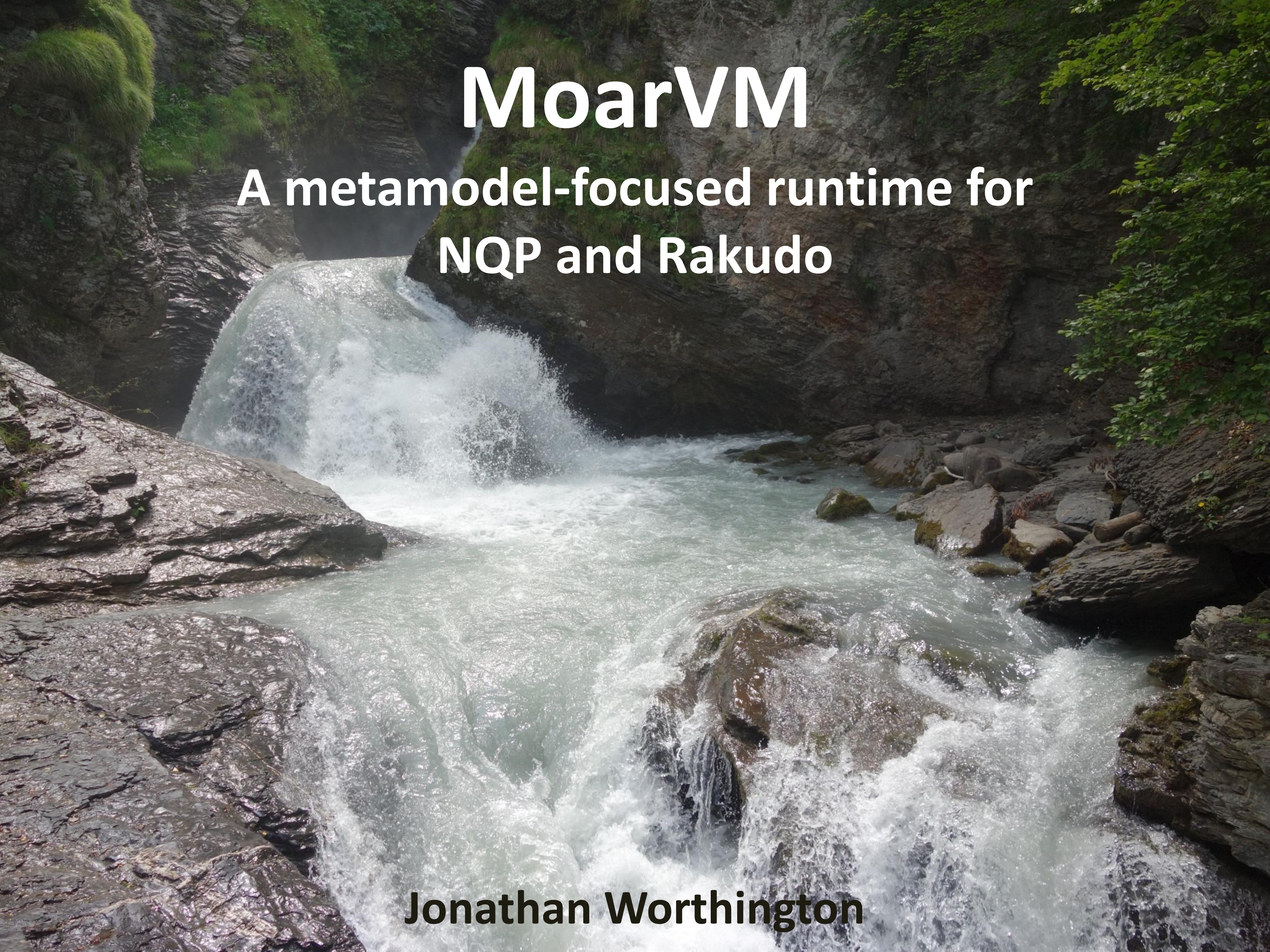


MoarVM

A metamodel-focused runtime for NQP and Rakudo

Jonathan Worthington

What does a VM typically do?

Execute instructions (possibly by interpreting, possibly by JIT compilation, often a combination)

Provide memory management (both allocation and deallocation, typically through garbage collection)

Offer a range of built-in data structures and instructions to operate on them (strings, arrays, objects, …)

Abstract away the details of the underlying OS and expose a common interface to IO, threading, etc.

