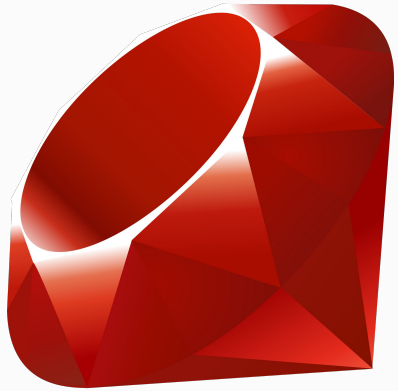
A History of Compiling Ruby

Chris Seaton

Senior Staff Engineer

Shopify

RubyConf 2021



What is a Ruby compiler?

- From Ruby code, to native machine code
- Can be via some intermediate system
- Can be ahead-of-time or just-in-time
- Requires some judgement and opinions to interpret, but I'm not trying to gate-keep

Did you know there have been at least...

25

...attempts to build a Ruby compiler?

Buy why?

- You can talk to anyone about compilers
- Compilers are really fun
- It's surprising how many there are... why has this happened?
- There's new interest in compilers for Ruby
- Lots of existing work to learn from - what's been tried before?
- We can trace compiler research through Ruby compilers
- Some of these projects were at risk of being lost to mists of time

What we're definitely not doing here...

- Talking about which is better or why
- Benchmarking or a competition

- Inevitably going to contain a few opinions

Compiler	Years active	Base VM	Stage	General approach	Frontend	Interpreter	Intermediate representations	Key authors
Hokstad	2008-present	Custom Ruby	AOT	Template compilation of an AST	Custom recursive descent and operator precedence parser	None	Enhanced AST	Hokstad
Hyperdrive	2019-2020	MRI	JIT	Tracing of YARV instructions then template compilation to Cranelift	Tracing YARV interpreter	Instrumented base interpreter	None	Matthews
IronRuby	2007-2011	Custom C#	JIT	Generation of CIL				Lam
JRuby	2006-present	Custom Java	JIT*	Generation of JVM bytecode	Parser to AST, to internal IR	Internal IR interpreter	CFG of linear RTL instructions	Nutter, Enebo, Sastry
LLRB	2017	MRI	JIT	Generation of LLVM IR				Kokubun
Ludicrous	2008-2009	MRI	JIT	Template compilation through DotGNU LibJIT				Brannan
MacRuby	2008-2013	MRI	AOT/JIT	Generation of LLVM IR				Sansonetti
MagLev	2008-2016	Custom Gemstone Smalltalk	JIT					McLain, Felgentreff
Natalie	2019-present	Custom C++	AOT	AST incrementally lowered to C++			Enhanced AST	Morgan
Ruby+OMR	2016-2017	MRI	JIT	Generation of J9 IR				Gaudet, Stoodley
RTL MJIT	2017	MRI	JIT	Generation of C				Makarov
Rubinius	2008-2016	Custom C++ and Ruby	JIT	Generation of LLVM IR	Parser to AST, to custom stack bytecode	Stack bytecode	None	Phoenix, Bussink, Shirai
Rhizome	2017	MRI, JRuby, Rubinius	JIT	Conventional speculative compiler with in-process assembler	Base bytecode or IR to custom bytecode	Stack bytecode	Graphical sea-of-nodes	Seaton
RubyComp	2004		AOT					Alexandersson
RubyX	2014-2020		AOT	Conventional compiler with in-process assembler	Parser to AST	None	Multiple IRs gradually removing abstraction and lowering from AST to linear	Rüger
Ruby.NET	2008	Custom C#		Generation of CIL				Kelly
Rucy	2021		AOT	Template compilation to eBPF				Uchio
Sorbet Compiler	2019-present	MRI	AOT	Generation of MRI LLVM IR 'C' extension	Parser to AST	None	Sorbet's typechecking IR	Tarjan, Petrashko, Froyd
TenderJIT	2021	MRI	JIT	Lazy Basic Block Versioning with in-process assembler	Template compiler of YARV bytecode	Base interpreter	None	Patterson
Topaz	2012-2014	Custom RPython and Ruby	JIT	Metatracing of a stack bytecode interpreter	Parser to AST	Stack bytecode interpreter		Gaynor, Felgentreff
TruffleRuby	2013-present	Custom Java and Ruby	JIT	Partial evaluation of self-specialising AST	Parser to AST	Self-specialising AST interpreter	Graphical sea-of-nodes	Seaton, Daloze, Menard, Chalupa, MacGregor
XRuby	2006-2008	Custom Java	AOT	Template compilation to Java bytecode	Parser to AST	None	None	Zhi
yarv2llvm	2008-2010	MRI	JIT	Generation of LLVM IR				Hideki
YARV MJIT	2018-present	MRI	JIT	Generation of C		Base interpreter		Kokubun
YJIT	2020-present	MRI	JIT	Lazy Basic-Block Versioning with in-process assembler	Template compiler of YARV bytecode	Base interpreter	None	Chevalier-Boisvert

Key Ruby compilers today

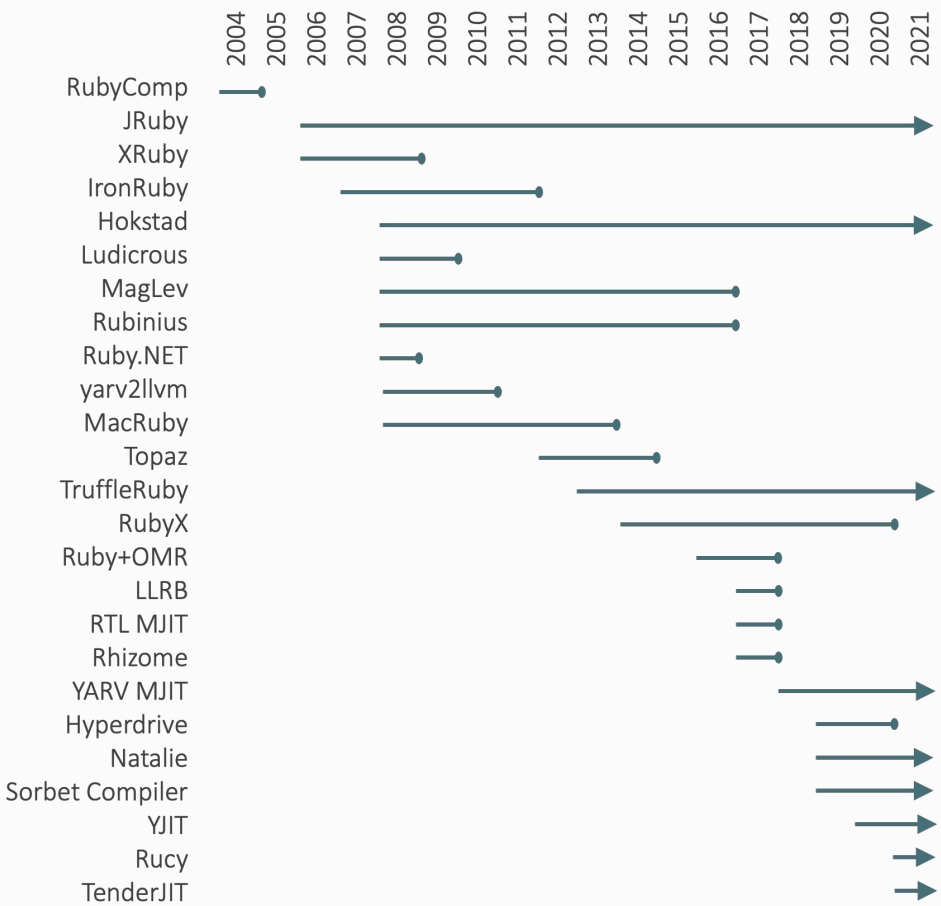
JRuby	Ruby on the JVM
TruffleRuby	Ruby on the GraalVM with a core in Ruby
YARV MJIT	MRI 3x3's JIT
YJIT	Shopify's new MRI JIT
Sorbet Compiler	Stripe's Ruby to C extension compiler

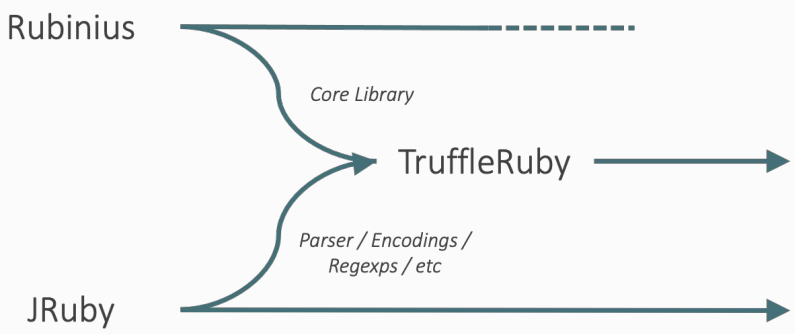
Key Ruby compilers that passed us by

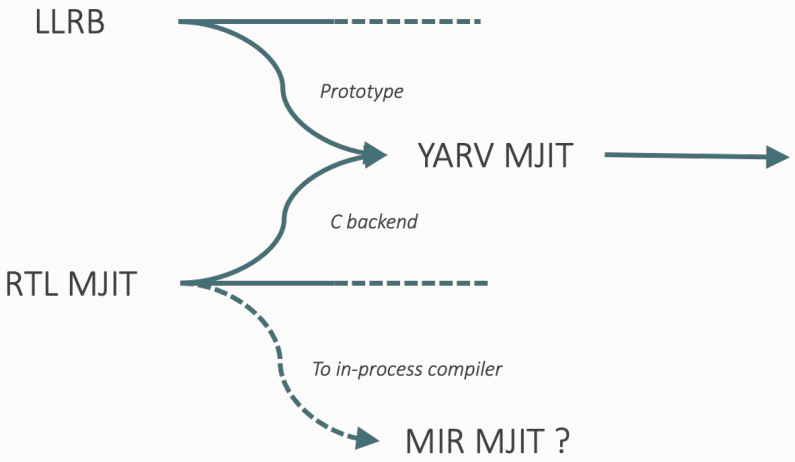
Rubinius	Ruby “in Ruby” with LLVM
Topaz	Ruby in RPython, like PyPy
IronRuby	Ruby in C#
Ruby+OMR	Ruby on J9

