

YJIT: Building a New JIT Compiler Inside CRuby

Maxime Chevalier-Boisvert @ RubyKaigi 2021



Agenda

- About YJIT
- Why we're building YJIT inside CRuby
- How it works
- Current performance results
- Performance bottlenecks in CRuby

YJIT

- Y JIT? Why not JIT?
- YARV bytecode
- YARV JIT
- Yet Another Ruby JIT



About YJIT

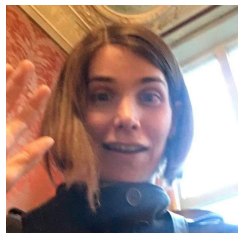
- Goal: speedups on real-world software
 - Focus on web workloads, Ruby on Rails
- CRuby codebase is very complex
 - Fall back to the interpreter for anything we don't support
 - Incrementally increase coverage, get more sophisticated
- Benefits of our approach:
 - Highly compatible with all existing Ruby code
 - Supports latest Ruby features
 - Highly underrated
- Built at Shopify, but fully open source (BSD)
 - We hope to eventually upstream this into CRuby
 - Find ways to help both YJIT and MJIT

<https://github.com/Shopify/yjit>



The Team

Project led by a team in the Ruby & Rails Infrastructure group at Shopify,
with multiple major contributions from GitHub



Maxime
@Love2Code



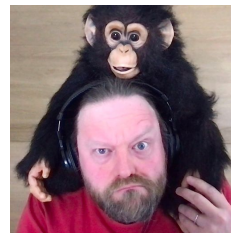
Alan Wu
@alanwusx



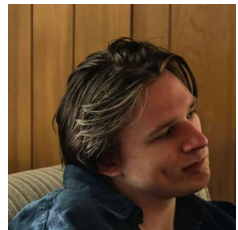
Aaron Patterson
@tenderlove



Kevin Newton
@kddnewton



Noah Gibbs
@codefolio



Jean Boussier



John Hawthorn
@jhawthorn



Eileen Uchitelle
@eileencodes

This project would not be
possible without them!

The Space of Ruby JITs

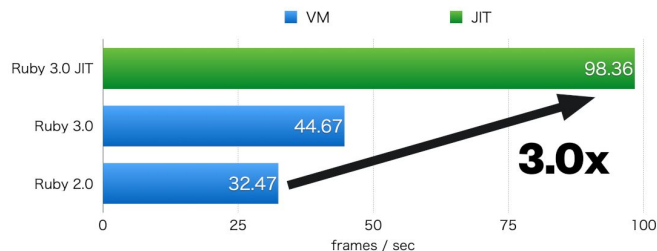


- MJIT
- JRuby
- TruffleRuby
- MIR
- MagLev
- Many more no longer maintained

MJIT

- JIT integrated inside CRuby, based on GCC
- Strengths:
 - Already in CRuby!
 - Compatible with latest Ruby features
 - Good speedups on smaller benchmarks
 - Officially supported on multiple platforms
- Trade-offs:
 - Performs best on synthetic benchmarks (large methods)
 - GCC is not equipped to optimize dynamically-typed code
 - Limited control over codegen pipeline, what GCC does
 - Doesn't always yield speedups on large programs

Optcarrot 3000 frames





**TRUFFLE
RUBY**

- Alternative implementation of Ruby based on Truffle/Graal
- Strengths:
 - In active development
 - Powerful optimizer based on partial evaluation
 - Huge speedups over CRuby on many benchmarks
 - Good support for C extensions
- Trade-offs:
 - Warmup time
 - Memory usage
 - Reimplementation of Ruby, not a drop-in replacement

Language Compatibility

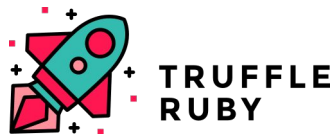
- JRuby

- Ruby 2.5.7 compatibility, working on 2.6



- TruffleRuby

- In the work since 2013, tremendous performance
- Reaching 2.7 compatibility



- LuaJIT

- Impressive optimizing JIT, well-loved by the community
- Compatible with LuaJIT 5.1 (Lua at 5.4)



- PyPy

- Initial release in 2007
- Clearly faster than CPython in most cases
- Partially compatible, Python 3.6 syntax (CPython at 3.9)



How YJIT Optimizes Ruby

[Home](#) > [Core Java](#)

So You Want To Optimize Ruby



By **Charles Nutter**

JavaWorld | OCT 15, 2012 9:26 AM PST

I was recently asked for a list of "hard problems" a Ruby implementation really needs to solve before reporting benchmark numbers. You know...the sort of problems that might invalidate early perf numbers because they impact how you optimize Ruby. This post is a rework of my response...I hope you find it informative!

Fixnum to Bignum promotion

In Ruby, Fixnum math can promote to Bignum when the result is out of Fixnum's range. On implementations that use tagged pointers to represent Fixnum (MRI, Rubinius, MacRuby), the Fixnum range is somewhat less than the base CPU bits (32/64). On JRuby, Fixnum is always a straight 64-bit signed value.

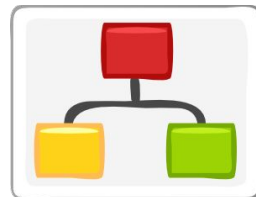
Ruby Optimization Challenges

- Every operation on every basic type can be redefined
 - Arithmetic ops on integers, floats, etc.
 - The meaning of `x != nil`
- Methods can be redefined
- Constants are not actually constant
- Callees can read/write locals in callers
- The method call logic is highly complex
 - CRuby implements *11 kinds* of method dispatch
- Real-world code has lots of small methods
 - It's method calls all the way down
 - Inlining is extra challenging
- C extensions API



Lazy Basic Block Versioning

- Research work started during PhD at UdeM
 - Built Higgs, an optimizing JIT for JavaScript
 - Optimizing dynamic languages, eliminating dynamic type checks
- Type specialization without type analysis
 - Traditional whole-program analysis is costly
 - What can we leverage in a JIT context?
- Reimagining of what a JIT compiler could be
 - Traditional JIT compilers are method-based
 - BBV operates at a lower level of granularity
 - Lightweight technique, single-pass
- Small but diverse body of literature



Simple and Effective Type Check Removal through Lazy Basic Block Versioning

Maxime Chevalier-Boisvert¹ and Marc Feeley²

1 DIRO, Université de Montréal
Montréal, QC, Canada
chevalma@iro.umontreal.ca

2 DIRO, Université de Montréal
Montréal, QC, Canada
feeley@iro.umontreal.ca

Abstract

Dynamically typed programming languages such as JavaScript and Python defer type checking to run time. In order to maximize performance, dynamic language VM implementations must attempt to eliminate redundant dynamic type checks. However, type inference analyses are often costly and involve tradeoffs between compilation time and resulting precision. This has led to the creation of increasingly complex multi-tiered VM architectures.

This paper introduces *lazy basic block versioning*, a simple JIT compilation technique which effectively removes redundant type checks from critical code paths. This novel approach lazily generates type-specialized versions of basic blocks on-the-fly while propagating context-dependent type information. This does not require the use of costly program analyses, is not restricted by the precision limitations of traditional type analyses and avoids the implementation complexity of speculative optimization techniques.

Interprocedural Type Specialization of JavaScript Programs Without Type Analysis

Maxime Chevalier-Boisvert

DIRO, Université de Montréal, Quebec, Canada
chevalma@iro.umontreal.ca

Marc Feeley

DIRO, Université de Montréal, Quebec, Canada
feeley@iro.umontreal.ca

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization, code generation, run-time environments

Keywords Just-In-Time Compilation, Dynamic Language, Optimization, Object Oriented, JavaScript

Abstract

Dynamically typed programming languages such as Python and JavaScript defer type checking to run time. VM implementations can improve performance by eliminating redundant dynamic type checks. However, type inference analyses are often costly and involve tradeoffs between compilation time and resulting precision. This has led to the creation of increasingly complex multi-tiered VM architectures.

Lazy basic block versioning is a simple JIT compilation technique which effectively removes redundant type checks from critical code paths. This novel approach lazily generates type-specialized versions of basic blocks on-the-fly while propagating context-dependent type information. This approach does not require the use of costly program analyses, is not restricted by the precision limitations of traditional type analyses.

This paper extends lazy basic block versioning to propa-

Truffle/JS on several benchmarks, both in terms of execution time and compilation time.

1. Introduction

The highly dynamic semantics of JavaScript (JS) make optimization difficult. Late binding, dynamic code loading and the eval construct make type analysis a hard problem. Precise type analyses also tend to be expensive, and are often considered too costly to be used in Just-In-Time (JIT) compilers.

Lazy Basic Block Versioning (BBV) [12] is an intraprocedural JIT compilation strategy which allows rapid and effective generation of type-specialized machine code on-the-fly without a separate type analysis pass (Section 2.2).

In this paper we introduce an interprocedural variant which extends BBV with mechanisms to propagate type information interprocedurally, across function call boundaries (Section 3), both function parameter and return value types. Combining these new elements with BBV yields a lightweight approach to interprocedurally type-specialize programs on-the-fly without performing global type inference or type analysis in a separate pass.

A detailed evaluation of the performance implications is provided in Section 5. Empirical results across 26 bench-

Interprocedural Specialization of Higher-Order Dynamic Languages Without Static Analysis

Baptiste Saleil¹ and Marc Feeley²

1 Université de Montréal
Montreal, Quebec, Canada
baptiste.saleil@umontreal.ca

2 Université de Montréal
Montreal, Quebec, Canada
feeley@iro.umontreal.ca

Abstract

Function duplication is widely used by JIT compilers to efficiently implement dynamic languages. When the source language supports higher order functions, the called function's identity is not generally known when compiling a call site, thus limiting the use of function duplication.

This paper presents a JIT compilation technique enabling function duplication in the presence of higher order functions. Unlike existing techniques, our approach uses dynamic dispatch at call sites instead of relying on a conservative analysis to discover function identity.

We have implemented the technique in a JIT compiler for Scheme. Experiments show that it is efficient at removing type checks, allowing the removal of almost all the run time type checks for several benchmarks. This allows the compiler to generate code up to 50% faster.

We show that the technique can be used to duplicate functions using other run time information opening up new applications such as register allocation based duplication and aggressive inlining.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases Just-in-time compilation, Interprocedural optimization, Dynamic language, Higher-order function, Scheme



The VM Already Knew That

Leveraging Compile-Time Knowledge to Optimize Gradual Typing

GREGOR RICHARDS, University of Waterloo, Canada, Canada

ELLEN ARTECA, University of Waterloo, Canada, Canada

ALEXI TURCOTTE, University of Waterloo, Canada, Canada

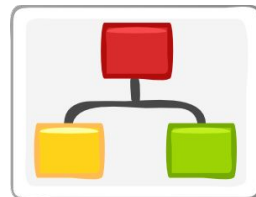
Programmers in dynamic languages wishing to constrain and understand the behavior of their programs may turn to gradually-typed languages, which allow types to be specified optionally and check values at the boundary between dynamic and static code. Unfortunately, the performance cost of these run-time checks can be severe, slowing down execution by at least 10x when checks are present. Modern virtual machines (VMs) for dynamic languages use speculative techniques to improve performance: If a particular value was seen once, it is likely that similar values will be seen in the future. They combine optimization-relevant properties of values into cacheable “shapes”, then use a single shape check to subsume checks for each property. Values with the same memory layout or the same field types have the same shape. This greatly reduces the amount of type checking that needs to be performed at run-time to execute dynamic code. While very valuable to the VM’s optimization, these checks do little to benefit the programmer aside from improving performance. We present in this paper a design for *intrinsic object contracts*, which makes the obligations of gradually-typed languages’ type checks an intrinsic part of object shapes, and thus can subsume run-time type checks into existing shape checks, eliminating redundant checks entirely. With an implementation on a VM for JavaScript used as a target for SafeTypeScript’s soundness guarantees, we demonstrate slowdown averaging 7% in fully-typed code relative to unchecked code, and no more than 45% in pessimal configurations.