Perl 6 and VMs

Original plan

Build the Parrot Virtual Machine in parallel with the Perl 6 language design, and then build a Perl 6 implementation that targets it

Reality today

Rakudo Perl 6, originally only targeting Parrot, now also runs on the JVM, with active work on other backends; additionally, the Niecza Perl 6 implementation targets the .NET CLR (Common Language Runtime)

Concerns

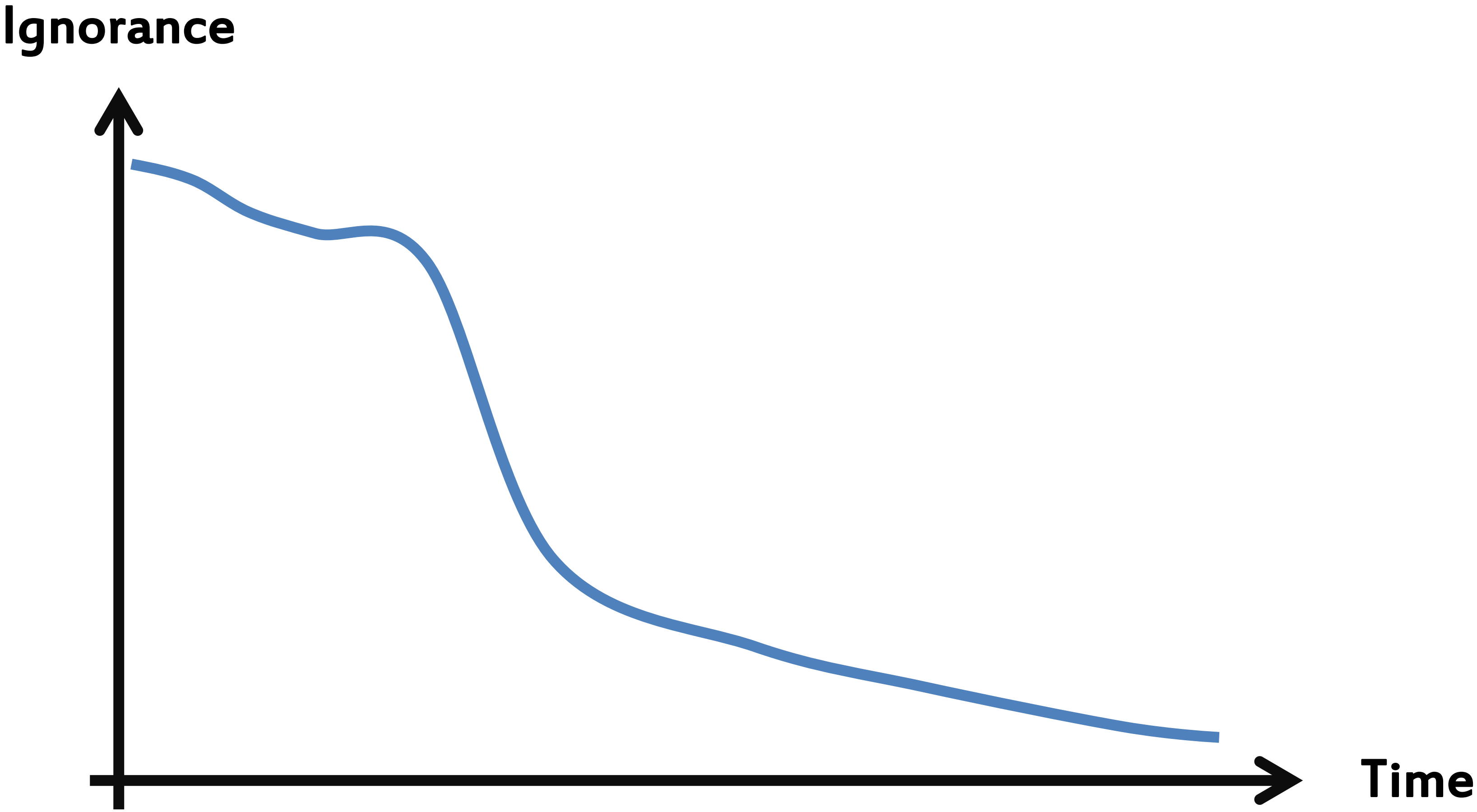
The Parrot project hasn't been as successful as hoped, due to a large number of factors

Performance has certainly been a problem, as is evolving a 10+ year old codebase that partially implements the visions of multiple architects over time

While running on the JVM and CLR is fine – or even desirable – for some potential Perl 6 users, others have reasons for not using these platforms ("Oracle are evil", "Microsoft are evil", "JVM startup is too slow"…)