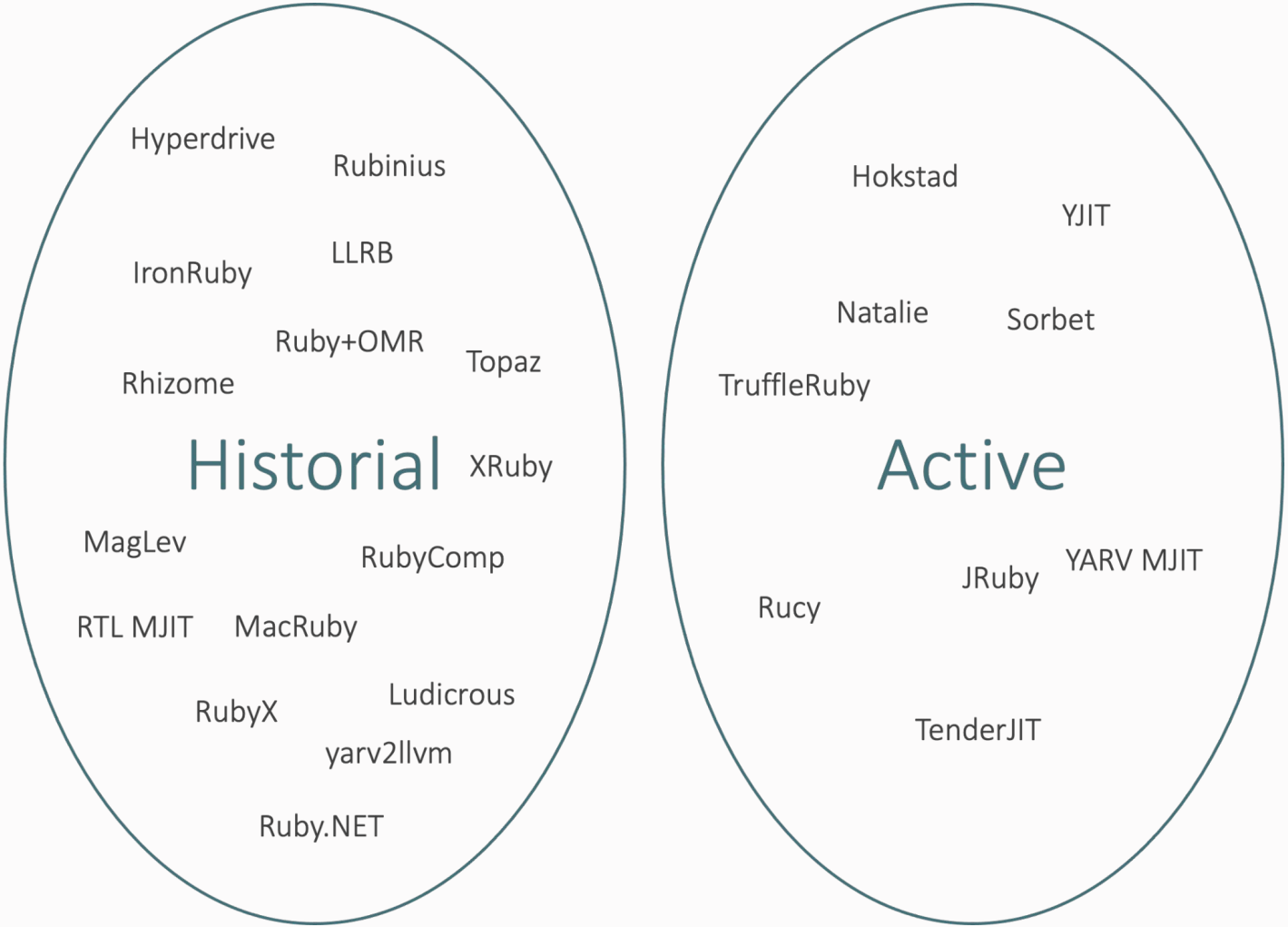
Fun niche compilers I'd recommend

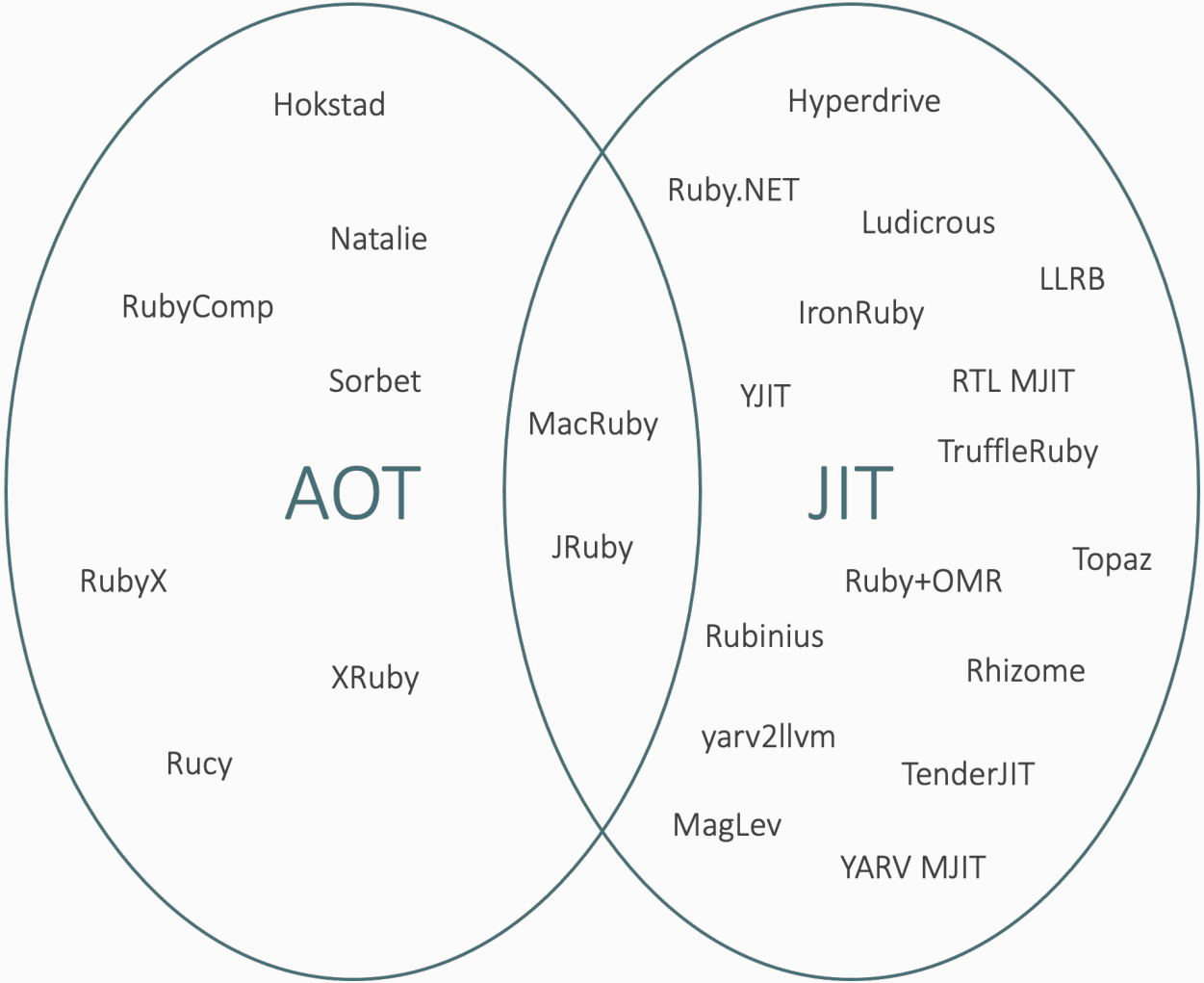
Hokstad	Ruby directly to ARM that's easy to understand
Hyperdrive	Ruby with tracing
Natalie	Ruby to C++ with great videos
RubyX	Ruby to ARM with a more advanced architecture

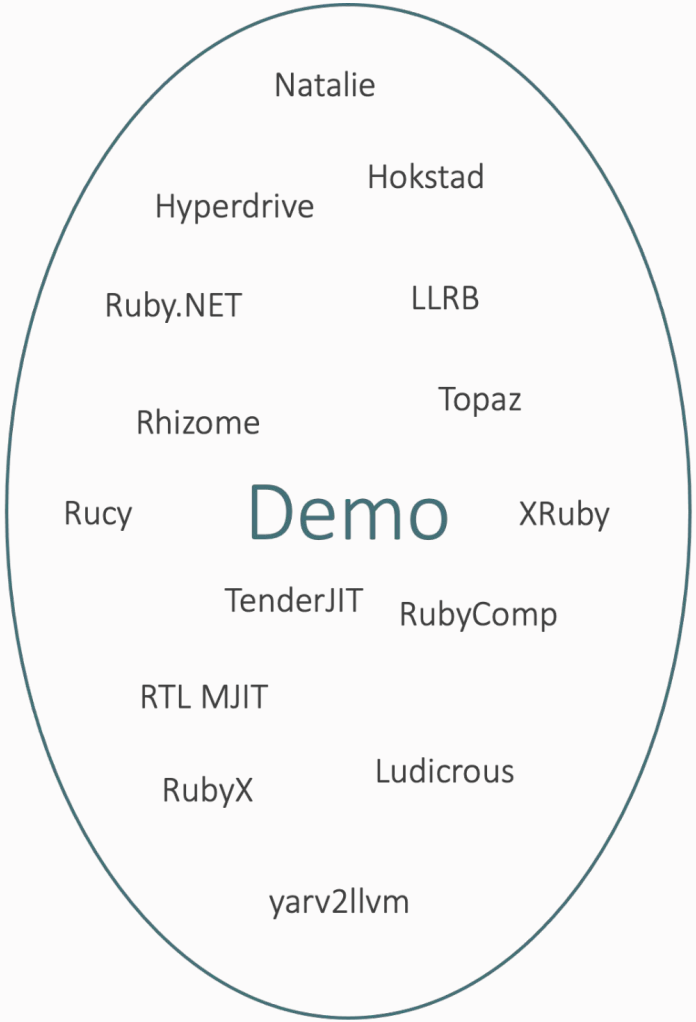

2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
RubyComp
JRuby
XRuby
IronRuby
Hokstad
Ludicrous
MagLev
Rubinius
Ruby.NET
yarv2llvm
MacRuby
Topaz
TruffleRuby
RubyX
Ruby+OMR
LLRB
RTL MJIT
Rhizome
YARV MJIT
Hyperdrive
Natalie
Sorbet Compiler
YJIT
Rucy
TenderJIT

