences

Much complexity disappears from the compiler and into the model objects

Boundary between user-defined and built-in objects in the model removed

Semantic analysis and optimization have very rich and automatically accurate model of the program to work with

The Other Big Problem

Parrot's Object Model

Committed to too much → inflexible

Committed to too little → slow

No native type support

**Poor base for meta-programming, and
decidedly on the runtime side of the fence**

For Perl 6 We Need...

Meta-programming

(So developers can tweak and extend the language)

Gradual typing


(Use type information well - but don't require it)

Representation polymorphism

(There's more than one way to store it in memory)

Meta-objects available at compile time

(So that we can use them in analysis and optimization)



**Finding the
heart of OO
and types**

Easy Yet Deep Questions

What is an **object**?

What is a **type**?

How are the two **related**?

The Operational View

Definitions of OO seem to have certain **operational “hot-paths”** in common:

Object allocation

State access (e.g. attribute lookup/binding)

Some form of dispatch (methods, messages, generic function application...)

The Operational View

With types there's really only one key operation:

Type membership checking

(Slight complication: sometimes only the thing we're checking against knows the answer, sometimes only the type knows the answer)

OO Primitives

Really, objects just have...

Some state

(A chunk of memory we can stick stuff in)

Some semantics

(Described by a meta-object)

Membership of one or more types

(Either the object knows it, or the type knows it)

Things That Needn't Be Primitive

Inheritance

Roles and flattening composition

Parametric types

Refinement types

Hopefully, things we didn't invent yet 😊

6model

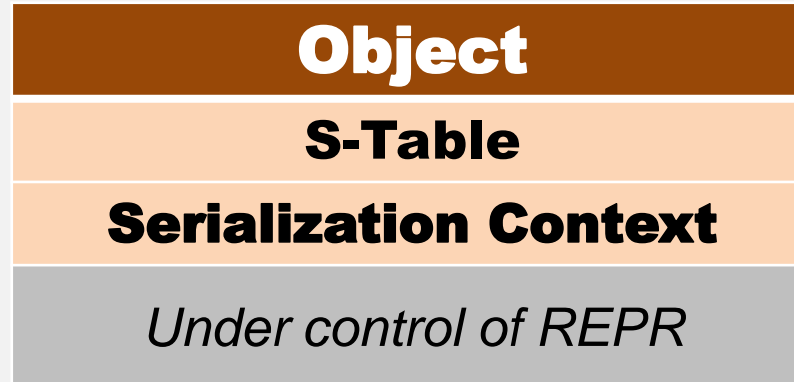
An Object Framework

Small core from which a rich set of meta-objects can be grown

Pays attention to how to make the common operational things fast

A little support for smudging the compile time – runtime boundary

An Object



Has a pointer to a shared table (“s-table”)

Has a pointer to a serialization context

6model stops caring beyond that point

An S-Table

S-Table
HOW (Meta-object)
REPR Function Table
REPR Data Store
<i>Various Caches</i>

Essentially, one of these per type (really, per pairing of meta-object and representation)

Meta-object is just another object – nothing more special than that

Representations

Low level – partly serve as the binding to the underlying virtual machine

REPRs have functions that may deal with...

Object memory allocation

Attribute storage

Boxing/unboxing to native types

GC integration

Storage aspects of type changes

Meta-objects

Objects that describe the semantics of other objects

How are method calls on the object dispatched?

What types is the object a member of?

And much, more more...

Growing a MOP

What's in the MOP?

Tend to have a meta-object for each sort of type in the language

ClassHOW

GrammarHOW

ParametricRoleHOW

ConcreteRoleHOW

SubsetHOW

EnumHOW

Factoring With Roles

**Many meta-objects have things in common,
and get most of their functionality by
composing roles**

**MethodContainer
MultipleInheritance
C3MRO
Trusting
Versioning**

User Defined Meta-objects

Tweak an existing one by subclassing

**Create a new one from scratch, and maybe
compose some of the provided roles so you
needn't implement so much**

**A combination of the two approaches can
sometimes be useful**

Archetypes

A way for a meta-object to describe the sorts of ways it can be used, and how it can produce other related types

nominal

nominalizable

inheritable

inheritalizable

generic

Meta-programming Example

Goal

Write a very basic implementation of
method modifiers

```
use MethodModifiers;
```

```
class C {  
    method m() is before { say "before"; }  
    method m() { say "in the method"; }  
    method m() is after { say "after"; }  
}
```

```
C.m;
```

Trait Handlers

To support the before and after traits, we add and export a couple of trait handlers

```
multi trait_mod:<is>(Method $m, :$before!) is export {  
  apply_modifier($m, 'before');  
}  
  
multi trait_mod:<is>(Method $m, :$after!) is export {  
  apply_modifier($m, 'after');  
}
```