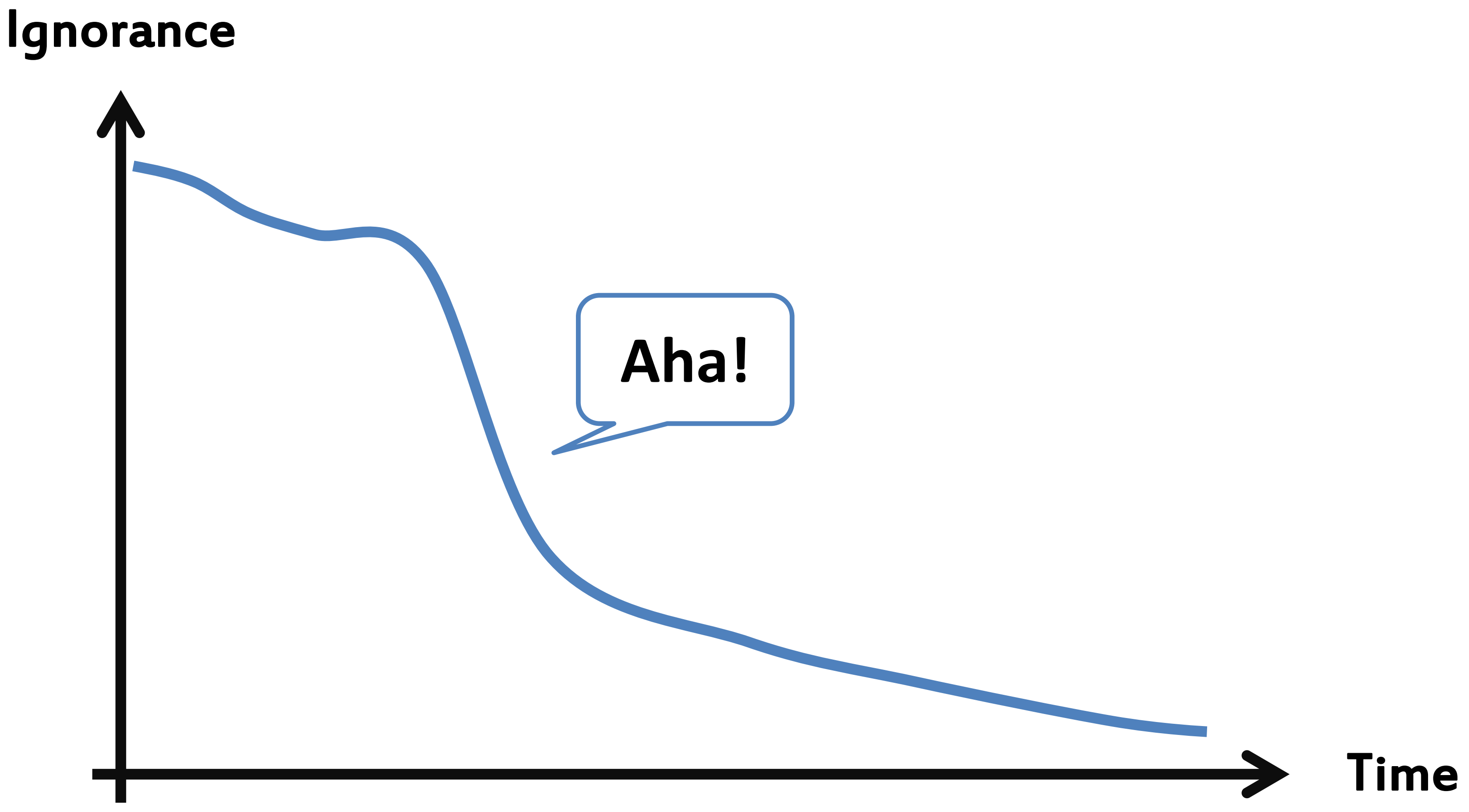
The ignorance curve

As time passes, ignorance of a domain decreases…



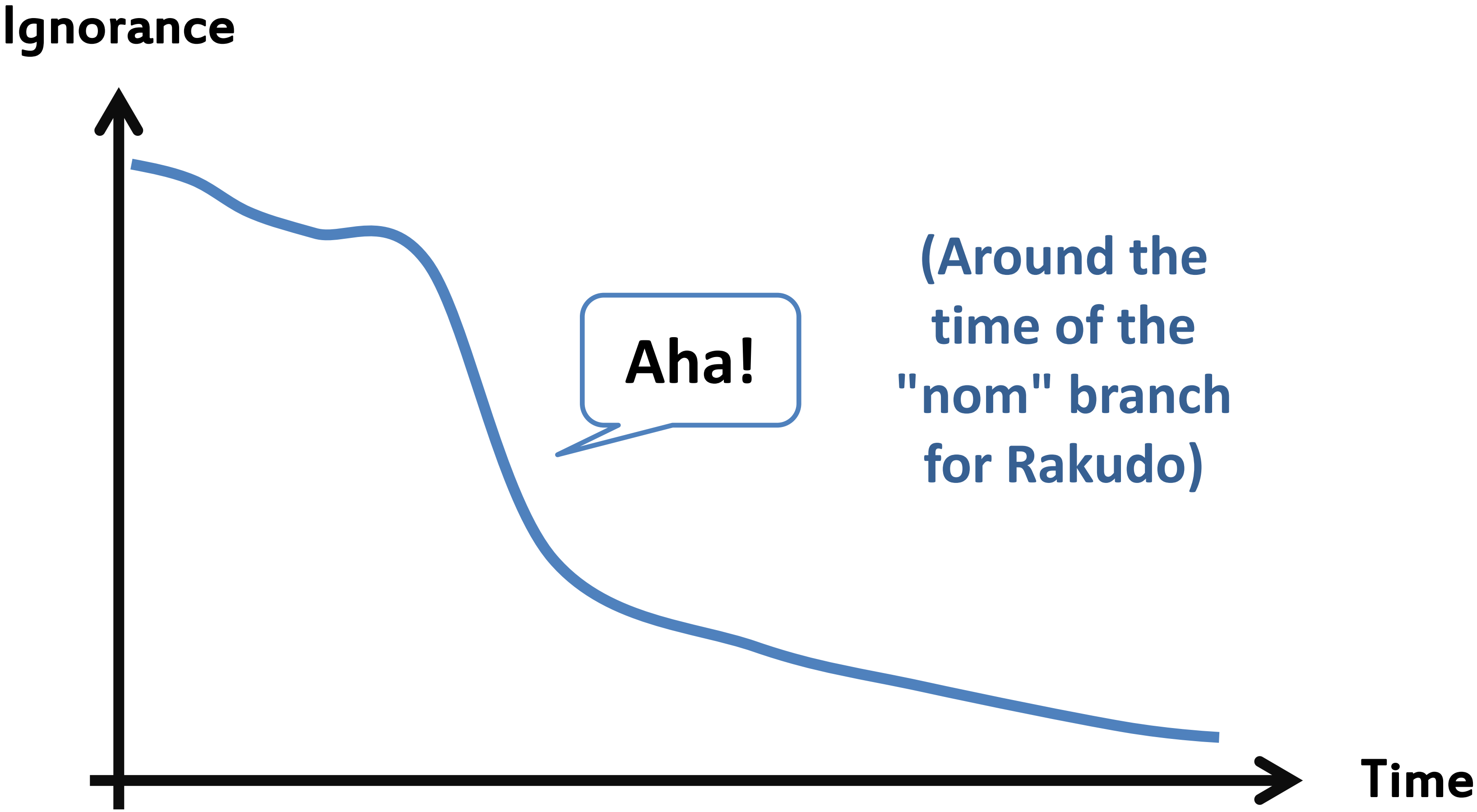
The ignorance curve

...and often there's an "aha!" moment



The ignorance curve

...and often there's an "aha!" moment



(Around the time of the "nom" branch for Rakudo)

We know what we need now

Parrot was built assuming Perl 6 would be like Perl 5 in some deep ways. It was as good a guess as could really be made, but implementing what Perl 6 worked out to be out of those assumptions felt a bit like this:

Example: namespaces

In the early days of Parrot development, much time was spent discussing how to implement namespaces

A single flat hash of names, name-mangled in some way…

No! Hierarchical namespaces, something hash-of-hash like…

In the end, it turned out that Perl 6 doesn't want a global namespace (because of separate compilation) and that stashes just hang off type objects representing packages

➔ Rakudo basically can't use Parrot namespaces

What if…

We take what we know now about what Perl 6 needs

and

Implement a VM that does precisely those things?

Ungoals

Run all the languages!

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

Have a textual assembly/intermediate language

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

~~Have a textual assembly/intermediate language~~

Don't waste time on this; compilers want to emit a tree

Ungoals

~~Run all the languages!~~
If we don't need a feature for Perl 6, we don't add it

~~Have a textual assembly/intermediate language~~
Don't waste time on this; compilers want to emit a tree

Publicly discuss every design call for weeks

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

~~Have a textual assembly/intermediate language~~

Don't waste time on this; compilers want to emit a tree

~~Publicly discuss every design call for weeks~~

Initial year of development in private; only go public when it's capable enough we can run some NQP on it

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

~~Have a textual assembly/intermediate language~~

Don't waste time on this; compilers want to emit a tree

~~Publicly discuss every design call for weeks~~

Initial year of development in private; only go public when it's capable enough we can run some NQP on it

Add threads later

Ungoals

~~Run all the languages!~~

If we don't need a feature for Perl 6, we don't add it

~~Have a textual assembly/intermediate language~~

Don't waste time on this; compilers want to emit a tree

~~Publicly discuss every design call for weeks~~

Initial year of development in private; only go public when it's capable enough we can run some NQP on it

~~Add threads later~~

Get threads and threaded GC in early, even if not perfect

Overall design

Use 6model, the object system designed for Rakudo Perl 6, for all of the object-like things