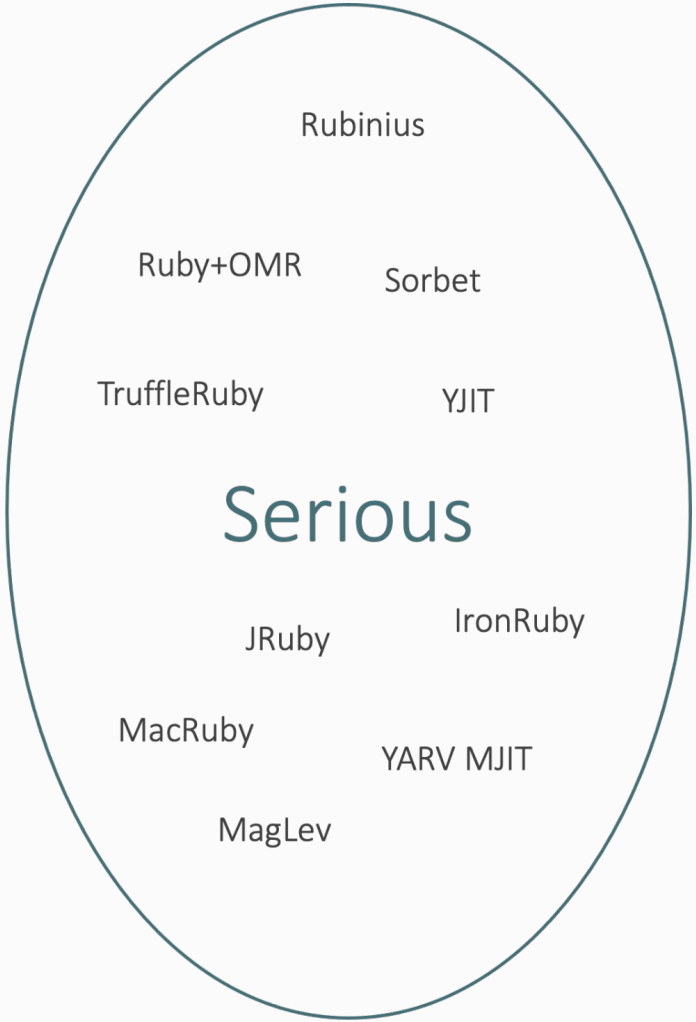

Rubinius
Core Library
TruffleRuby
Parser / Encodings /
Regexps / etc
JRuby

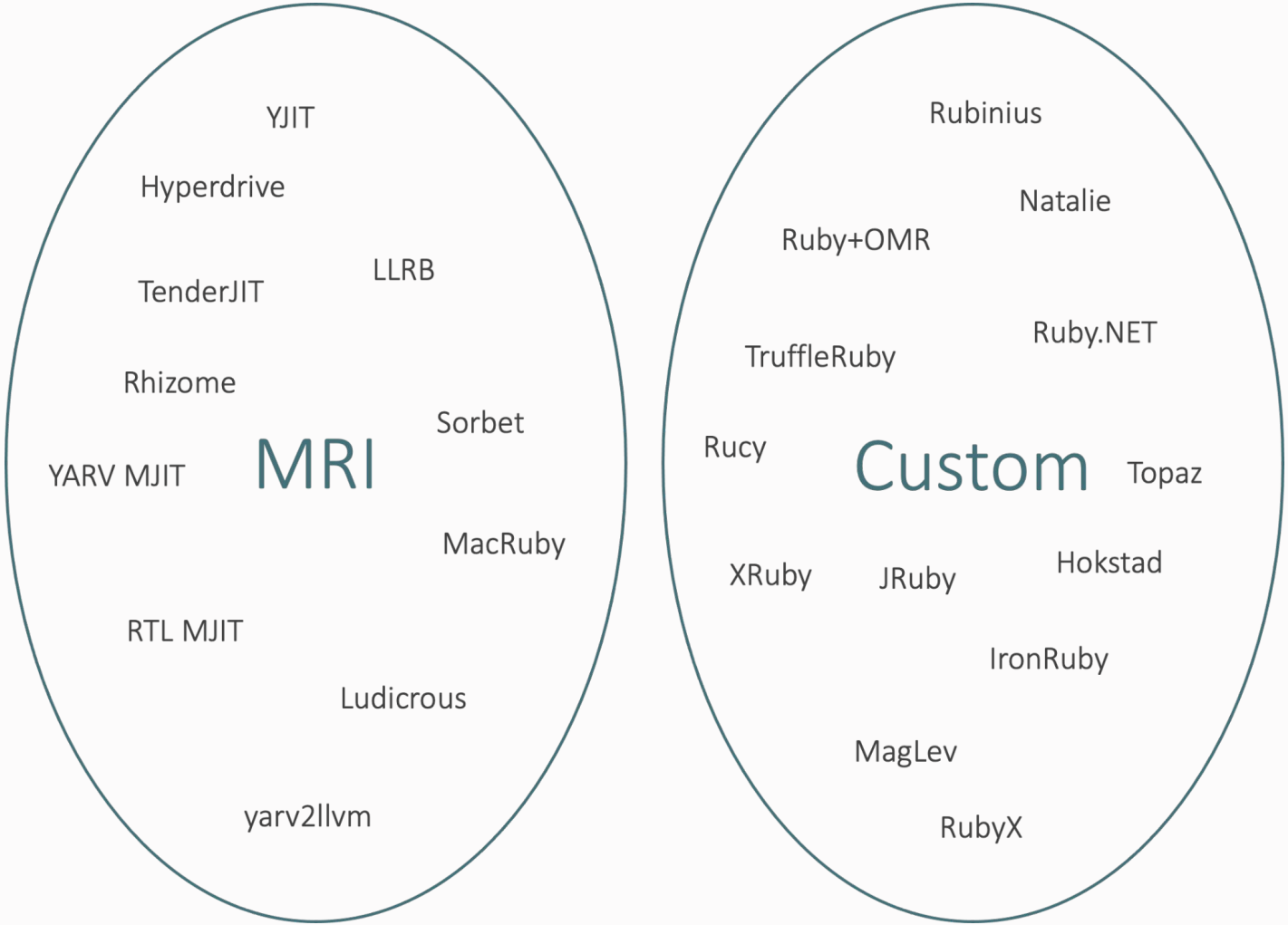

LLRB
Prototype
YARV MJIT
C backend
RTL MJIT
To in-process compiler
MIR MJIT ?

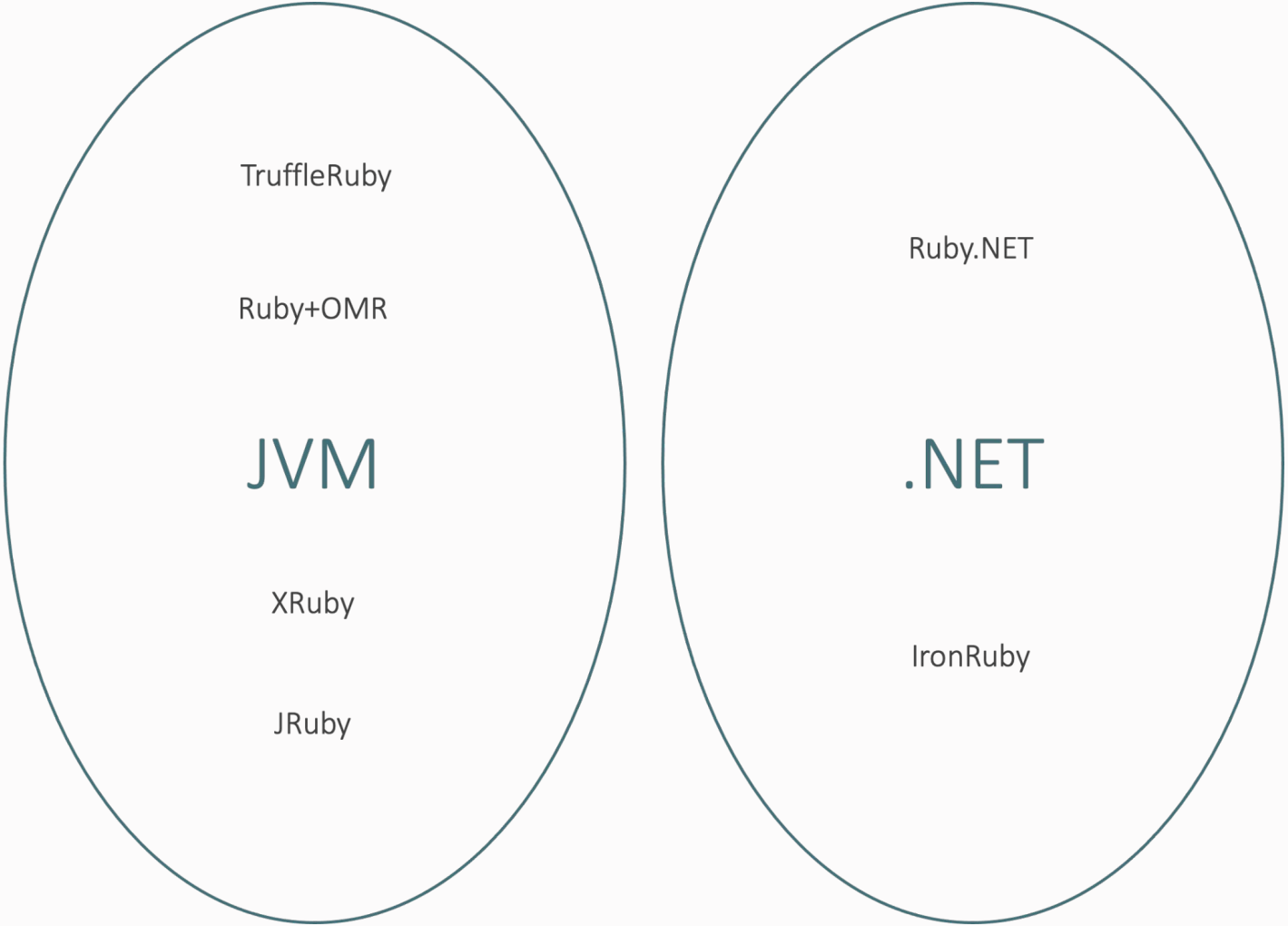

Hyperdrive
Rubinius
LLRB
IronRuby
Ruby+OMR
Topaz
Rhizome
Historial
XRuby
MagLev
RubyComp
RTL MJIT
MacRuby
Ludicrous
RubyX
yarv2llvm
Ruby.NET
Hokstad
YJIT
Natalie
Sorbet
TruffleRuby
Active
YARV MJIT
JRuby
Rucy
TenderJIT