CCS Concepts: • **Software and its engineering** → **Just-in-time compilers**; **Runtime environments**;

Additional Key Words and Phrases: Gradual typing, run-time type checking

Two Key Components of Lazy BBV

- Versioning of basic blocks
 - Accumulate and propagate (type) information
 - Specialize basic block based on contexts
 - Selective tail-duplication, “unfold” the control-flow graph
- Lazy code generation
 - Lazy tail-duplication
 - It's like lazy evaluation for code
 - CFG as an infinite data structure
 - Generate code only when needed, truly Just-In-Time



```
if is_fixnum(n) # A
```

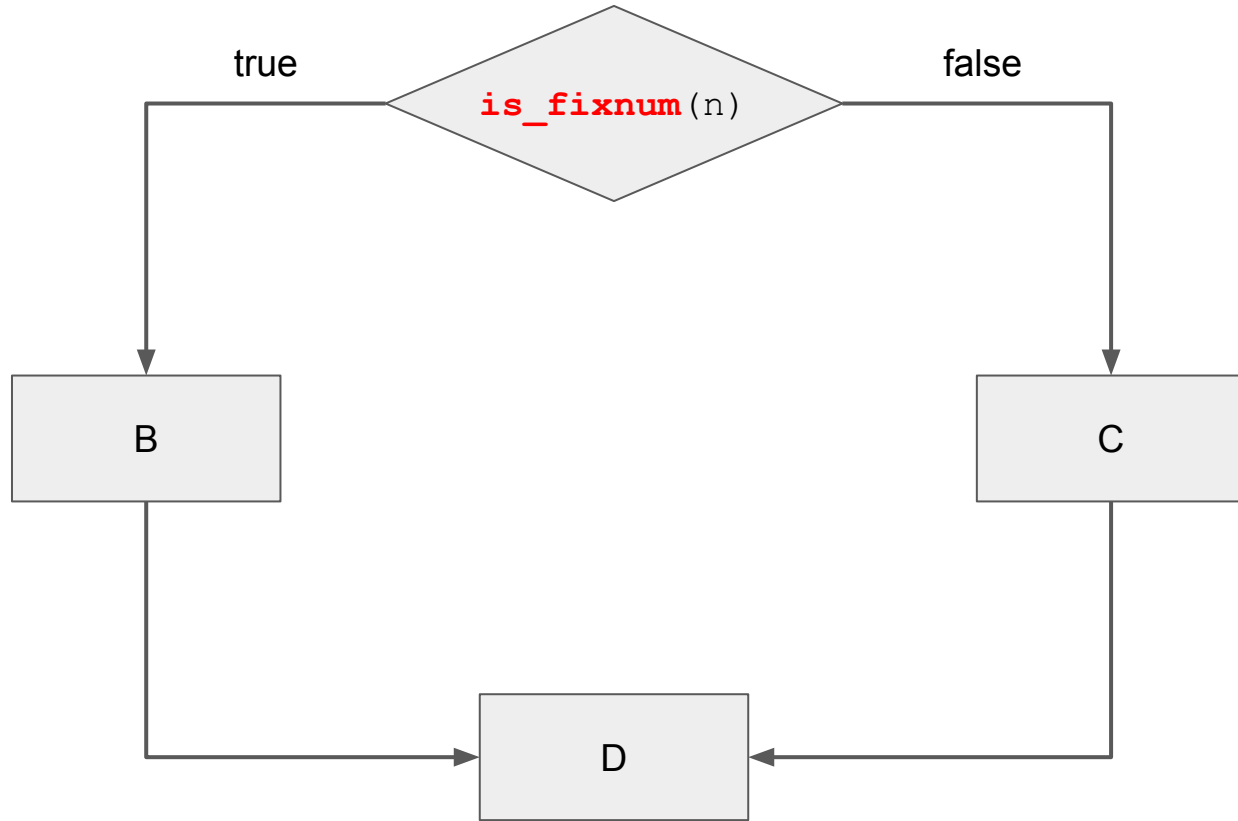
```
    # B ...
```

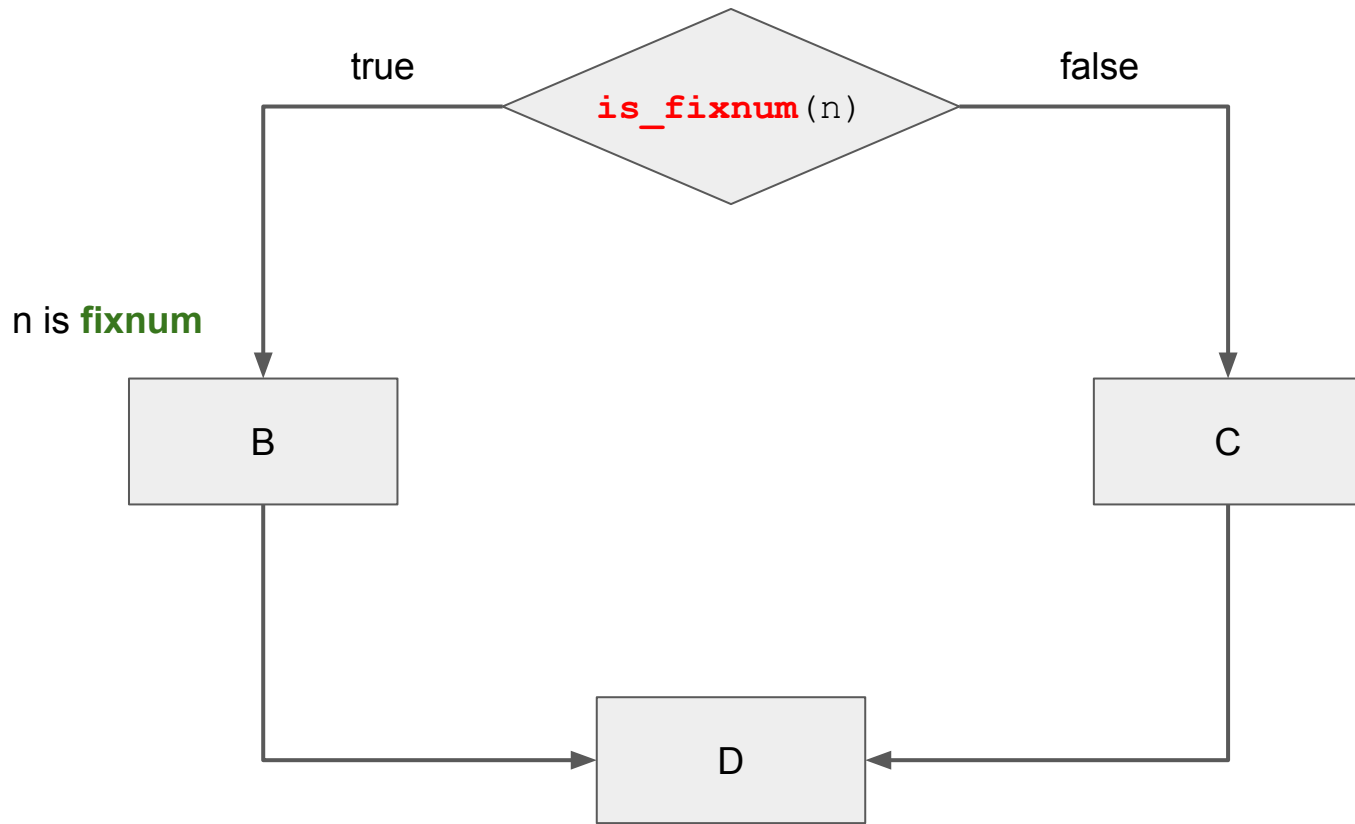
```
else
```

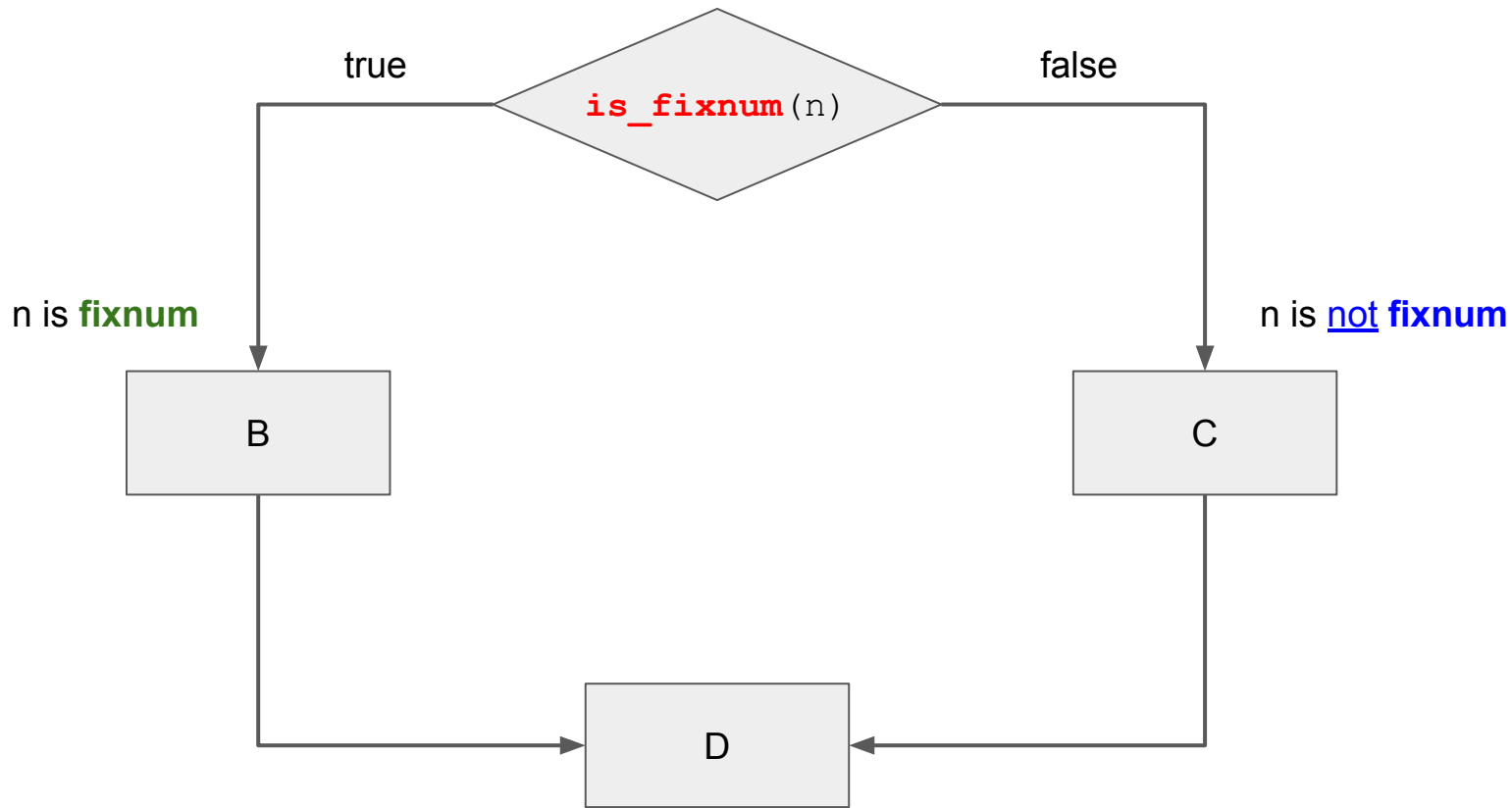
```
    # C ...
```

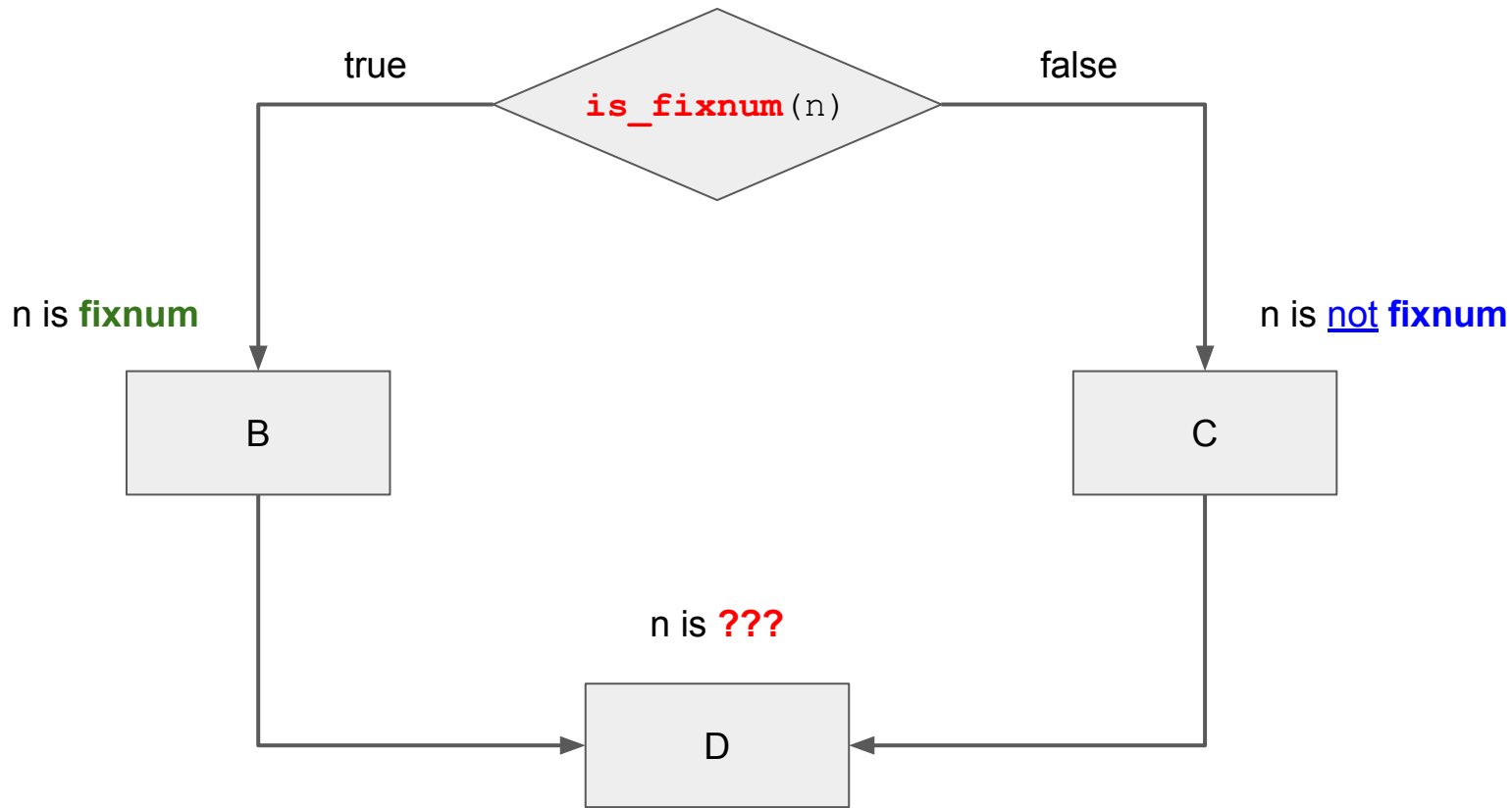
```
end
```

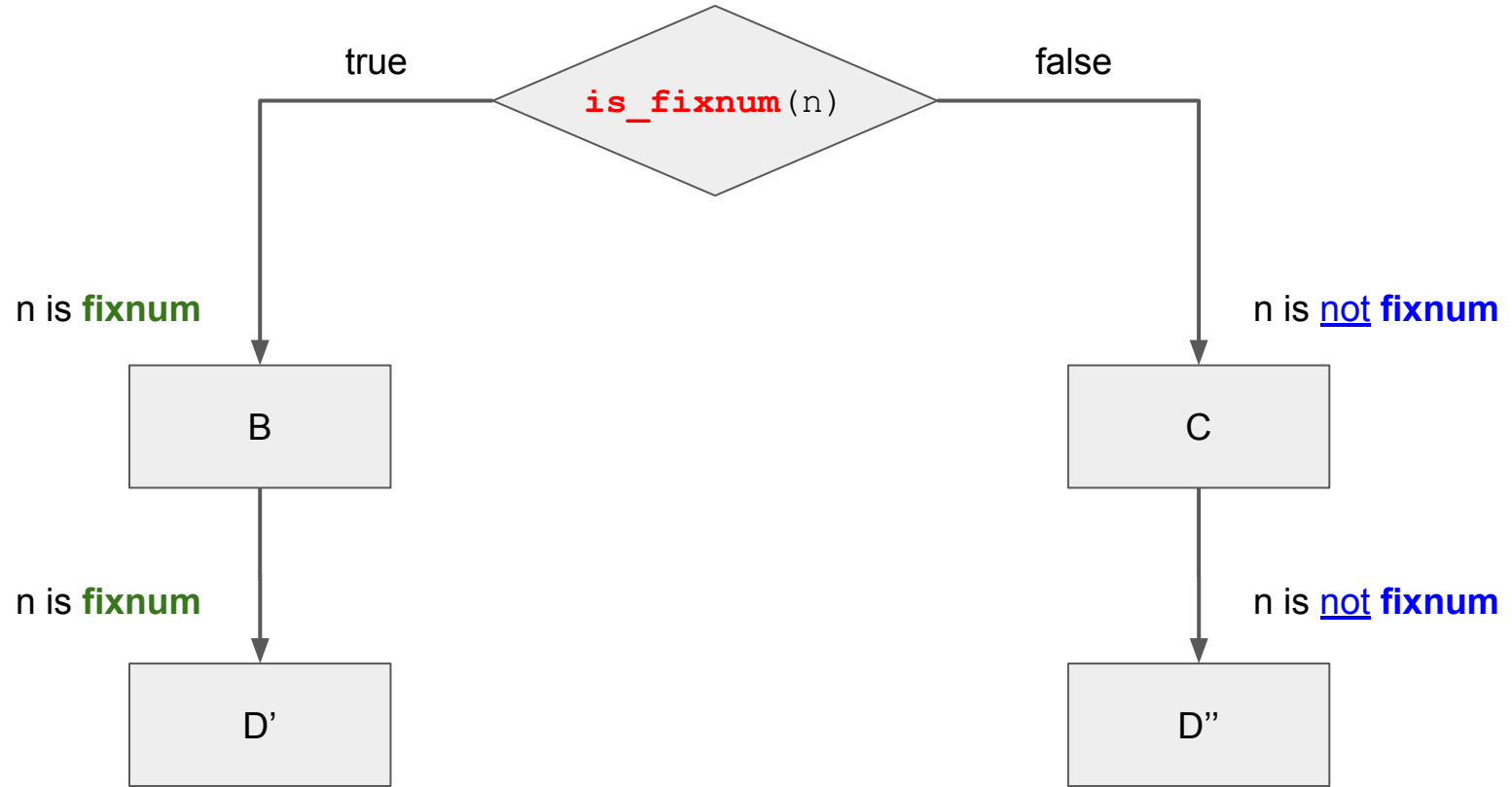
```
# D ...
```

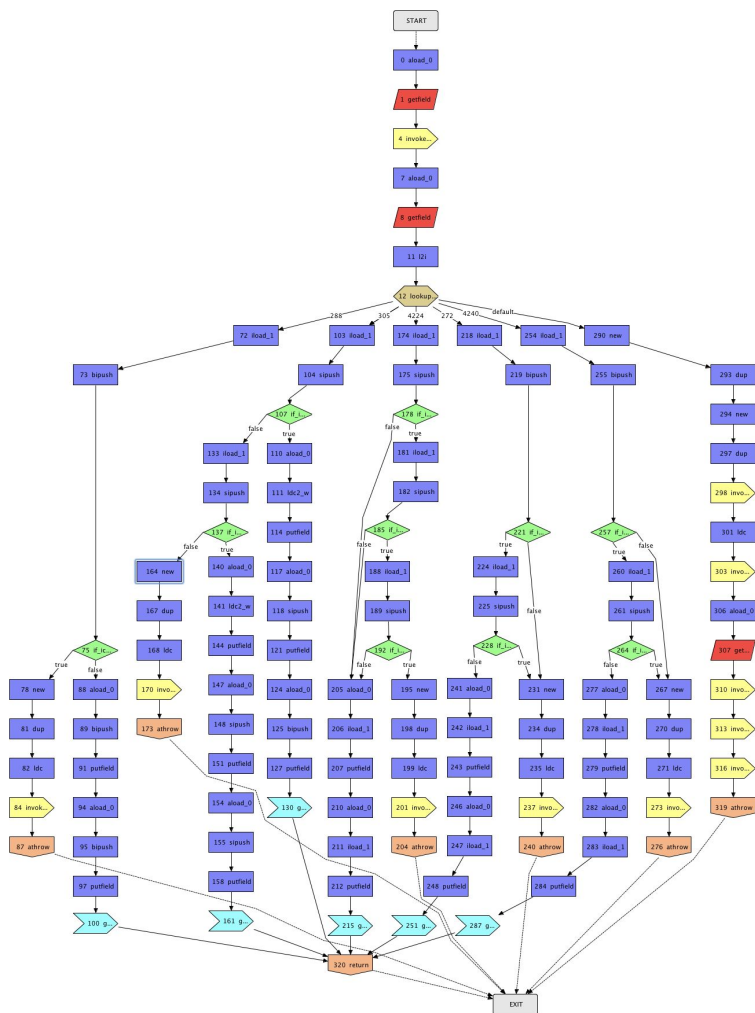


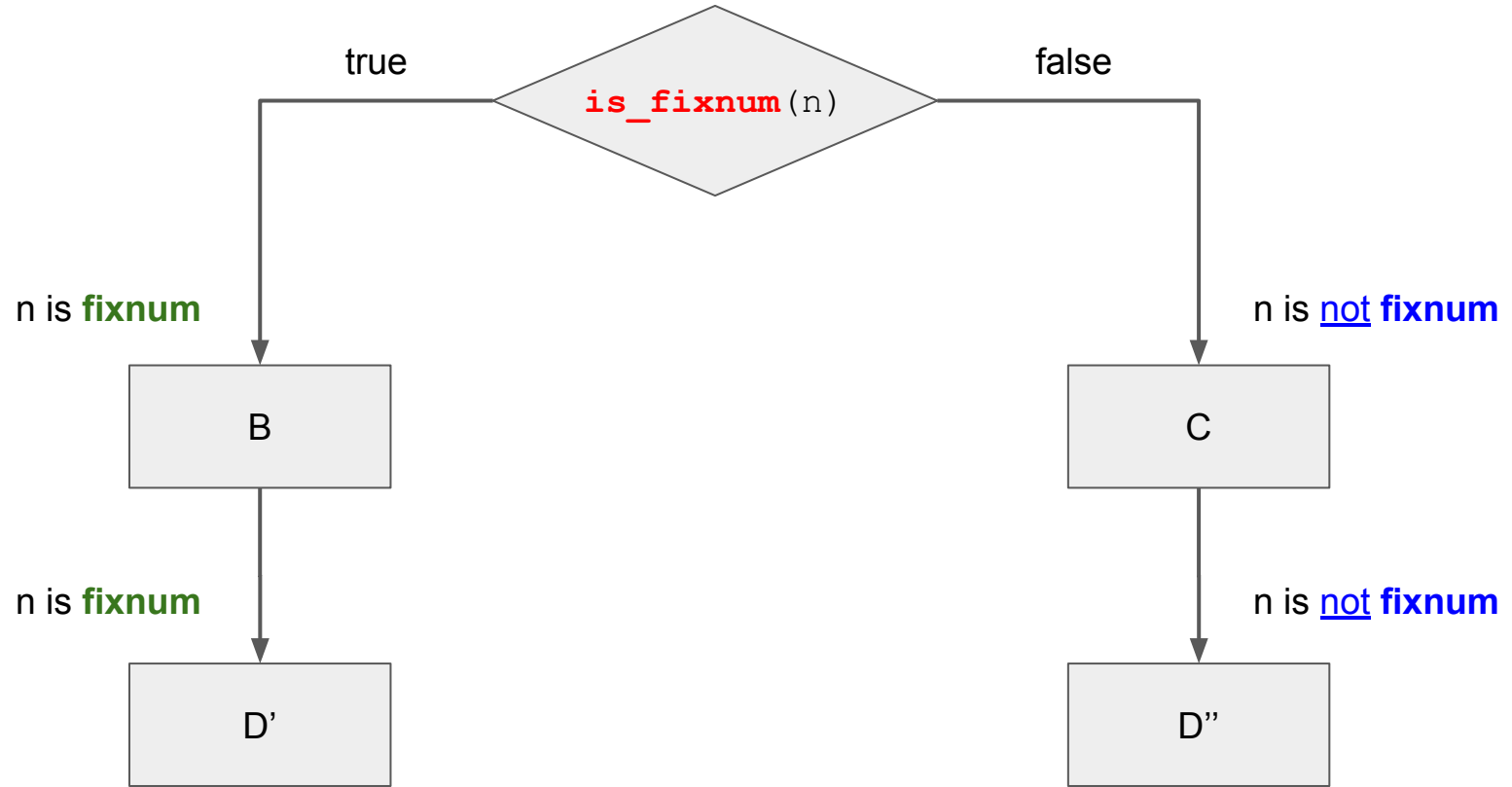


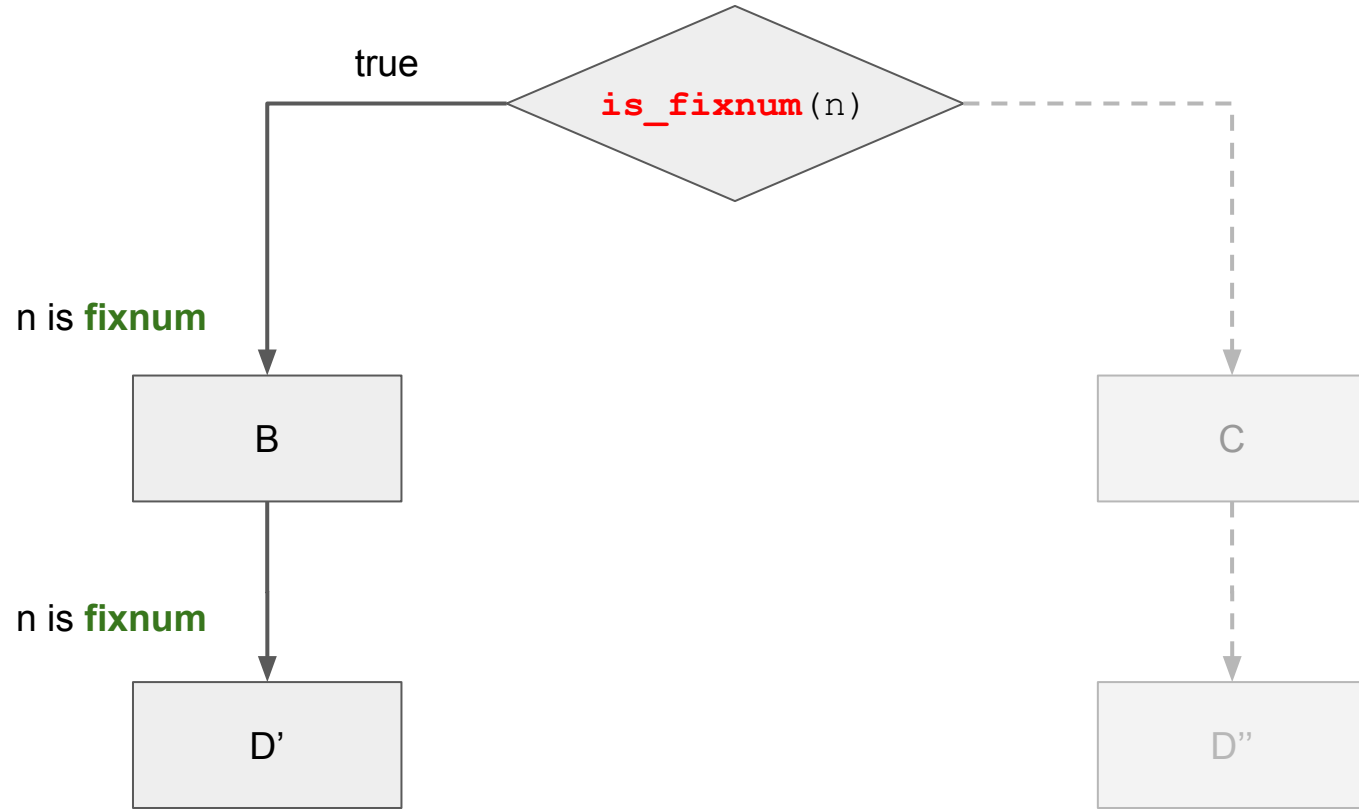


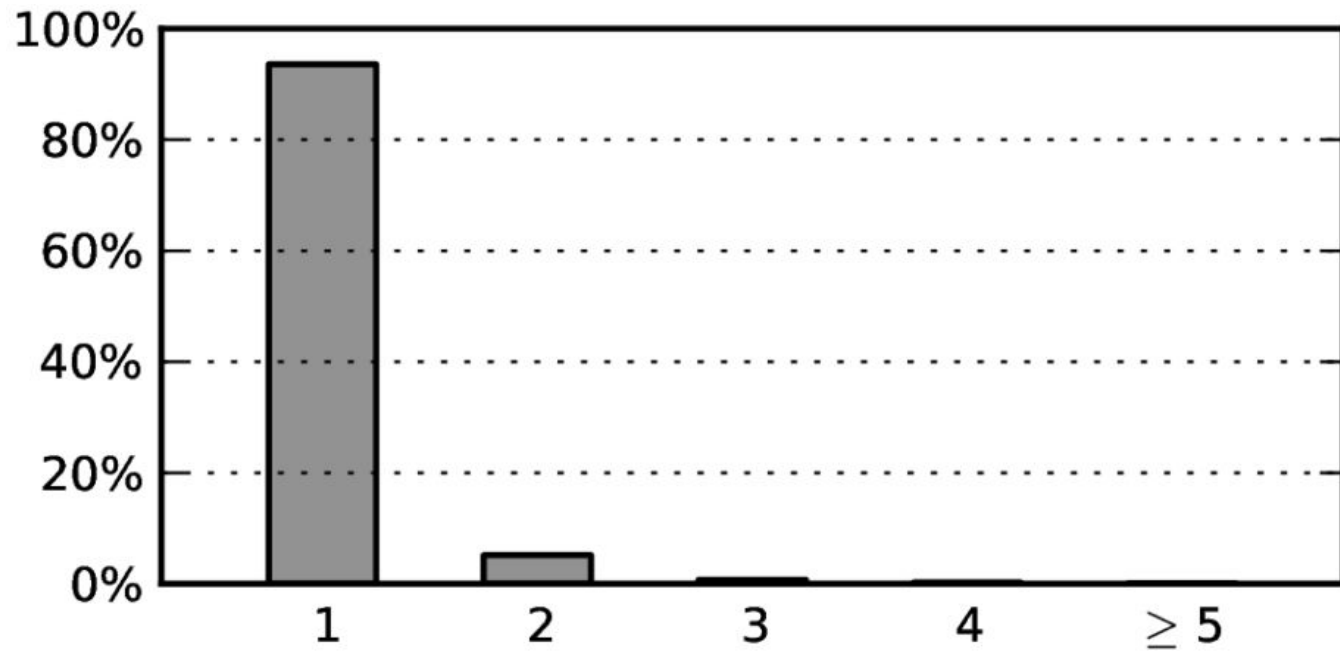




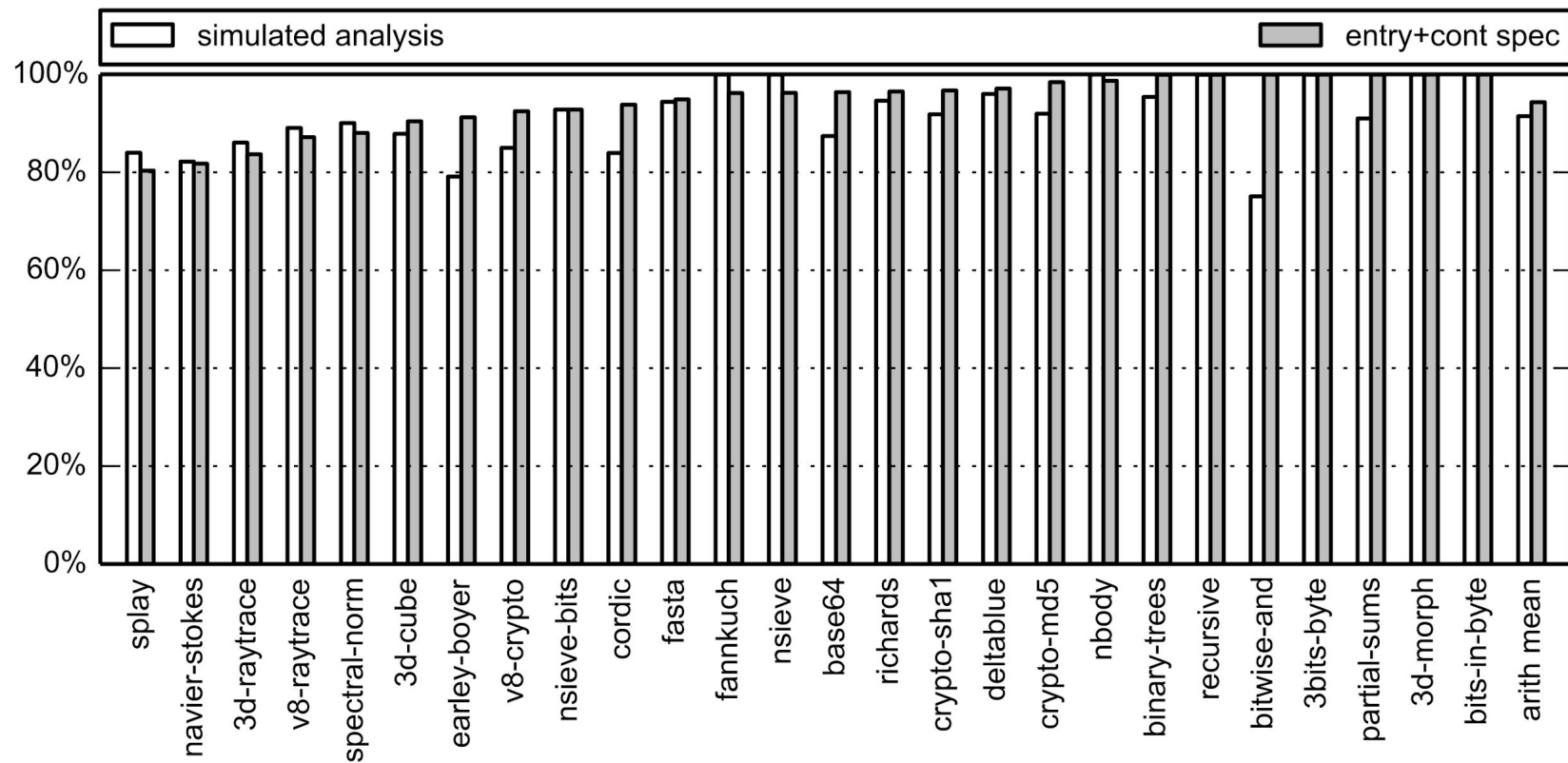








Relative occurrence of block version counts (bucket counts)