Generational garbage collection with parallel (but not concurrent) collection

Instruction set aligned with the nqp:: opcode set

Unicode support with NFG strings

Use 3rd-party libraries for things outside the core domain

6model

Provides primitives for building an object system

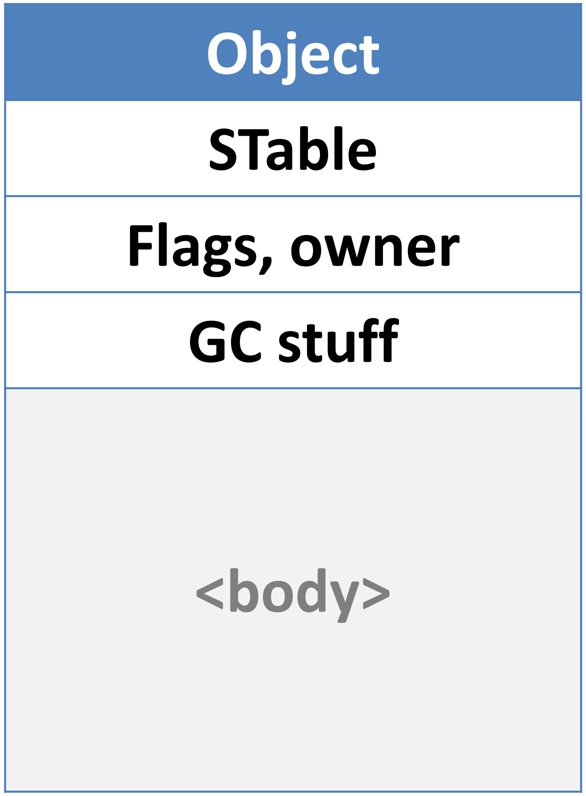
Every object in MoarVM is a 6model object
→ one object system for the whole VM

By "object" we mean…

The things you think of as objects
Arrays
Hashes
Boxed integers, floats, etc.
Threads, handles, …