Hokstad
Natalie
RubyComp
Sorbet
AOT
RubyX
XRuby
Rucy
MacRuby
JRuby
Hyperdrive
Ruby.NET
Ludicrous
LLRB
IronRuby
YJIT
RTL MJIT
TruffleRuby
JIT
Topaz
Ruby+OMR
Rubinius
Rhizome
yarv2llvm
TenderJIT
MagLev
YARV MJIT

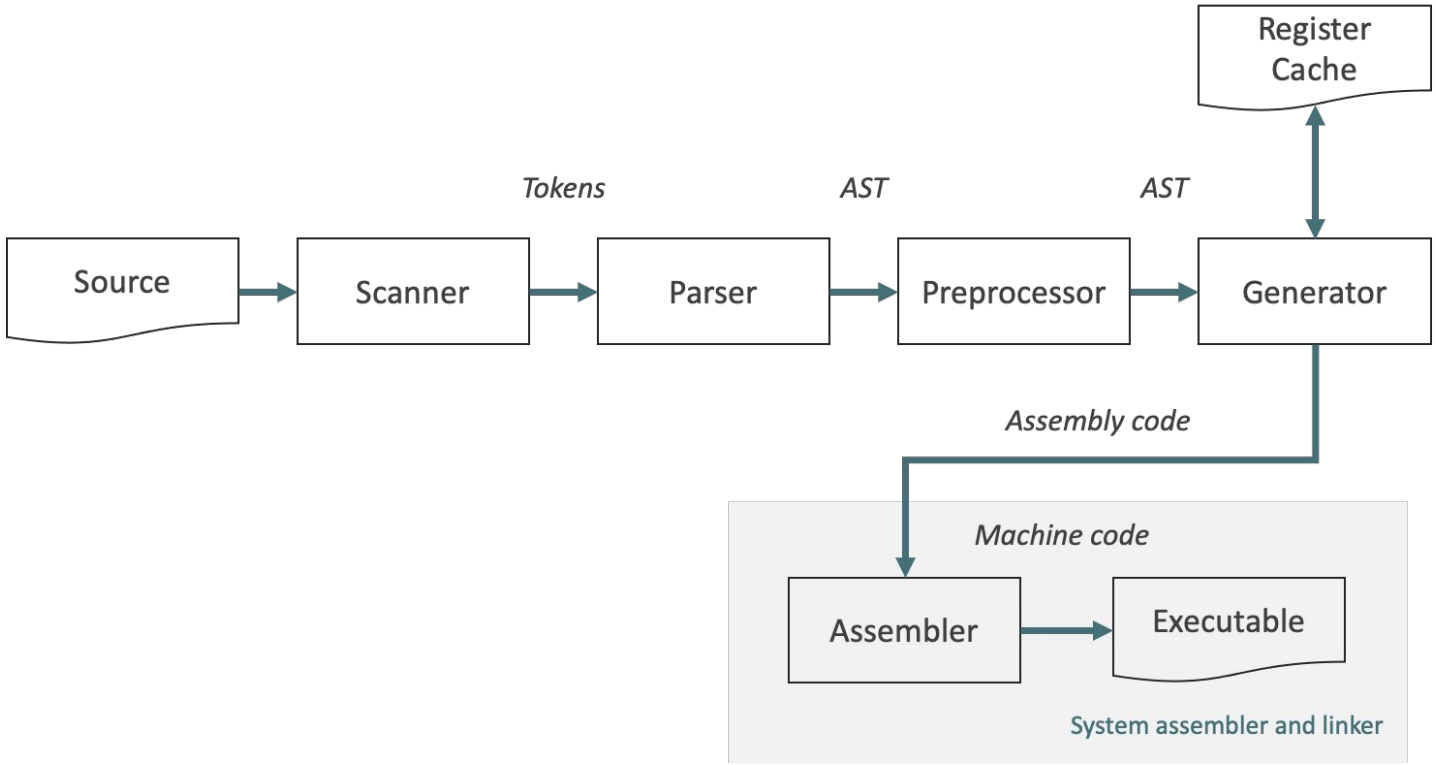

Natalie
Hokstad
Hyperdrive
Ruby.NET
LLRB
Topaz
Rhizome
Rucy
Demo
XRuby
TenderJIT
RubyComp
RTL MJIT
Ludicrous
RubyX
yarv2llvm


Rubinius
Ruby+OMR
Sorbet
TruffleRuby
YJIT
Serious
JRuby
IronRuby
MacRuby
YARV MJIT
MagLev

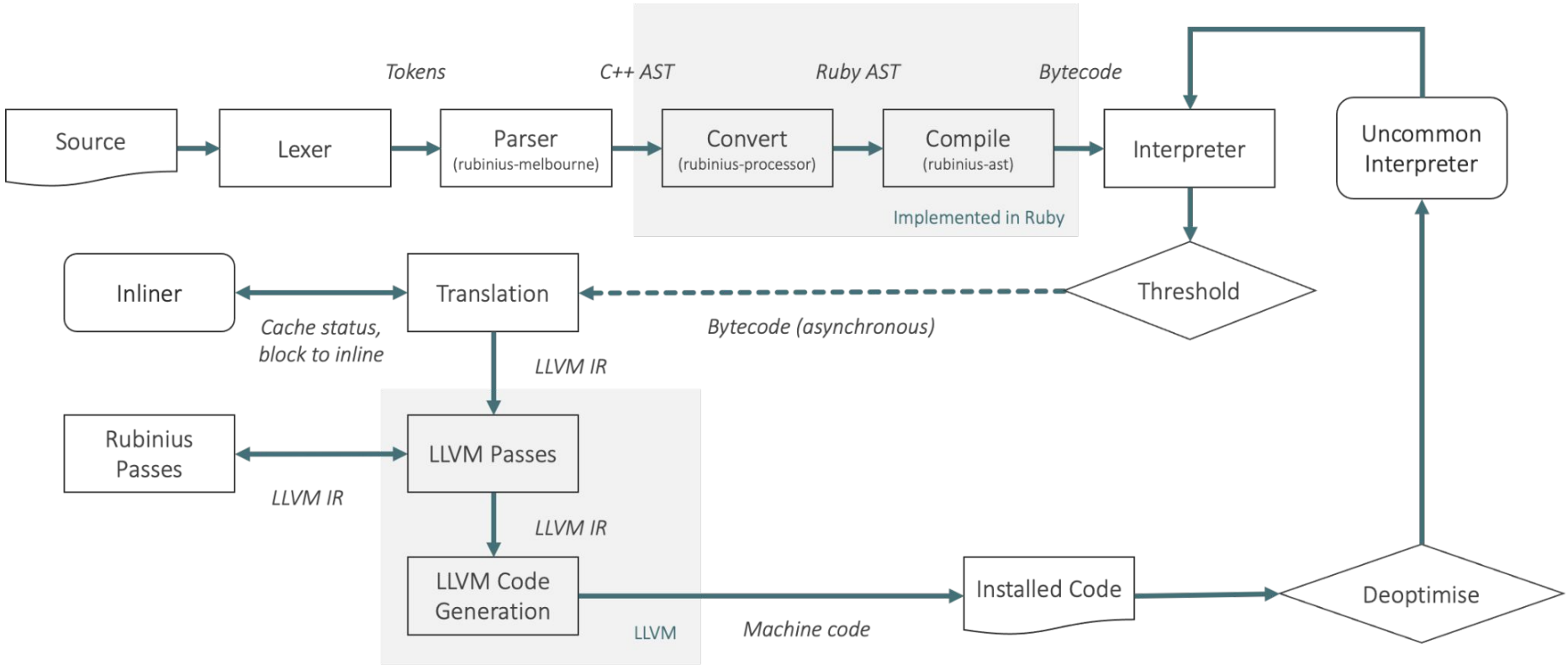

YJIT
Hyperdrive
LLRB
TenderJIT
Rhizome
Sorbet
MRI
YARV MJIT
MacRuby
RTL MJIT
Ludicrous
yarv2llvm
Rubinius
Natalie
Ruby+OMR
Ruby.NET
TruffleRuby
Rucy
Custom
Topaz
XRuby
JRuby
Hokstad
IronRuby
MagLev
RubyX