Proportion of type tests eliminated with BBV vs an idealized type analysis (higher is better)

Deferred Compilation

- In a JIT, often want to specialize code based on run-time types
 - Traditional JITs typically do this by profiling first
 - Two implementations, or two modes of compilation
- Psyco and the *unlift* operator
 - Promote run-time values into compile-time types
- We call this deferred compilation and type capture
 - Defer compilation as late as possible
 - Pause compilation until we can peek at run-time values



Representation-Based Just-In-Time Specialization and the Psyco Prototype for Python

Armin Rigo

School of Electronics and Computer Science
University of Southampton
SO17 1BJ
United Kingdom

arigo@tunes.org

ABSTRACT

A powerful application of specialization is to remove interpretative overhead: a language can be implemented with an interpreter, whose performance is then improved by specializing it for a given program source. This approach is only moderately successful with very high level languages, where the operation of each single step can be highly dependent on run-time data and context. In the present paper, the Psyco prototype for the Python language is presented. It introduces two novel techniques. The first is *just-in-time specialization*, or *specialization by need*, which introduces the “unlifting” ability for a value to be promoted from run-time to compile-time during specialization – the inverse of the lift operator of partial evaluation. Its presence gives an unusual and powerful perspective on the specialization process. The second technique is *representations*, a theory of data-oriented specialization generalizing the traditional specialization domains (i.e. the compile-time/run-time dichotomy).

designed to lead themselves naturally to more efficient execution techniques – typically static compilation – while others are not. The example on which we will focus is the Python [21] language. Python is trivially strongly typed, i.e. types are attached to the values and any variable can contain values of any type. Moreover, all operations that appear in the source code are fully polymorphic, with their semantics usually defined by the run-time types of the involved values. The operations are heavily overloaded by built-in types and can be further overloaded by user-defined types.