Inside 6model

An object has a header…



Object

STable

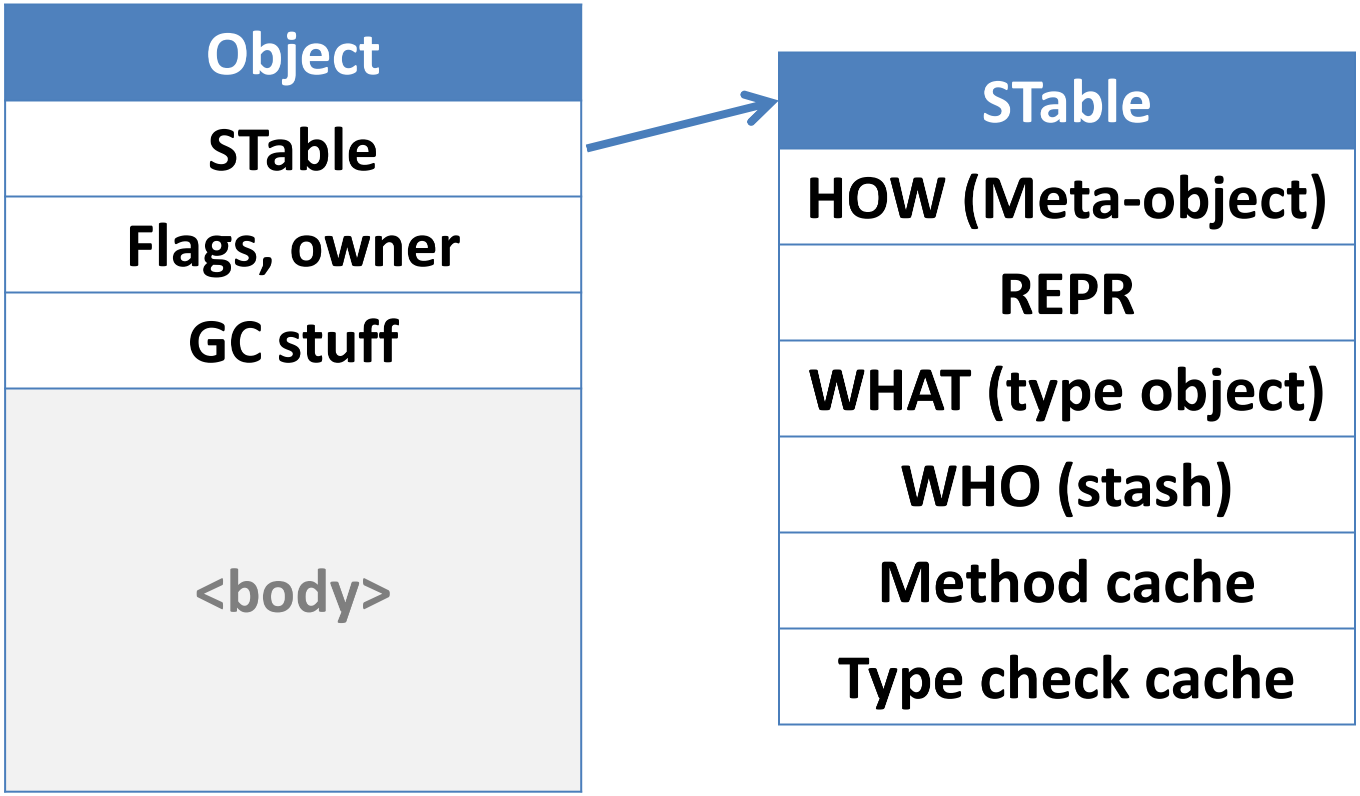
Flags, owner

GC stuff

<body>

Inside 6model

…which points to an STable (representing a type)…



Object

STable

Flags, owner

GC stuff

<body>

STable

HOW (Meta-object)

REPR

WHAT (type object)

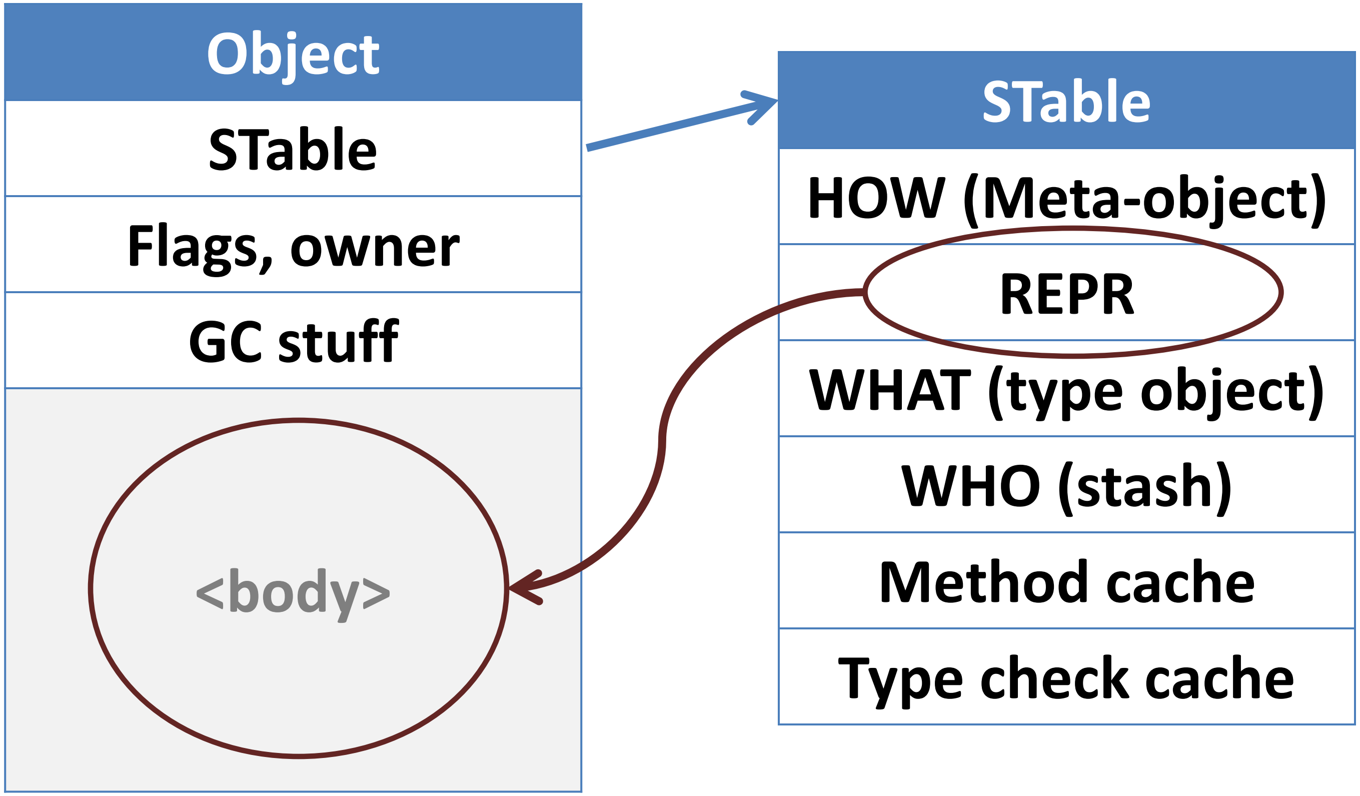
WHO (stash)