TruffleRuby
Ruby+OMR
JVM
XRuby
JRuby
Ruby.NET
.NET
IronRuby

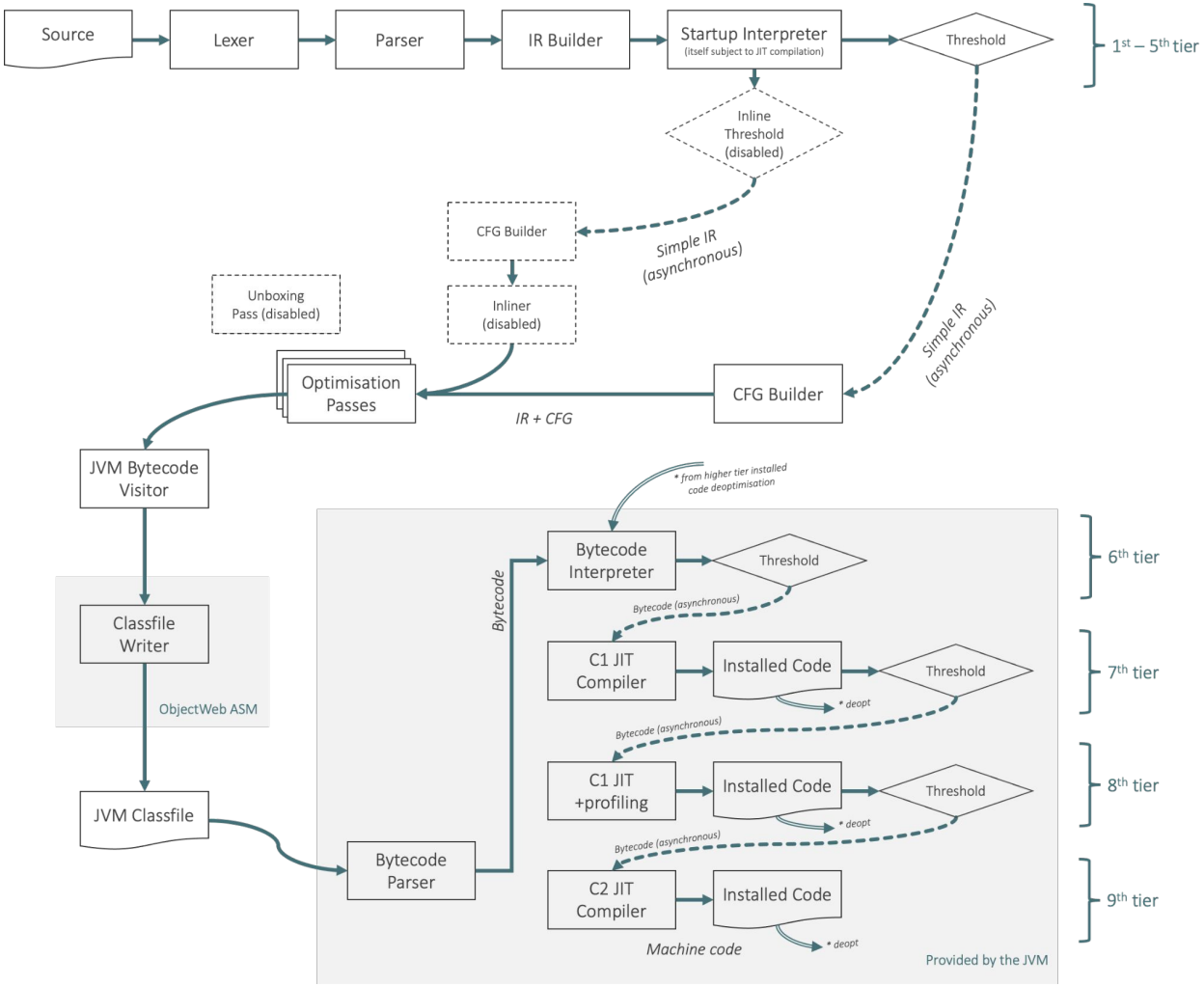
Deep dive into Hokstad

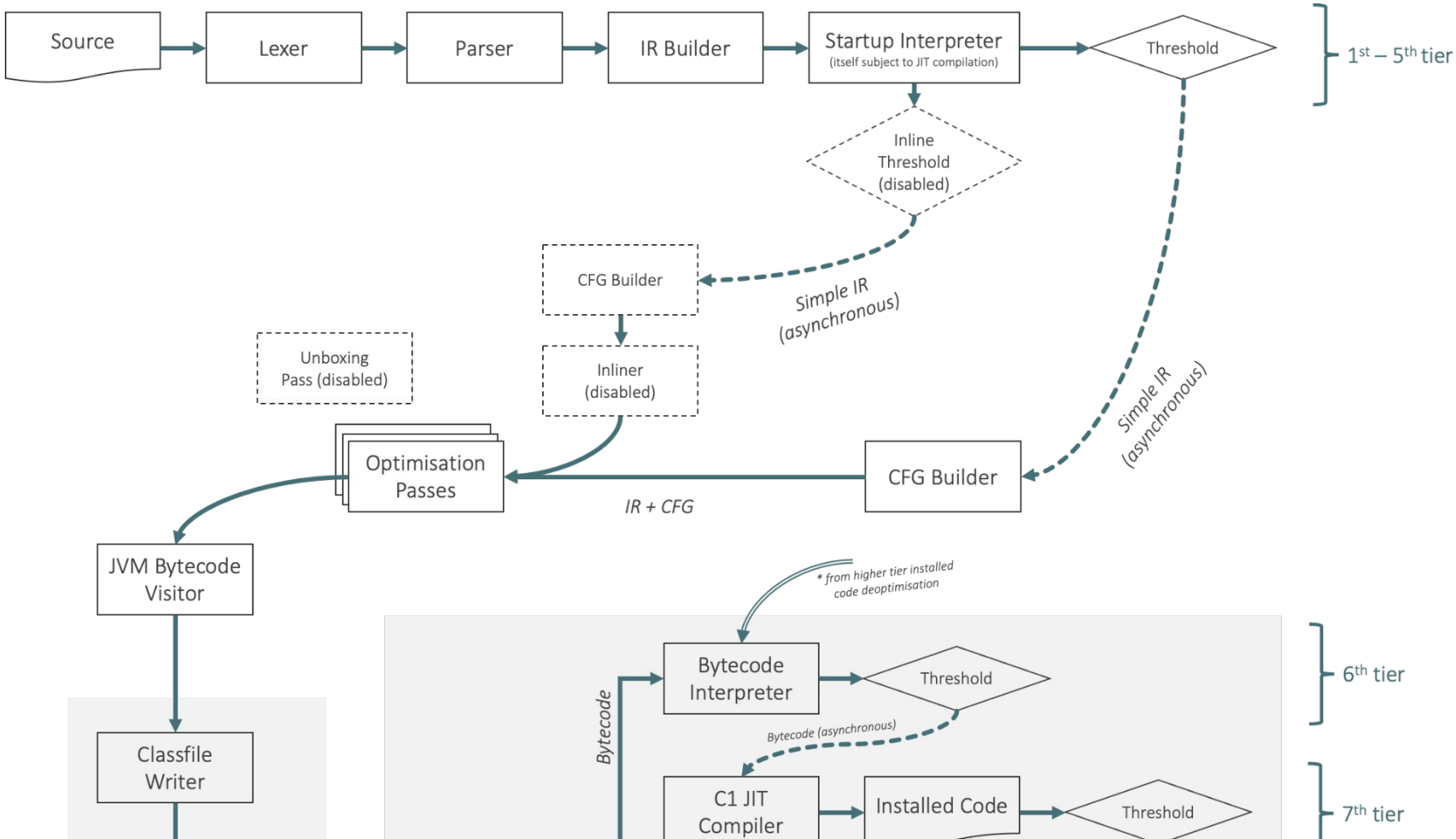

Register
Cache
Tokens
AST
AST
Source
Scanner
Parser
Preprocessor
Generator
Assembly code
Machine code
Assembler
Executable
System assembler and linker