Python is thus essentially an interpreted language, although a number of attempts have been made to compile it, notably Python2C and its successor Pyrex¹. The problem is that merely removing the interpretative overhead (associated with decoding and dispatching individual bytecode operations) falls short of the expected major performance increase, because each operation still has to determine the types of the involved values at run-time. In particular, val-

```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```



```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

```
# getlocal_WC_0 obj@0
```

```
# getlocal_WC_0 idx@1
```

```
# opt_aref
```

```
# leave
```

```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

```
# getlocal_WC_0 obj@0
# getlocal_WC_0 idx@1
# opt_aref
# leave
```

```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

```
# getlocal_WC_0 obj@0
# getlocal_WC_0 idx@1
# opt_aref
# leave
```

```
def get(obj, idx)
  return obj[idx]
end
```

```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

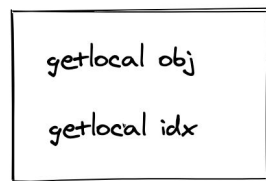
```
# getlocal_WC_0 obj@0
# getlocal_WC_0 idx@1
# opt_aref
# leave
```

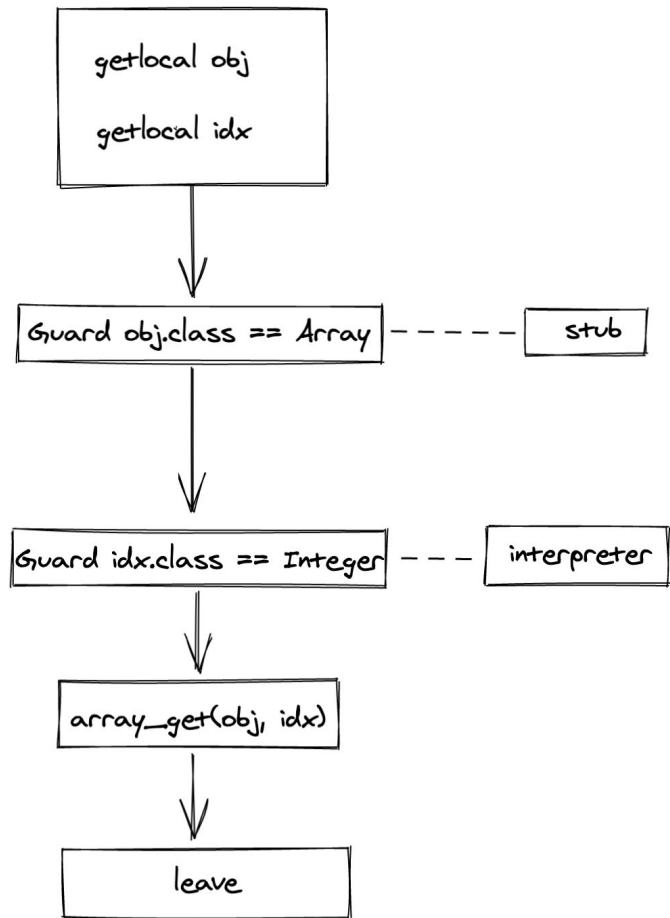
```
def get(obj, idx)
  return obj[idx]
end
```

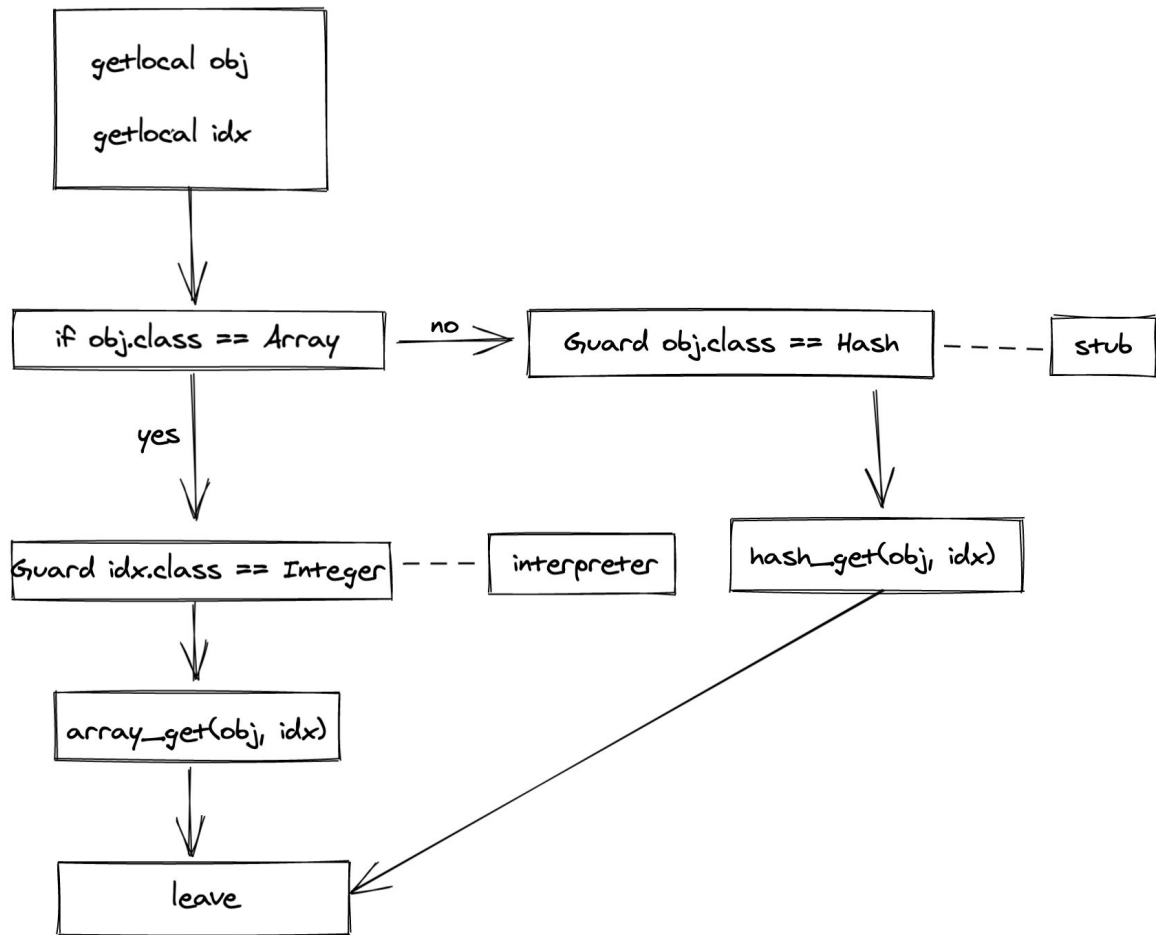
```
puts get([0, 1, 2], 1)
puts get({ a:1, b:2 }, :a)
```