Method cache

Type check cache

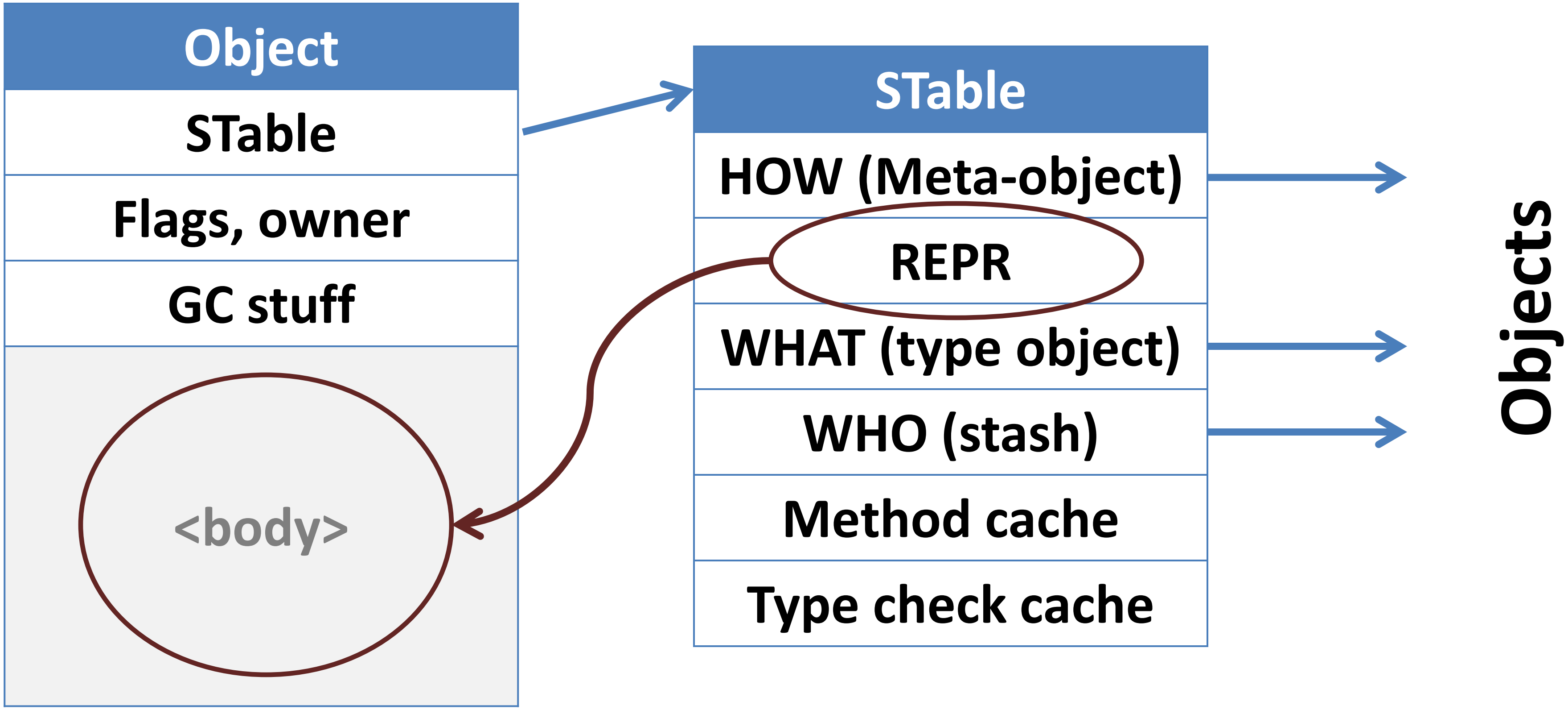
Inside 6model

…which has a representation that manages the body…



Inside 6model

…and points to some objects important to the type



Object

STable

Flags, owner

GC stuff

<body>

STable

HOW (Meta-object)

REPR

WHAT (type object)

WHO (stash)

Method cache

Type check cache

Objects

Representations

All about the use of memory by an object

REPR API has a common part (allocation, GC marking) along with several sub-protocols for different ways of using memory:

Attributes

Positional

Boxing

Associative

Representations are orthogonal to type (and thus dis-interested in method dispatch, type check, etc.) and also non-virtual (if you know the REPR, can inline stuff)

GC needs

Perl 6 produces a LOT of short-lived objects as it runs
→ allocation should be cheap
→ we'll need to run GC frequently
→ throwing away objects should be cheap

By contrast, other objects – such as meta-objects for declared classes – live for a very long time
→ don't want to examine them every collection
→ when we know they will live for a long time from initial allocation, want to use that knowledge