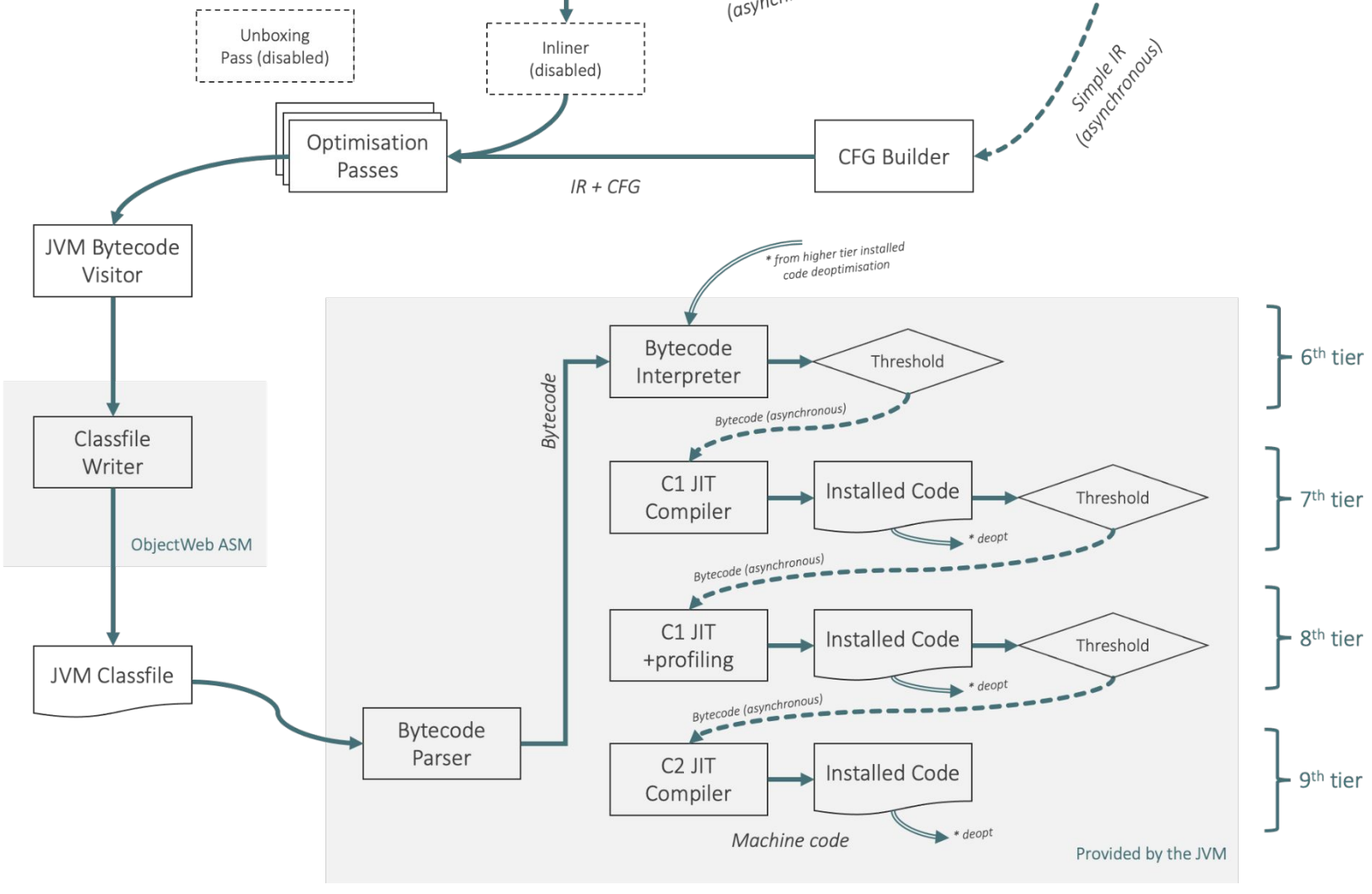
Deep dive into Rubinius


Source
Lexer
Parser
(rubinius-melbourne)
Convert
(rubinius-processor)
Compile
(rubinius-ast)
Implemented in Ruby
Tokens
C++ AST
Ruby AST
Bytecode
Interpreter
Uncommon Interpreter
Inliner
Translation
Threshold
Cache status, block to inline
Bytecode (asynchronous)
LLVM IR
Rubinius Passes
LLVM Passes
LLVM IR
LLVM IR
LLVM Code Generation
LLVM
Machine code
Installed Code
Deoptimise

Deep dive into JRuby


Source
Lexer
Parser
IR Builder
Startup Interpreter
(itself subject to JIT compilation)
Threshold
1st – 5th tier
Inline Threshold (disabled)
CFG Builder
Simple IR (asynchronous)
Unboxing Pass (disabled)
Inliner (disabled)
Optimisation Passes
CFG Builder
IR + CFG
Simple IR (asynchronous)
JVM Bytecode Visitor
Classfile Writer
ObjectWeb ASM
JVM Classfile
Bytecode Parser
* from higher tier installed code deoptimisation
Bytecode
Bytecode Interpreter
Threshold
6th tier
Bytecode (asynchronous)
C1 JIT Compiler
Installed Code
Threshold
* deopt
7th tier
Bytecode (asynchronous)
C1 JIT +profiling
Installed Code
Threshold
* deopt
8th tier
Bytecode (asynchronous)
C2 JIT Compiler
Installed Code
* deopt
9th tier
Machine code
Provided by the JVM


Source
Lexer
Parser
IR Builder
Startup Interpreter
(itself subject to JIT compilation)
Threshold
1st – 5th tier
Inline
Threshold
(disabled)
CFG Builder
Simple IR
(asynchronous)
Unboxing
Pass (disabled)
Inliner
(disabled)
Simple IR
(asynchronous)
Optimisation
Passes
CFG Builder
IR + CFG
JVM Bytecode
Visitor
* from higher tier installed
code deoptimisation
Bytecode
Interpreter
Threshold
6th tier
Bytecode
Bytecode (asynchronous)
Classfile
Writer
C1 JIT
Compiler
Installed Code
Threshold
7th tier


Unboxing Pass (disabled)
Inliner (disabled)
Simple IR (asynchronous)
Optimisation Passes
CFG Builder
IR + CFG
JVM Bytecode Visitor
* from higher tier installed code deoptimisation
Classfile Writer
ObjectWeb ASM
Bytecode
Bytecode Interpreter
Threshold
6th tier
Bytecode (asynchronous)
C1 JIT Compiler
Installed Code
Threshold
7th tier
* deopt
Bytecode (asynchronous)
JVM Classfile
C1 JIT +profiling
Installed Code
Threshold
8th tier
* deopt
Bytecode (asynchronous)
Bytecode Parser
C2 JIT Compiler
Installed Code
9th tier
Machine code
* deopt
Provided by the JVM

Compiler research through Ruby

Tracing

Meta-tracing

Lazy Basic Block Versioning

Self-Specialising ASTs

Partial Evaluation and the Futamura Projection

Tracing

Research idea applied for Ruby in the Hyperdrive JIT by Jacob Matthews

Dynamic Native Optimization of Interpreters

Gregory T. Sullivan
gregs@ai.mit.edu
Artifical Intelligence Lab

Derek L. Bruening
iye@lcs.mit.edu
Laboratory for Computer Science

Iris Baron
iris@lcs.mit.edu
Laboratory for Computer Science

Timothy Garnett
timothyg@lcs.mit.edu
Laboratory for Computer Science

Saman Amarasinghe
saman@lcs.mit.edu
Laboratory for Computer Science

Massachusetts Institute of Technology
Cambridge, MA

Abstract

For domain specific languages, "scripting languages", dynamic languages, and for virtual machine-based languages, the most straightforward implementation strategy is to write an interpreter. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

There is a long history of approaches to removing interpretive overhead from programming language implementations. In practice, what often happens is that, once an interpreted language becomes popular, pressure builds to improve performance until eventually a project is undertaken to implement a native *Just In Time* (JIT) compiler for the language. Implementing a JIT is usually a large effort, affects a significant part of the existing language implementation, and adds a significant amount of code and complexity to the overall code base.

In this paper, we present an innovative approach that dynamically removes much of the interpreted overhead from language implementations, with minimal instrumentation of the original interpreter. While it does not give the performance improvements of hand-crafted native compilers, our system provides an appealing point on the language implementation spectrum.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

IVME'03, June 12, 2003, San Diego, California, USA.

1 Introduction

Certain kinds of programming languages lend themselves to an interpreted implementation: languages with a high degree of dynamism (e.g., dynamic OO languages), or with a premium on fast startup and smaller programs (e.g., scripting languages), or where simplicity of implementation is important. Indeed, it can be argued that nearly all sufficiently complex applications have elements of interpretation throughout[1].

Typically, the language front end will transform the surface syntax of a program to an intermediate representation of the program (e.g., bytecodes) that is then interpreted. A simple interpreter consists of a loop that fetches the next bytecode, dispatches to the routine handling that bytecode, then loops. There are many ways to improve upon this simple mechanism, but as long as the execution of the program is driven by a representation of the program other than as a stream of native instructions, there will be some "interpretive overhead".

We take an approach that is a combination of native *Just In Time* (JIT) compiler and *partial evaluation* techniques. We start with a dynamic optimization system called *DynamoRIO*, jointly developed at HP Labs and MIT. We give a quick overview of the DynamoRIO system in Section 2.

DynamoRIO records sequences of native instructions, which are subsequently partially evaluated with respect to the in-memory representation of the program being interpreted.

Suppose the application program is represented as an immutable vector of bytecodes, and we have a long trace of native instructions. If we look carefully at the trace, there will typically be a value (we call it the "Logical PC") used as an index into the bytecode vector, and there will be sequences of instructions as follows:

[1] Greenspun's Tenth Rule of Programming: "Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."

Meta-tracing

Tracing the Meta-Level: PyPy's Tracing JIT Compiler

Carl Friedrich Bolz
University of Düsseldorf
STUPS Group
Germany
cfbolz@gmx.de

Antonio Cuni
University of Genova
DISI
Italy
cuni@disi.unige.it

Maciej Fijalkowski
merlinux GmbH
fijal@merlinux.eu

Armin Rigo
arigo@tunes.org

ABSTRACT

We attempt to apply the technique of Tracing JIT Compilers in the context of the PyPy project, i.e., to programs that are interpreters for some dynamic languages, including Python. Tracing JIT compilers can greatly speed up programs that spend most of their time in loops in which they take similar code paths. However, applying an unmodified tracing JIT to a program that is itself a bytecode interpreter results in very limited or no speedup. In this paper we show how to guide tracing JIT compilers to greatly improve the speed of bytecode interpreters. One crucial point is to unroll the bytecode dispatch loop, based on two hints provided by the implementer of the bytecode interpreter. We evaluate our technique by applying it to two PyPy interpreters: one is a small example, and the other one is the full Python interpreter.

1. INTRODUCTION

Dynamic languages have seen a steady rise in popularity in recent years. JavaScript is increasingly being used to implement full-scale applications which run within a browser, whereas other dynamic languages (such as Ruby, Perl, Python, PHP) are used for the server side of many web sites, as well as in areas unrelated to the web.

One of the often-cited drawbacks of dynamic languages is the performance penalties they impose. Typically they are slower than statically typed languages. Even though there has been a lot of research into improving the performance of dynamic languages (in the SELF project, to name just one example [18]), those techniques are not as widely used as one would expect. Many dynamic language implementations use completely straightforward bytecode-interpreters without any advanced implementation techniques like just-in-time compilation. There are a number of reasons for this. Most of them boil down to the inherent complexities of using compilation. Interpreters are simple to implement, understand, extend and port whereas writing a just-in-time compiler is an error-prone task that is made even harder by the dynamic features of a language.

A recent approach to getting better performance for dynamic languages is that of tracing JIT compilers [16, 7]. Writing a tracing JIT compiler is relatively simple. It can be added to an existing interpreter for a language, the interpreter takes over some of the functionality of the compiler and the machine code generation part can be simplified.

The PyPy project is trying to find approaches to generally ease the implementation of dynamic languages. It started as a Python implementation in Python, but has now extended its goals to be generally useful for implementing other dynamic languages as well. The general approach is to implement an interpreter for the language in a subset of Python. This subset is chosen in such a way that programs in it can be compiled into various target environments, such as C/Posix, the CLI or the JVM. The PyPy project is described in more detail in Section 2.