```
puts RubyVM::InstructionSequence.disasm(method(:get))
```

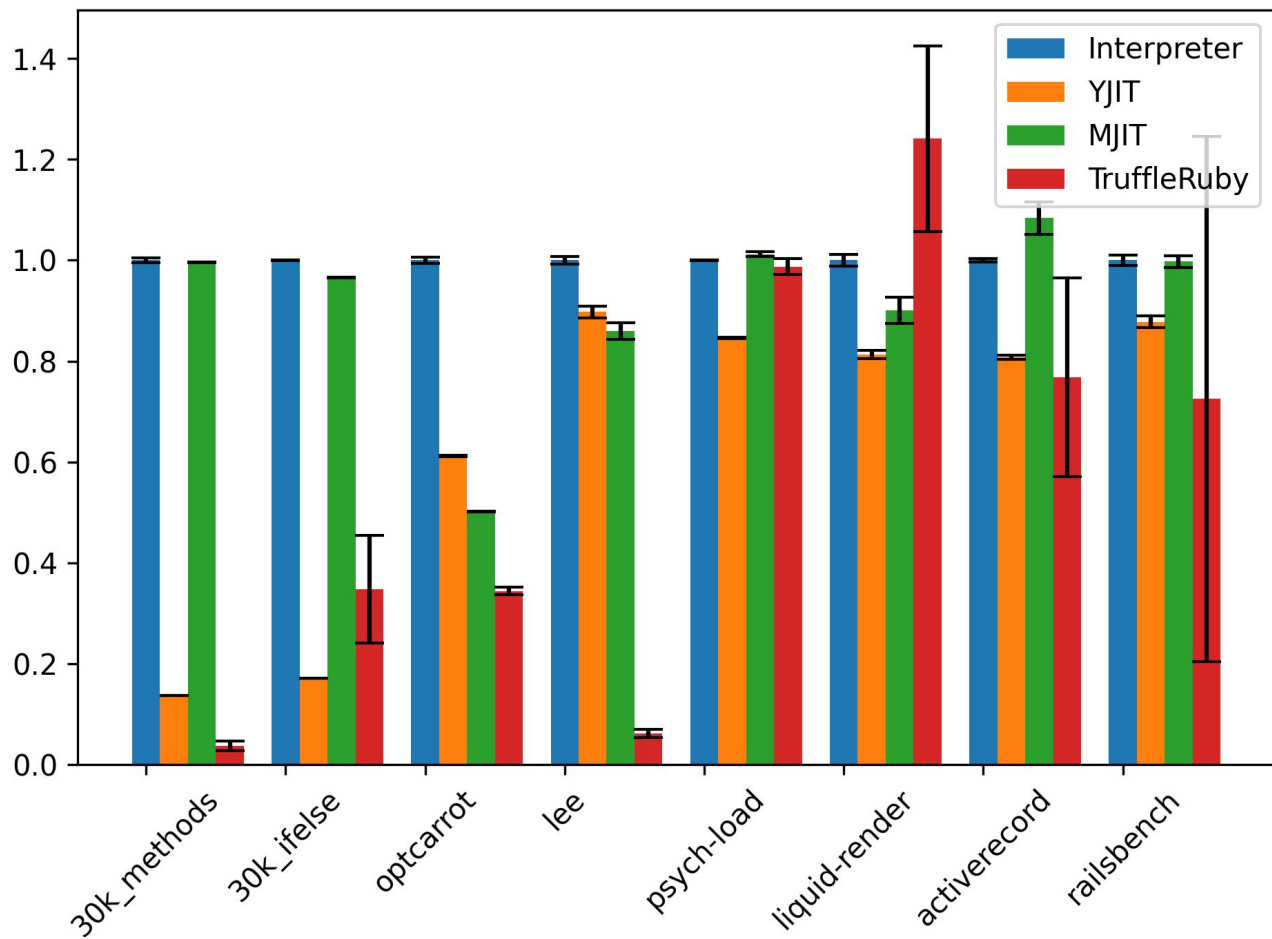
```
# getlocal_WC_0 obj@0
# getlocal_WC_0 idx@1
# opt_aref
# leave
```





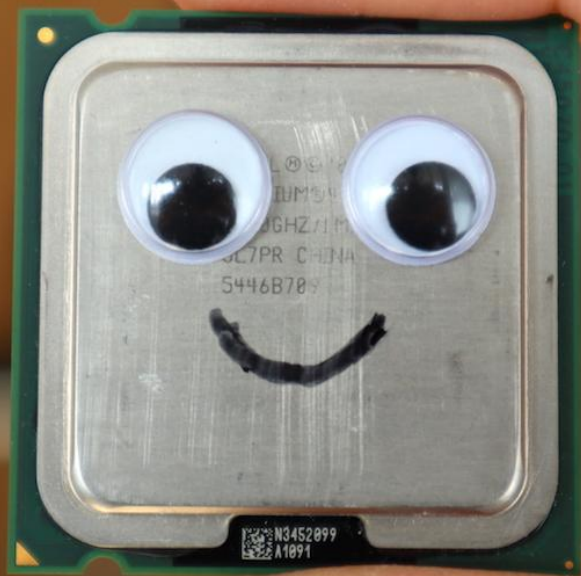


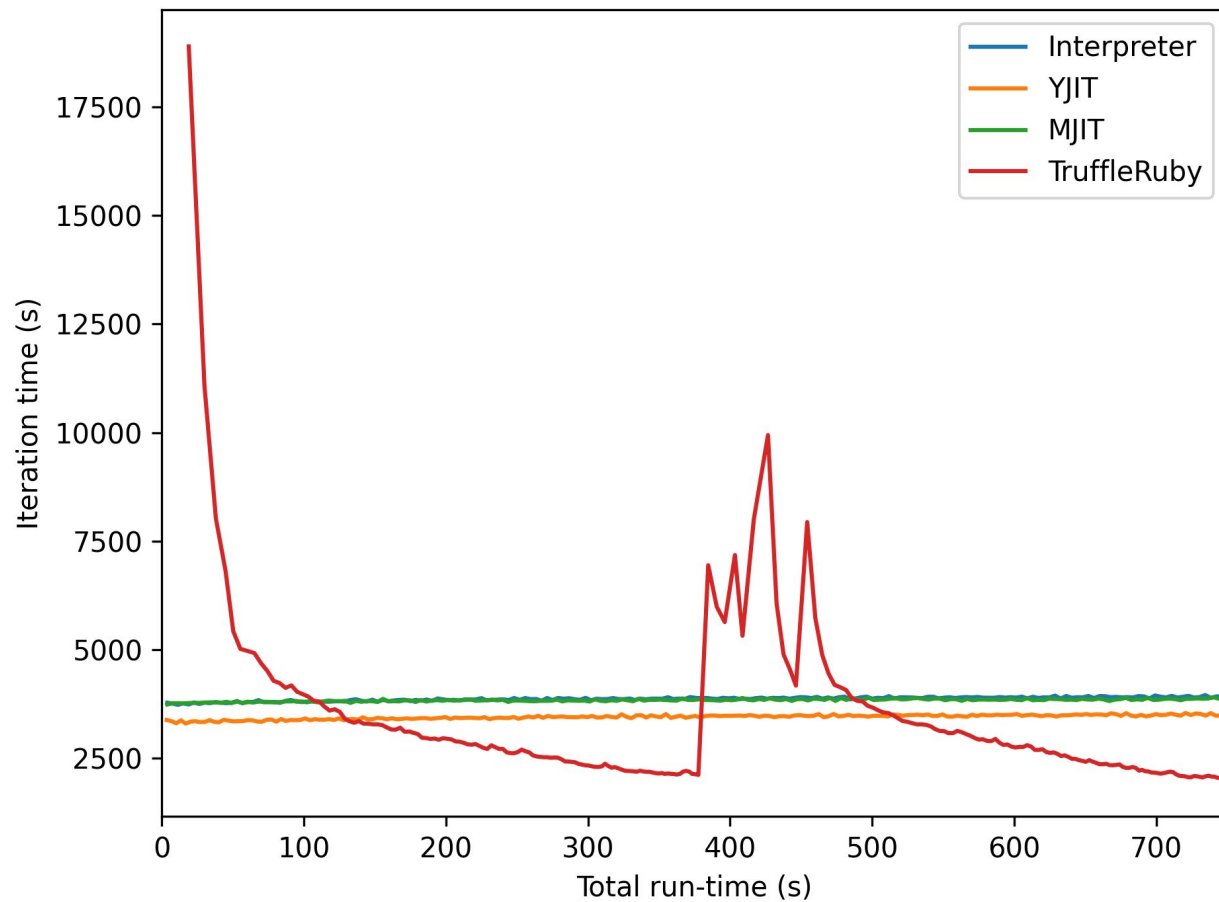
Some Early Results



Execution time relative to the interpreter (lower is better)













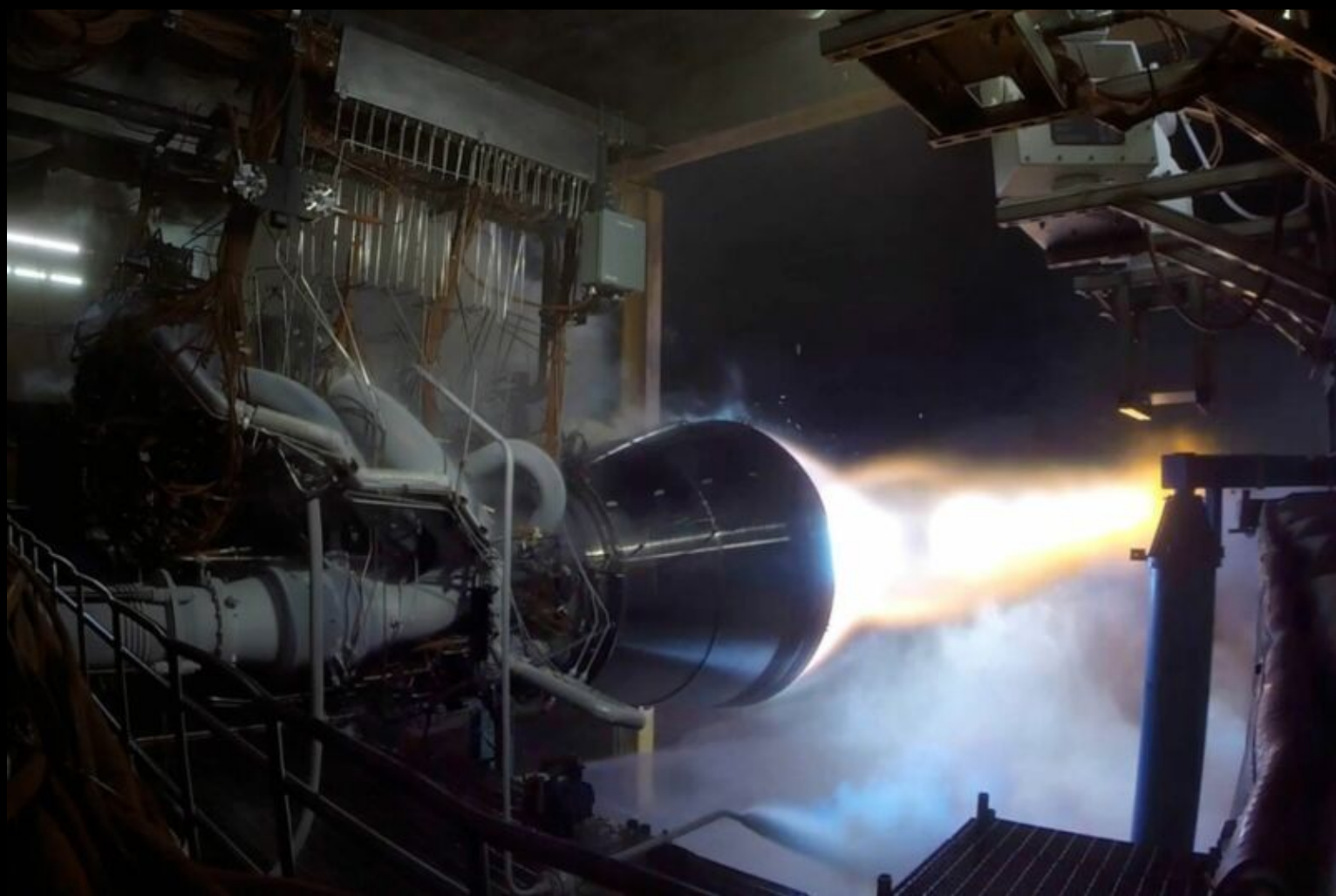
Time per iteration vs total execution time on railsbench (lower is better)

Compatibility

-  Recently added TracePoint support (thanks Alan Wu!)
-  Passing all CRuby tests (`make check`)
-  Matching Ruby head in the Shopify Core CI
-  Matching Ruby head in the GitHub backend CI
-  Matching Ruby head in the Shopify' StoreFront Renderer CI
-  Able to run Shopify's StoreFront Renderer in production



Performance Bottlenecks in CRuby







Maximizing Performance

- Race cars aren't just regular cars with more powerful engines
- They also have:
 - More aerodynamic design
 - Channel airflow to cool engine
 - Lighter body (eg: carbon fiber)
 - Racing suspension
 - Racing tires
 - High-performance brakes
 - Special tuning
- For maximum performance, the entire design is highly specialized
- Engineering always involves tradeoffs

Re-engineering CRuby

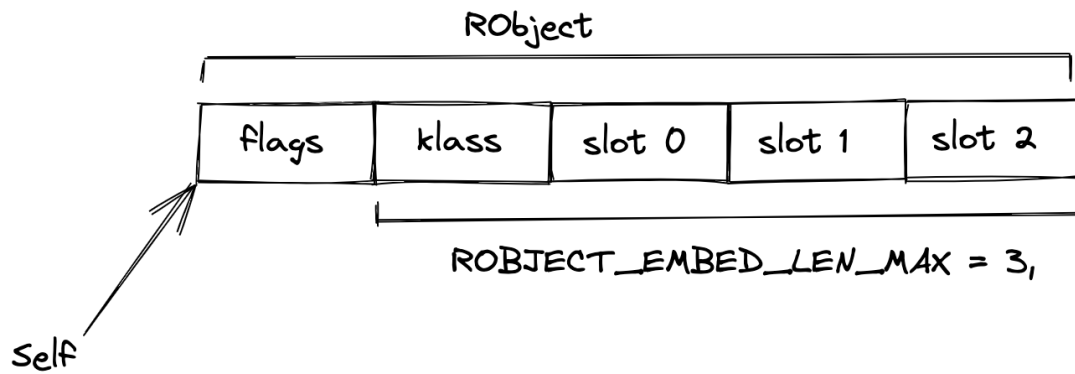
- To be fast, adding a JIT compiler to CRuby is not enough
 - CRuby was designed with an interpreter in mind
 - Design priorities for an interpreter are different
- For maximum performance, some re-engineering is necessary 🔧
- I am not suggesting that we rewrite all of CRuby :-)
 - Target areas where a JIT compiler needs to interface with CRuby
 - What are the biggest performance bottlenecks?
 - What are some things that would be easy to improve?
 - Help both YJIT and MJIT in the process

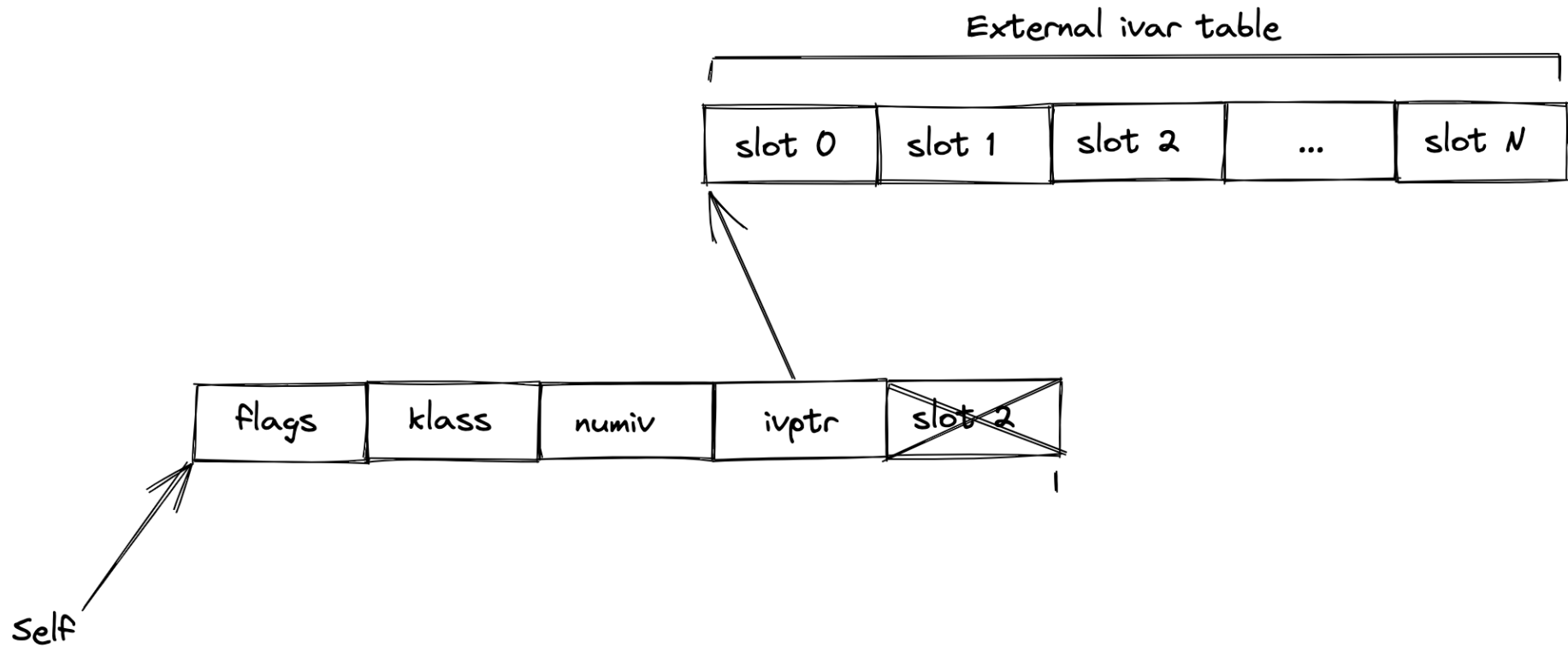


The Biggest Bottlenecks

- Ruby is object-oriented:
 - Lots of short methods with many method calls in them
 - Lots of instance variable accesses
- Much of the bulk of the machine code we generate is:
 - Method calls (send & leave)
 - Instance variable accesses
- Currently, both of these require many steps
 - Not something a JIT compiler can easily remove
 - The format of objects and stack frames is defined by the interpreter

Instance Variable Reads in YJIT





Overhead in Ruby ivar Accesses

- Almost all Ruby objects have more than 3 ivars
 - So almost all ivar reads/writes in CRuby need two levels of indirection!
 - This could be fixed with variable-width allocation