Called at compile time, and before the method is added to the class

Attaching The Modifier

Applying a modifier just mixes in a role to the method to store the name of it

```
sub apply_modifier($m, $mod) {
  if $m.?modifier -> $cur_mod {
    die "'$m.name()' already has modifier $cur_mod"
  }
  $m does role {
    has $.modifier;
    method set_modifier($mod) {
      $!modifier := $mod;
    }
  };
  $m.set_modifier($mod);
}
```

Subclassing ClassHOW

For this example, we'll just subclass ClassHOW (default class implementation)

```
my class ClassWithMethodModifiers
  is Metamodel::ClassHOW is Mu {
  # ...
}
```

More flexible would have been to define it as a role, or at least provide it as one so it can be combined with other extensions

Storing Modified Methods

We override `add_method` and have it stash away any methods with modifiers

```
has %!modified_methods;  
  
method add_method(Mu $obj, $name, $meth) {  
  if $meth.?modifier -> $mod {  
    %!modified_methods{$name} // = [];  
    %!modified_methods{$name}.push($meth);  
  }  
  else {  
    callsame;  
  }  
}
```

Method Table Computation

Fiddle with method table computation

```
method method_table($obj) {
  my %meth_table = callsame;

  for %!modified_methods.kv -> $name, @mods {
    unless %meth_table.exists($name) {
      die "Modified '$name' has no normal method";
    }
    my %mod_map = @mods.classify: { .modifier };
    my $normal = %meth_table{$name};
    %meth_table{$name} = make_disp(%mod_map, $normal);
  }

  %meth_table
}
```

Method Table Computation

make_disp just creates a closure that
does the dispatching as needed

```
sub make_disp(%mod_map, $normal) {  
  -> $obj, *@pos, *%named {  
    for %mod_map<before>.list {  
      $obj.$_(|@pos, |%named);  
    }  
    my $res := $obj.$normal(|@pos, |%named);  
    for %mod_map<after>.list {  
      $obj.$_(|@pos, |%named);  
    }  
    $res;  
  }  
}
```

And Finally...

We export it as the default HOW to use for classes anywhere that uses the module

```
my module EXPORTHOW { }  
EXPORTHOW.WHO.<class> = ClassWithMethodModifiers;
```

Note that we neglected to handle inheritance and didn't fix up introspection to include the modifying methods, which a full implementation would require

Status

Rakudo Using 6model

Rakudo has been extensively refactored to use 6model

We now build meta-objects at compile time and use them at runtime

**Vastly better (though not yet complete)
BEGIN support and meta-programming**

Speed Improvements

Depends heavily on the benchmark

Startup time is about the same for now

Runtime improved (for example a Mandelbrot benchmark ran 4-5x faster)

This is just switching to 6model – we've barely started to really exploit it yet

Regressions?

Inevitably there will be some regressions in such a major refactor

Thankfully, we have a large test suite, and modules have tests too

Not as drastic as alpha → ng, which was essentially a re-write; this time we kept the grammar and actions for the most part

Portability

Perl 6 should run everywhere – not just on every platform, but on every VM

We've eliminated just about all of the PIR from the codebase in the process of this refactoring

Big win for our future porting efforts

Rakudo Roadmap

Serialization

At the moment, we retain the sequence of “events” that occurred during compilation and use them to re-construct the environment in pre-compiled code

In the next couple of months, we’ll switch to serializing the environment instead

Big startup win expected, fixes BEGIN

Optimizers

We'll also be starting work on optimizers in the next couple of months

Optimizer for Rakudo will aim to improve the performance of Perl 6 code

Also need an optimizer for NQP (which much of the compiler is written in)

Compile-time Multi Dispatch

We now have sufficient understanding of the multi-candidates in a given lexical scope at compile time to start making dispatch choices then

Need more tracking of type information within the program

Should give us a modest win

Multi-sub Inlining

Once we know what code we're calling, we can go a step further: inline it

Need cross-compilation unit inlining (or we can't inline operators from the setting)

This is where we get big wins – not to mention that it opens the door to further optimizations on the flattened code

Method inlining?

Difficult because method calls are late bound, and everything in Perl 6 is virtual

There's still hope! We can guess. 😊

Just need to re-check at runtime that the inline was indeed OK, and be able to “undo” it if not

Native Types et al

We've started work on these, though we really need the inlining for them to shine

Also need to implement packed arrays and compact structs

Then there's the other bits of S09...

And much more...

The goal remains to deliver a Perl 6 implementation that's stable, performant and that you'll love using

The last year has seen many exciting developments

The next few releases will bring a great deal more improvements

Thank You!

Questions?

Blog: <http://6guts.wordpress.com/>

Twitter: jnthnwrthngtn

Email: jnthn@jnthn.net