In this paper we discuss ongoing work in the PyPy project to improve the performance of interpreters written with the help of the PyPy toolchain. The approach is that of a tracing JIT compiler. Contrary to the tracing JITs for dynamic languages that currently exist, PyPy's tracing JIT operates "one level down", i.e., it traces the execution of the interpreter, as opposed to the execution of the user program. The fact that the program the tracing JIT compiles is in our case always an interpreter brings its own set of problems. We describe tracing JITs and their application to interpreters in Section 3. By this approach we hope to arrive at a JIT compiler that can be applied to a variety of dynamic languages, given an appropriate interpreter for each of them. The process is not completely automatic but needs a small number of hints from the interpreter author, to help the tracing JIT. The details of how the process integrates into the rest of PyPy will be explained in Section 4. This work is not finished, but has already produced some promising results, which we will discuss in Section 5.

The contributions of this paper are:

- Applying a tracing JIT compiler to an interpreter.
- Finding techniques for improving the generated code.

2. THE PYPY PROJECT

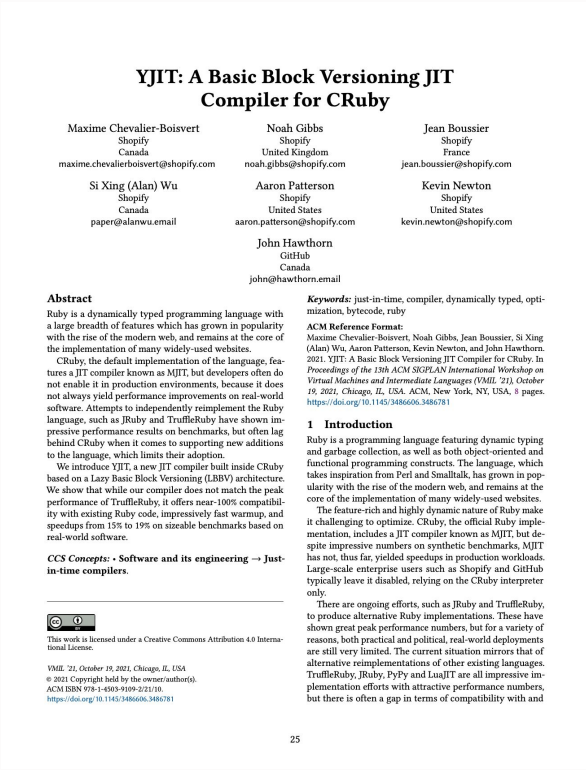
The PyPy project[1] [21, 5] is an environment where flex-

[1]http://codespeak.net/pypy

Research idea applied for Ruby in the Topaz JIT by Alex Gaynor et al

Lazy Basic Block Versioning

Research idea applied for Ruby in YJIT by Maxime Chevalier-Boisvert et al



YJIT: A Basic Block Versioning JIT Compiler for CRuby

Maxime Chevalier-Boisvert
Shopify
Canada
maxime.chevalierboisvert@shopify.com

Noah Gibbs
Shopify
United Kingdom
noah.gibbs@shopify.com

Jean Boussier
Shopify
France
jean.boussier@shopify.com

Si Xing (Alan) Wu
Shopify
Canada
paper@alanwu.email

Aaron Patterson
Shopify
United States
aaron.patterson@shopify.com

Kevin Newton
Shopify
United States
kevin.newton@shopify.com

John Hawthorn
GitHub
Canada
john@hawthorn.email

Abstract

Ruby is a dynamically typed programming language with a large breadth of features which has grown in popularity with the rise of the modern web, and remains at the core of the implementation of many widely-used websites.

CRuby, the default implementation of the language, features a JIT compiler known as MJIT, but developers often do not enable it in production environments, because it does not always yield performance improvements on real-world software. Attempts to independently reimplement the Ruby language, such as JRuby and TruffleRuby have shown impressive performance results on benchmarks, but often lag behind CRuby when it comes to supporting new additions to the language, which limits their adoption.

We introduce YJIT, a new JIT compiler built inside CRuby based on a Lazy Basic Block Versioning (LBBV) architecture. We show that while our compiler does not match the peak performance of TruffleRuby, it offers near-100% compatibility with existing Ruby code, impressively fast warmup, and speedups from 15% to 19% on sizeable benchmarks based on real-world software.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**.

VMIL '21, October 19, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9109-2/21/10.
https://doi.org/10.1145/3486606.3486781

Keywords: just-in-time, compiler, dynamically typed, optimization, bytecode, ruby

ACM Reference Format:
Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. 2021. YJIT: A Basic Block Versioning JIT Compiler for CRuby. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '21), October 19, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3486606.3486781

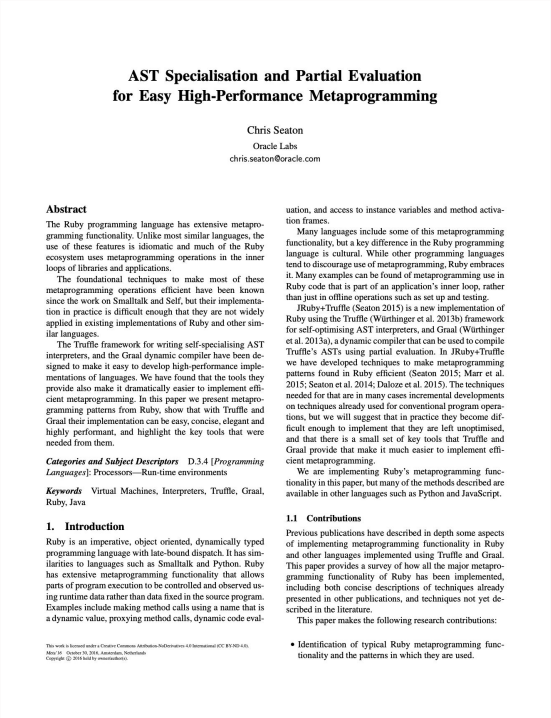
1 Introduction

Ruby is a programming language featuring dynamic typing and garbage collection, as well as both object-oriented and functional programming constructs. The language, which takes inspiration from Perl and Smalltalk, has grown in popularity with the rise of the modern web, and remains at the core of the implementation of many widely-used websites.

The feature-rich and highly dynamic nature of Ruby make it challenging to optimize. CRuby, the official Ruby implementation, includes a JIT compiler known as MJIT, but despite impressive numbers on synthetic benchmarks, MJIT has not, thus far, yielded speedups in production workloads. Large-scale enterprise users such as Shopify and GitHub typically leave it disabled, relying on the CRuby interpreter only.

There are ongoing efforts, such as JRuby and TruffleRuby, to produce alternative Ruby implementations. These have shown great peak performance numbers, but for a variety of reasons, both practical and political, real-world deployments are still very limited. The current situation mirrors that of alternative reimplementations of other existing languages. TruffleRuby, JRuby, PyPy and LuaJIT are all impressive implementation efforts with attractive performance numbers, but there is often a gap in terms of compatibility with and

Self-Specialising ASTs



AST Specialisation and Partial Evaluation for Easy High-Performance Metaprogramming

Chris Seaton
Oracle Labs
chris.seaton@oracle.com

Abstract

The Ruby programming language has extensive metaprogramming functionality. Unlike most similar languages, the use of these features is idiomatic and much of the Ruby ecosystem uses metaprogramming operations in the inner loops of libraries and applications.

The foundational techniques to make most of these metaprogramming operations efficient have been known since the work on Smalltalk and Self, but their implementation in practice is difficult enough that they are not widely applied in existing implementations of Ruby and other similar languages.

The Truffle framework for writing self-specialising AST interpreters, and the Graal dynamic compiler have been designed to make it easy to develop high-performance implementations of languages. We have found that the tools they provide also make it dramatically easier to implement efficient metaprogramming. In this paper we present metaprogramming patterns from Ruby, show that with Truffle and Graal their implementation can be easy, concise, elegant and highly performant, and highlight the key tools that were needed from them.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

Keywords Virtual Machines, Interpreters, Truffle, Graal, Ruby, Java

1. Introduction

Ruby is an imperative, object oriented, dynamically typed programming language with late-bound dispatch. It has similarities to languages such as Smalltalk and Python. Ruby has extensive metaprogramming functionality that allows parts of program execution to be controlled and observed using runtime data rather than data fixed in the source program. Examples include making method calls using a name that is a dynamic value, proxying method calls, dynamic code evaluation, and access to instance variables and method activation frames.

Many languages include some of this metaprogramming functionality, but a key difference in the Ruby programming language is cultural. While other programming languages tend to discourage use of metaprogramming, Ruby embraces it. Many examples can be found of metaprogramming use in Ruby code that is part of an application's inner loop, rather than just in offline operations such as set up and testing.

JRuby+Truffle (Seaton 2015) is a new implementation of Ruby using the Truffle (Würthinger et al. 2013b) framework for self-optimising AST interpreters, and Graal (Würthinger et al. 2013a), a dynamic compiler that can be used to compile Truffle's ASTs using partial evaluation. In JRuby+Truffle we have developed techniques to make metaprogramming patterns found in Ruby efficient (Seaton 2015; Marr et al. 2015; Seaton et al. 2014; Daloze et al. 2015). The techniques needed for that are in many cases incremental developments on techniques already used for conventional program operations, but we will suggest that in practice they become difficult enough to implement that they are left unoptimised, and that there is a small set of key tools that Truffle and Graal provide that make it much easier to implement efficient metaprogramming.

We are implementing Ruby's metaprogramming functionality in this paper, but many of the methods described are available in other languages such as Python and JavaScript.

1.1 Contributions

Previous publications have described in depth some aspects of implementing metaprogramming functionality in Ruby and other languages implemented using Truffle and Graal. This paper provides a survey of how all the major metaprogramming functionality of Ruby has been implemented, including both concise descriptions of techniques already presented in other publications, and techniques not yet described in the literature.

This paper makes the following research contributions:

- Identification of typical Ruby metaprogramming functionality and the patterns in which they are used.