- Ruby classes don't have all the info we need
 - Is the object using an external ivar table? Check flags every time we access an ivar
 - Is the object frozen or not? Check flags every time we write an ivar
 - We could address this by using object shapes
- Two comparisons to check if something is a heap object
 - Necessary because of Ruby's tagging scheme
 - This check happens all the time
- We have the technology, we can make CRuby better





Ruby »

Ruby master

Search:

Ruby master

[Overview](#)[Activity](#)[Roadmap](#)[Issues](#)[Wiki](#)[Repository](#)

Feature #18045 CLOSED

...



Variable Width Allocation Phase II

Added by [peterzhu2118](#) (Peter Zhu) 28 days ago. Updated [about 8 hours ago](#).

Status: Closed
Priority: Normal
Assignee: [peterzhu2118](#) (Peter Zhu)
Target version: -

[\[ruby-core:104684\]](#)

Description

GitHub PR: <https://github.com/ruby/ruby/pull/4680>

Feature description

Since merging the initial implementation in [#17570](#), we've been working on improving the performance of the allocator (which was one of the major drawbacks in the initial implementation). We've chosen to return to using a freelist-based allocator and added support for variable slot size pages in what we call "size pools". We're still keeping the `USE_RVARGC` compile-time flag that maintains current behaviour when it is not explicitly enabled.

Summary

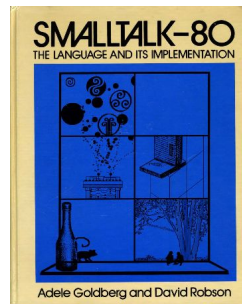
- We now use pages with different slot sizes in pools to speed up allocation. Objects will be allocated in the smallest slot size that fits the requested size. This brings us back to the freelist-based allocation algorithm and significantly increases allocation performance.

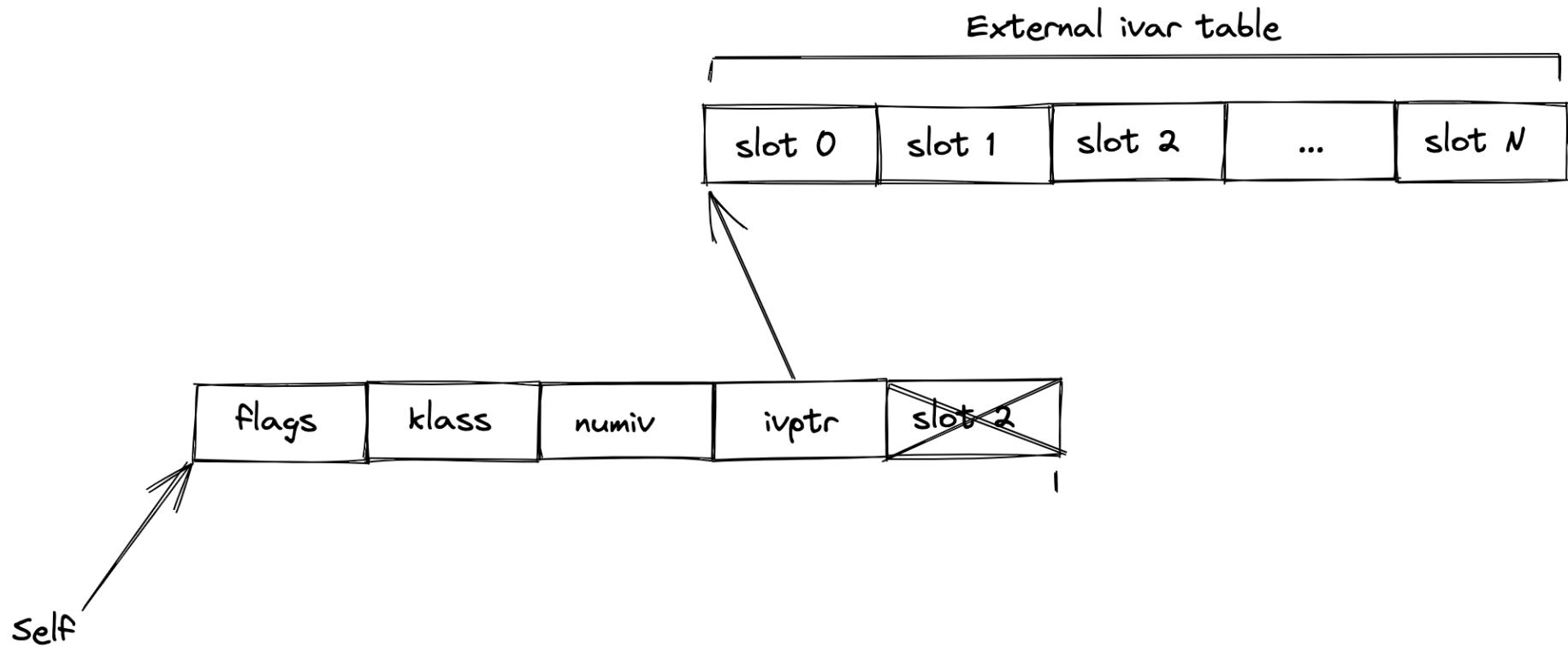
Custom queries

[Backport 2.2](#)[Backport 2.3](#)[Backport 2.4](#)[Backport 2.5](#)[Backport 2.6](#)[Backport 2.7](#)[Backport 3.0](#)[bugs: unassigned](#)[DevelopersMeeting](#)[matz](#)

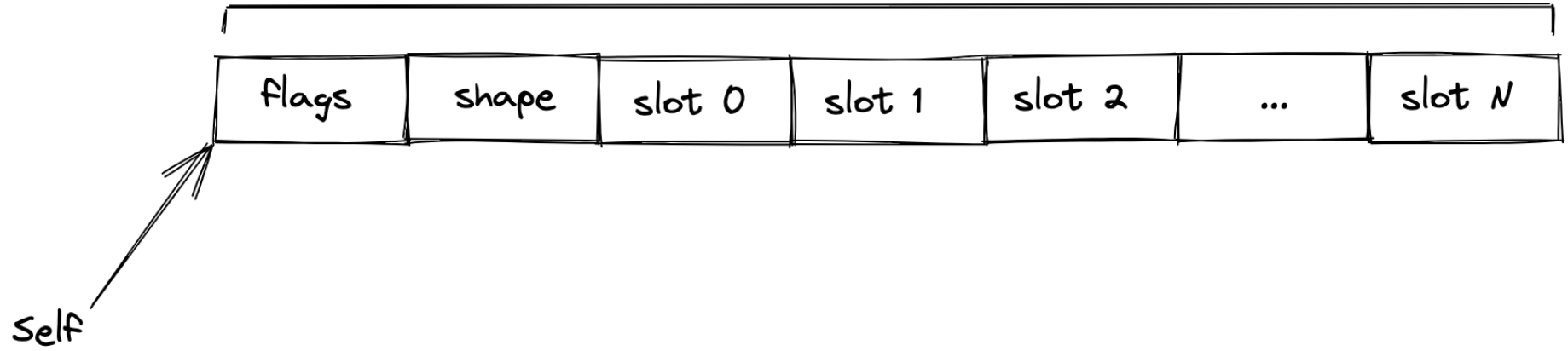
Object Shapes

- Idea dates back to Smalltalk, one of the languages that inspired Ruby
- Won't go into the details in this talk:
 - My colleague Chris Seaton gave a talk about this earlier today
 - Each object has a shape pointer
 - Shapes are metadata about an object
- Tested and proven technology, used by
 - Chrome's V8
 - Firefox's SpiderMonkey
 - WebKit's JavaScriptCore (Safari)
 - TruffleRuby
- There's no reason why it couldn't work for CRuby as well!





Variable-sized object (VWA)