Would also like to use multiple threads

GC design

2 generations, known as nursery and gen2

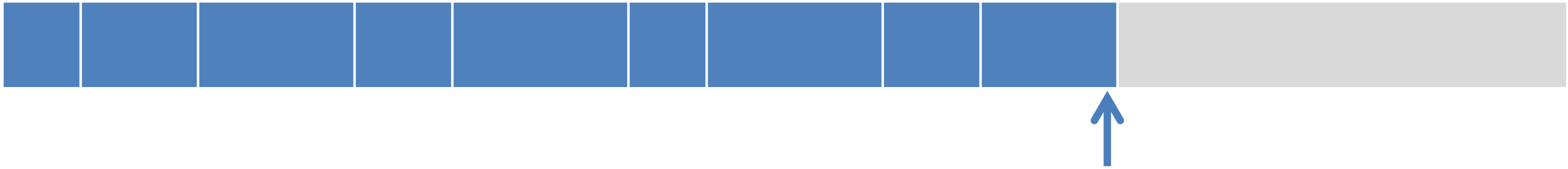
The nursery is where most objects are allocated (when we know we have long lived, allocate right in gen2)

That surviving 2 nursery collections are promoted

GC design: nursery

Semi-space copying collector

Allocate objects one after the other in a memory chunk



Next allocation here

When it fills, copy each living object into a new memory chunk, thus compacting them

Those that are dead are simply not copied

GC design: gen2

Most objects are stored in size-specific pools, which avoids fragmentation

Objects found to be dead during a full garbage collection are added to a free list, chained through the pool

Once in gen2, objects don't move (for now, at least; maybe some day we'll do compaction)

Objects too large to fit into any of the sized pools are allocated and managed separately

nqp::op aligned opcode set

In NQP and Rakudo, all operations that we can perform in a VM-independent way are captured in the nqp:: op set

Covers arithmetic, string manipulation, array and hash operations, object operations, I/O, and a few specialized things for the grammar engine

The MoarVM instruction set is largely derived from this
➔ well aligned with what we need

This alignment and more compact instruction code seems to lead to bytecode files 1/3 the size of on Parrot

Unicode support

MoarVM includes the Unicode Character Database so far as we need it for Perl 6

No external dependencies (like ICU)

Rather well compressed; even with all of this included, the full MoarVM executable weighs in at ~2.5MB

Support the various case change operations, character property lookups (also used for regex character classes), character name resolution…

NFG

NFC (Normalization Form C) will always collapse a codepoint followed by a combining codepoint into a single codepoint if one is available

o (U+006F) + ¨ (U+0308) → ö (U+00F6)

NFG (G = Grapheme) takes it a step further; if a single codepoint is not available, it makes one up (relying on being able to use negative integers to represent these)

This means we can treat even strings with combining characters as fixed width and get things right!

Tree → bytecode, no assembler

The MoarVM AST (commonly written "MAST") is a low-level tree representation of a program

We turn this directly into MoarVM bytecode, with various bits of validation along the way to catch common code generation mistakes

12 different node types

CompUnit	Frame	Op	SVal
IVal	NVal	Label	Local
Lexical	Call	Annotated	HandlerScope

Threadsafe, with a lock-free bias

Considered thread safety of the VM's data structures from the start, and watch for violations in code review

We use mutexes in some places

However, many places – especially anything on a hot path – uses atomic operations in place of locks

For example, frame reference counts are incremented and decremented using atomic operations

Scales way better than locks all over the place!

Use existing libraries

APR (but libuv soon)

For I/O and thread abstraction

uthash

For our hashes

libatomic_ops

For atomic operations

libtommath

For big integer operations

MoarVM status

The heart of the VM is in place:

Bytecode interpreter
Most of 6model
Generational, parallel GC
String and Unicode support
Basic thread support
Basic I/O support

Current branches are attacking NFG as well as migrating from using the APR to using libuv, which will enable provision of asynchronous I/O

NQP on MoarVM status

Along with MoarVM, we've been building an NQP cross-compiler, just as happened earlier on in the JVM port

Runs on Parrot, turns QAST into a MAST tree, then has the tree turned into MoarVM bytecode

By now it can cross-compile and run the majority of the NQP test suite, as well as many of the NQP libraries

On course to achieve a self-hosted NQP on MoarVM some time in September

In the next six months…

Get NQP bootstrapped on MoarVM

Get Rakudo running and passing spectests on MoarVM

Perl 5 interoperability (ask diakopter++ for more info)

Harden and exercise threading; add the primitives we need for Rakudo Promise, Channel, etc.

Asynchronous I/O support

And, of course, lots of bug hunting/fixing

Looking further…

Rakudo * distribution release on MoarVM

Full NFG support

6model-aware JIT compilation, including inlining and specializing code by type

Runloop meta-model, for providing a mechanism to build profilers, debuggers, etc.

No doubt, lots more performance work and bug fixes

Want to know moar?

IRC Channel:
#moarvm on freenode.org

Git repository:
https://github.com/MoarVM/MoarVM

Thank you!

Questions?

Blog: 6guts.wordpress.com
Twitter: @jnthnwrthngtn
Email: jnthn@jnthn.net