Meta'16 , October 30, 2016, Amsterdam, Netherlands

Research idea applied for Ruby in TruffleRuby by Chris Seaton et al

rubybib.org/#seaton2016

Partial Evaluation

Practical Partial Evaluation for High-Performance Dynamic Language Runtimes

Thomas Würthinger* Christian Wimmer* Christian Humer* Andreas Wöß*
Lukas Stadler* Chris Seaton* Gilles Duboscq* Doug Simon* Matthias Grimmer†
*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
{thomas.wuerthinger, christian.wimmer, christian.humer, andreas.woess, lukas.stadler, chris.seaton, gilles.m.duboscq, doug.simon}@oracle.com matthias.grimmer@jku.at

Abstract

Most high-performance dynamic language virtual machines duplicate language semantics in the interpreter, compiler, and runtime system. This violates the principle to not repeat yourself. In contrast, we define languages solely by writing an interpreter. The interpreter performs specializations, e.g., augments the interpreted program with type information and profiling information. Compiled code is derived automatically using partial evaluation while incorporating these specializations. This makes partial evaluation practical in the context of dynamic languages: It reduces the size of the compiled code while still compiling all parts of an operation that are relevant for a particular program. When a speculation fails, execution transfers back to the interpreter, the program re-specializes in the interpreter, and later partial evaluation again transforms the new state of the interpreter to compiled code. We evaluate our approach by comparing our implementations of JavaScript, Ruby, and R with best-in-class specialized production implementations. Our general-purpose compilation system is competitive with production systems even when they have been heavily optimized for the one language they support. For our set of benchmarks, our speedup relative to the V8 JavaScript VM is 0.83x, relative to JRuby is 3.8x, and relative to GNU R is 5x.

CCS Concepts • **Software and its engineering → Runtime environments**

Keywords dynamic languages; virtual machine; language implementation; optimization; partial evaluation

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06…$15.00
http://dx.doi.org/10.1145/3062341.3062381

1. Introduction

High-performance virtual machines (VMs) such as the Java HotSpot VM or the V8 JavaScript VM follow the design that was first implemented for the SELF language [25]: a multi-tier optimization system with adaptive optimization and de-optimization. The first execution tier, usually an interpreter or fast-compiling baseline compiler, enables fast startup. The second execution tier, a dynamic compiler generating optimized machine code for frequently executed code, provides good peak performance. Deoptimization transitions execution from the second tier back to the first tier, i.e., replaces stack frames of optimized code with frames of unoptimized code when an assumption made by the dynamic compiler no longer holds (see Section 5.2 for details).

Multiple tiers increase the implementation and maintenance costs for a VM: In addition to a language-specific optimizing compiler, a separate first-tier execution system must be implemented [2, 4, 23]. Even though the complexity of an interpreter or a baseline compiler is lower than the complexity of an optimizing compiler, implementing them is far from trivial [48]. Additionally, they need to be maintained and ported to new architectures. But more importantly, the semantics of the language need to be implemented multiple times in different styles: For the first tier, language operations are usually implemented as templates of assembler code. For the second tier, language operations are implemented as graphs of language-specific compiler intermediate representation. And for the runtime system (code called from the interpreter or compiled code using runtime calls), language operations are implemented in C or C++.

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura [15], and is known as the *first Futamura projection*. To the best of our knowledge no prior high-performance language implementation used this approach.

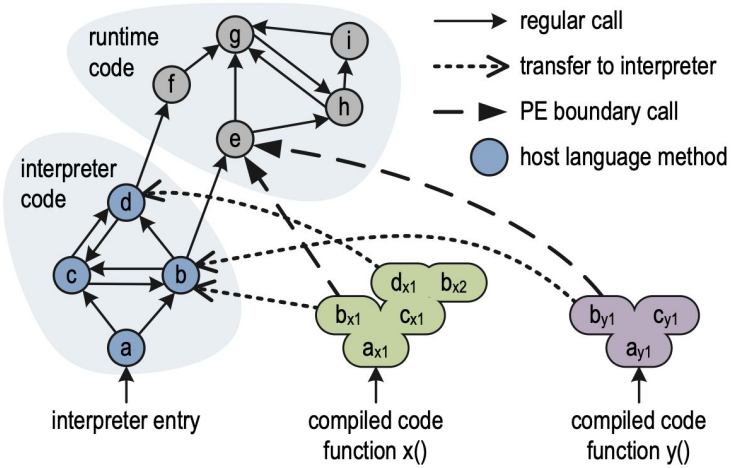


Figure 1: Interactions of interpreter code, compiled code, and runtime code.

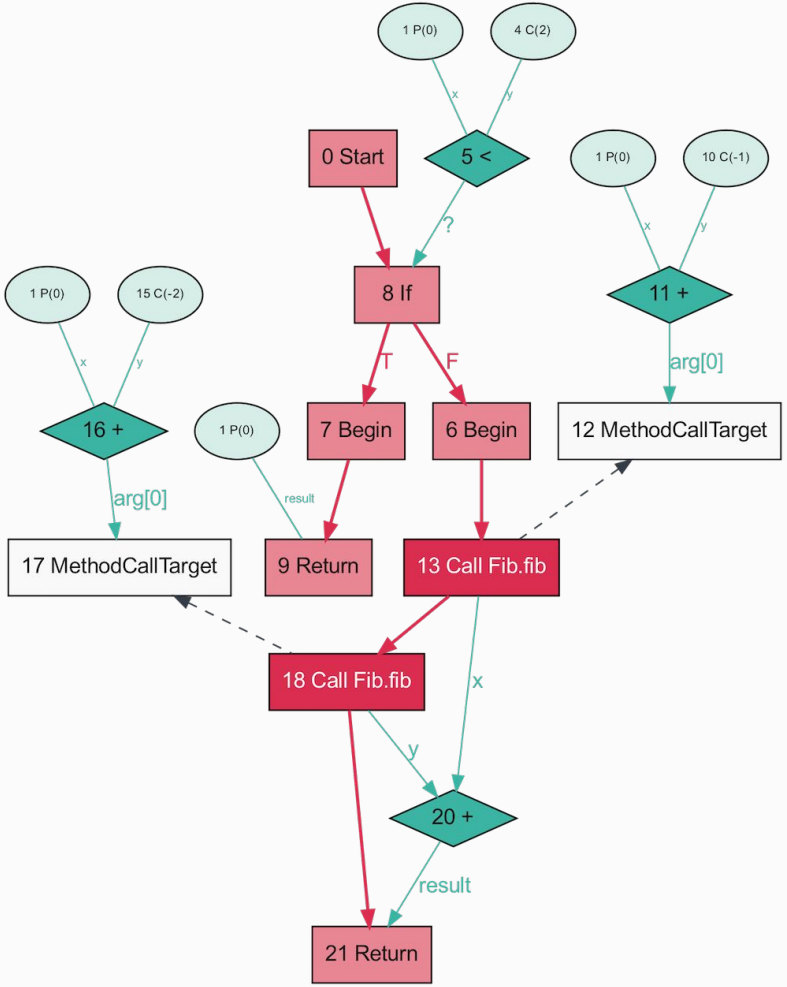
rubybib.org/#wuerthinger2017

Partial Evaluation

Some themes to ruminate on...

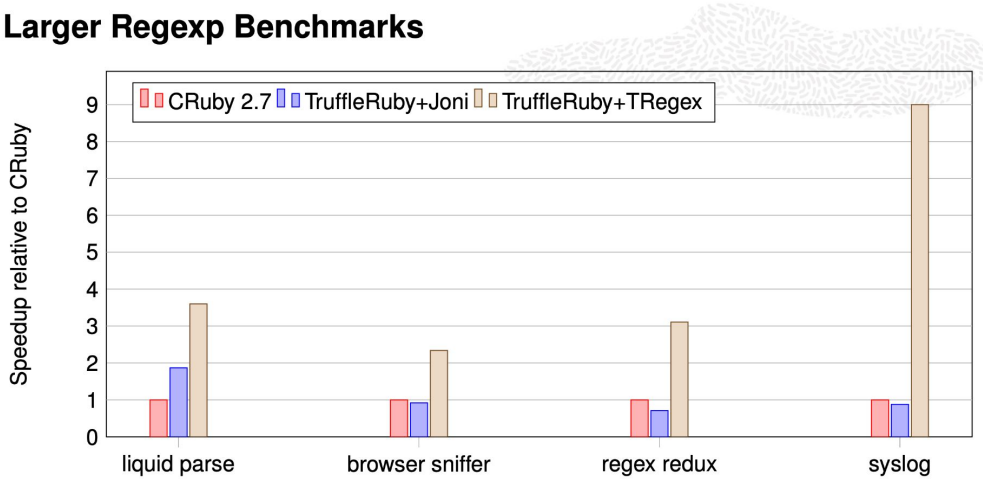
Where we might need to go next?

Intermediate representations


1 P(0)
4 C(2)
x
y
0 Start
5 <
?
8 If
1 P(0)
10 C(-1)
x
y
11 +
arg[0]
1 P(0)
15 C(-2)
x
y
16 +
arg[0]
T
F
1 P(0)
7 Begin
6 Begin
12 MethodCallTarget
result
17 MethodCallTarget
9 Return
13 Call Fib.fib
18 Call Fib.fib
x
y
20 +
result
21 Return

Where we might need to go next?

Polyglot compilation



Talks to catch on compilers at RubyConf

YJIT	**Building a new JIT Compiler inside CRuby**, Wed 10:50 am
JRuby	**JRuby in 2021**, Tue 1:40 pm
YARV MJIT	**Optimizing Ruby Production Perf with MRI JIT**, *watch anytime*
TruffleRuby	**JIT Compiling Ruby Regexps**, *watch anytime*
Sorbet Compiler	**Compiling Ruby to Native Code with Sorbet and LLVM**, Wed 1 pm
TenderJIT	**Some Assembly Required**, Tue 10:50 am

People to watch and read for Ruby compilers

Tim Morgan Live-streaming his Natalie Ruby compiler on YouTube

youtube.com/c/TimMorgan

Vidar Hokstad 45-article blog post series about his Ruby compiler

hokstad.com/compiler

Aaron Patterson Live-streaming his TenderJIT compiler on YouTube

youtube.com/c/TenderlovesCoolStuff

Why are Ruby compilers so cool?

- You can build one yourself
- So much freedom for how to design and architect them
- Tons of open space for new ideas
- You can apply research
- Really active area!

ruby-compilers.com

The Ruby Compiler Survey

rubybib.org

Academic writing on the Ruby programming language

@ChrisGSeaton