```
class Foo
  def initialize
    @v0 = 1
    @v1 = 1
    @v2 = 1
    @v3 = 1
  end

  def get_v3
    @v3
  end
end

foo = Foo.new
foo.get_v3

puts YJIT.disasm(Foo.instance_method(:get_v3))
```



```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18] ; read self
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax ; push value on the stack
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```



```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known class
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], rcx
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known shape
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known shape
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known shape
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known shape
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd:  mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1:  test al, 7
1129b85e4:  jne 0x11a9b802b
1129b85ea:  cmp rax, 8
1129b85ee:  jbe 0x11a9b802b
; guard known shape
1129b85f4:  movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608:  test word ptr [rax], 0x2000
1129b860d:  jne 0x11a9b806c
1129b8613:  mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b:  cmp rax, 0x34
1129b861f:  movabs rcx, 8
1129b8629:  cmov rax, rcx
1129b862d:  mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd: mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1: test al, 7
1129b85e4: jne 0x11a9b802b
1129b85ea: cmp rax, 8
1129b85ee: jbe 0x11a9b802b
; guard known shape
1129b85f4: movabs rcx, 0x10e742138
1129b85fe: cmp qword ptr [rax + 8], 0x3973
1129b8602: jne 0x11a9b804e
; guard ivar is on extended table
1129b8608: test word ptr [rax], 0x2000
1129b860d: jne 0x11a9b806c
1129b8613: mov rax, qword ptr [rax + 0x18]
1129b8617: mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b: cmp rax, 0x34
1129b861f: movabs rcx, 8
1129b8629: cmov rax, rcx
1129b862d: mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
1129b85dd: mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1: test al, 7
1129b85e4: jne 0x11a9b802b
1129b85ea: cmp rax, 8
1129b85ee: jbe 0x11a9b802b
; guard known shape
1129b85f4: movabs rcx, 0x10e742138
1129b85fe: cmp qword ptr [rax + 8], 0x3973
1129b8602: jne 0x11a9b804e
; guard ivar is on extended table
1129b8608: test word ptr [rax], 0x2000
1129b860d: jne 0x11a9b806c
1129b8613: mov rax, qword ptr [rax + 0x18]
1129b8617: mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b: cmp rax, 0x34
1129b861f: movabs rcx, 8
1129b8629: cmov rax, rcx
1129b862d: mov qword ptr [rbx], rax
```



```
; getinstancevariable (@v3)
1129b85dd: mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1: test al, 7
1129b85e4: jne 0x11a9b802b
1129b85ea: cmp rax, 8
1129b85ee: jbe 0x11a9b802b
; guard known shape
1129b85f4: movabs rcx, 0x10e742138
1129b85fe: cmp qword ptr [rax + 8], 0x3973
1129b8602: jne 0x11a9b804e
; guard ivar is on extended table
1129b8608: test word ptr [rax], 0x2000
1129b860d: jne 0x11a9b806c
1129b8613: mov rax, qword ptr [rax + 0x18]
1129b8617: mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b: cmp rax, 0x34
1129b861f: movabs rcx, 8
1129b8629: cmov rax, rcx
1129b862d: mov qword ptr [rbx], rax
```

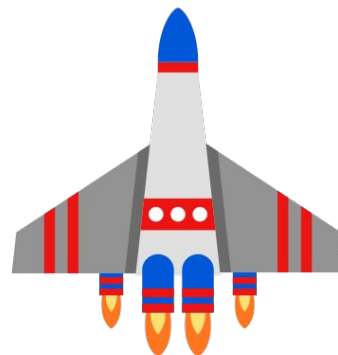
```
; getinstancevariable (@v3)
1129b85dd: mov rax, qword ptr [r13 + 0x18]
; guard self is heap
1129b85e1: test al, 7
1129b85e4: jne 0x11a9b802b
1129b85ea: cmp rax, 8
1129b85ee: jbe 0x11a9b802b
; guard known shape
1129b85f4: movabs rcx, 0x10e742138
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
; guard ivar is on extended table
1129b8608: test word ptr [rax], 0x2000
1129b860d: jne 0x11a9b806c
1129b8613: mov rax, qword ptr [rax + 0x18]
1129b8617:  mov rax, qword ptr [rax + 0x18]
; check if ivar is Qundef
1129b861b: cmp rax, 0x34
1129b861f: movabs rcx, 8
1129b8629: cmov rax, rcx
1129b862d: mov qword ptr [rbx], rax
```

```
; getinstancevariable (@v3)
; guard known shape
1129b85fe:  cmp qword ptr [rax + 8], 0x3973
1129b8602:  jne 0x11a9b804e
1129b8617:  mov rax, qword ptr [rax + 0x18]
```

Looking Forward

Future Plans

- More sophisticated backend
 - Separate type-specialization logic from codegen
 - Simple register allocator
 - Basic instruction selection
 - Eventual ARM64 support
- Bring object shapes (aka hidden classes) to CRuby
 - Current object model very complex, and not ideal for JITs
 - Could be both simpler and more efficient



Collaboration with Ruby Core Developers

- JITs, being more complex than interpreters, are harder to refactor
 - Stability is good for us
- Some things that could hurt YJIT (& also MJIT)
 - Major changes to the CRuby bytecode
 - Adding complexity and special cases
 - Changing the semantics of method calls
 - More complex instance variable lookups
- YJIT & MJIT have similar needs
 - Our plan is to upstream object shapes separately from YJIT
 - We could design a faster calling convention together
 - Also happy to discuss other improvements to CRuby



Conclusion

- Early in the YJIT project
 - Results still modest
 - Clear path to much bigger speedups
- Building this new JIT inside CRuby
 - Huge compatibility advantage
 - Already passing all the CRuby tests
- Lazy Basic Block Versioning (LBBV):
 - Lazy evaluation for code
 - Capture type information as late as possible
 - Precision advantage
 - Linearization of code



Contact & More

- To learn more about the technology:
 - *Simple and Effective Type Check Removal through Lazy Basic Block Versioning.* M Chevalier-Boisvert and M Feeley. ECOOP 2015.
 - *Interprocedural Type Specialization of JavaScript Programs Without Type Analysis.* M Chevalier-Boisvert and M Feeley. ECOOP 2016.
 - Recordings of both talks on YouTube
- If you'd like to reach out and talk about compilers
 - Ping me at maxime.chevalierboisvert@shopify.com
 - Also @Love2Code via Twitter
- Are you looking for a fun, stimulating and flexible work environment?
 - Shopify is hiring!
 - Permanently remote positions
 - Send resume to maxime.chevalierboisvert@shopify.com



YJIT is on GitHub: <https://github.com/Shopify/yjit>