Threaded Code Generation with a Meta-tracing JIT Compiler

Yusuke Izawa
izawa@prg.is.titech.ac.jp
Tokyo Institute of Technology
Japan

Hidehiko Masuhara
masuhara@acm.org
Tokyo Institute of Technology
Japan

Carl Friedrich Bolz-Tereick
cfbolz@gmx.de
Heinrich-Heine-Universität Düsseldorf
Germany

Youyou Cong
cong@c.titech.ac.jp
Tokyo Institute of Technology
Japan

ABSTRACT

Language implementation frameworks such as RPython and Truffle/Graal are effective tools for creating a high-performance language with lower effort than implementing from scratch. The two frameworks support only a single JIT compilation strategy, trace-based compilation and method-based compilation, but they have its own advantages and disadvantages. We proposed a *meta-hybrid JIT compiler framework* to take advantages of the two strategies as a language implementation framework. We also implemented a proof-of-concept framework called BacCaml.

As a next step, in this position paper, we propose a new approach to realize a method-based baseline JIT compiler along with a trace-based JIT compilation. We aim to use it for further speed-up by preventing the path-divergence problem, which causes serious slowdown. We also show how to implement the baseline JIT compiler with minimal changes on top of RPython.

KEYWORDS

JIT compiler, meta-tracing JIT compiler, RPython, threaded code

1 INTRODUCTION

As more and more applications require a high-performance runtime, the complexity of VMs is increasing. In responding to this trend, the code size of VMs becomes bigger and bigger. For example, in early 2021, the latest OpenJDK consists of over 3 million source lines of code (SLOC) only in Java, and CRuby is composed of about 2 million SLOC in Ruby and C. This means that creating a new practical and sophisticated language from scratch requires language developers is a more complicated implementation task.

Using language implementation frameworks [3, 21] is one of the important ways to the reduce engineering effort for language implementers. A language implementation framework is a toolchain to create a high-performance VM without implementing from scratch; from an interpreter definition, it generates a fast VM empowered by a JIT compiler. There are two state-of-the-art frameworks, namely RPython [3] and Truffle/Graal [21]. RPython is a restricted subset of Python. On the other hand, Truffle/Graal generates a VM on the top of GraalVM. Many VMs such as PyPy [3], GraalPython [16], Topaz [18], TruffleRuby [15], RSqueak [14], and TruffleSqueak [4] are generated by RPython and Truffle/Graal; they archived higher performance than original interpreters.

Currently, users of a language implementation framework have to choose one JIT compilation strategy; trace-based [1, 5, 9] or method-based [17, 19] strategies. It is because RPython and Truffle/Graal support only a single strategy; RPython generates trace-based, while Truffle/Graal generates a method-based JIT compiler. However, the two strategies have its own advantages and disadvantages [7, 8, 10, 12], thus language developers have to carefully choose which one is better for the language they are going to realize beforehand.

To make frameworks less constrained by a compilation strategy, we proposed a *meta-hybrid JIT compiler framework* [10]. Its hybrid JIT compiler can use both strategies in a single framework, since we constructed the two strategies by extending a (meta-) tracing JIT compiler. We found that there existed programs which ran faster by a hybrid strategy than others, but there was a room for further improvements and production-level experiments since our BacCaml framework [10] is still proof-of-concept.

In the next phase, we are going to implement our hybrid JIT approach in RPython itself to make the hybrid JIT more practical. In this position paper, as a first step for it, we propose method-based baseline JIT approach on the top of RPython [1]. A baseline JIT compiler aims to less startup time and memory footprint. The objective of creating the baseline JIT on RPython is preventing the performance degradation problem, which was reported in [7] and [8] (we call this problem *path-divergence problem* here). This problem is that paths that are rarely executed are selected for compilation. The resulting traces from less-executed paths cause many guard failures, so it leads poor performance in trace-based JIT compilation. Therefore, we aim to apply baseline JIT compilation for programs with path-diverged functions. We implement this not realize it creating a new compiler, but design an interpreter for traversing all paths of a target function and stitching a resulting trace to create a trace tree which covers the entire of a target function.

ICOOOLPS '21, Jul 13, 2021, Virtual

[1]Our proposal is work-in-progress, thus we introduce the idea and basic mechanism of the RPython baseline JIT compiler in this paper.

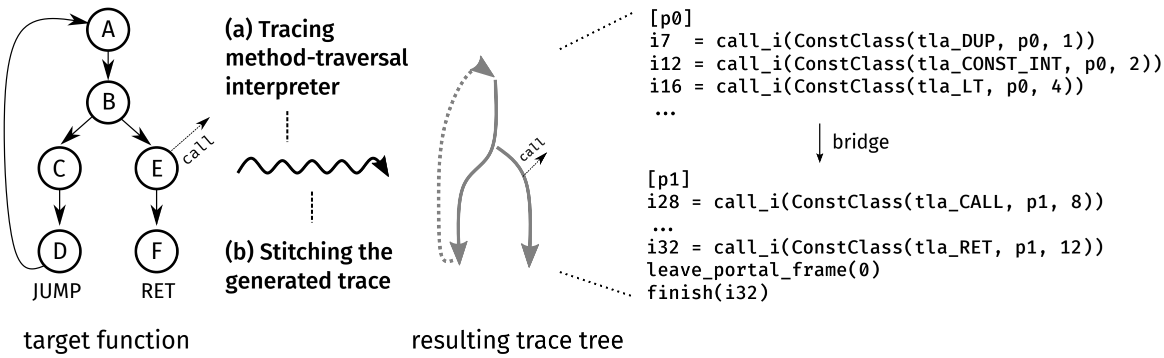

Figure 1: A sketch of how RPython method-based baseline JIT compiler works. From the target function in the left-hand side, it generates the trace tree shown in the right-hand side.

2 THE COMPILATION TACTIC OF RPYTHON'S BASELINE JIT

In this section, we present how to realize method-based baseline JIT strategy on top of RPython not by implementing from scratch.

2.1 Compilation Principle

The principle of the method-based baseline JIT is based on threaded code [2], which is a technique where the emitted code consists entirely of calls to subroutines. The objective of introducing it is for less startup and compilation time in the RPython. Our baseline JIT compilation strategy is method-based, so we compile not a linear execution path but the whole body of a target function. We apply it for reducing the occurrence of the path-divergence problem.

For less implementation effort, we don't create two compilers individually but realize the baseline JIT compiler as an extension of RPython's trace-based JIT compiler. In other words, we construct a trace tree which covers all paths of a target function. Briefly speaking, the baseline JIT traces all paths with a specially instrumented interpreter, cuts and stitches the obtained trace to make a trace tree.

Figure 1 shows a high-level example of RPython baseline JIT compiler. The left-hand side of Figure 1 represents the control flow of a target function. B – C – E is a conditional branch, D is a back-edge instruction, and F is a return. The compiler finally generates a trace tree [2], which covers a function body as shown in the right-hand side of Figure 1. In contrast to trace-based compilation, it keeps the original control flow, we can see that the bodies of subroutines are not inlined but call instructions to them are left.

To produce such a trace tree, the tracer of RPython baseline JIT has to sew and stitch generated traces. We call this behavior *trace tailoring*. Technically speaking, the compiler traces a special instrumented interpreter namely *method-traversal interpreter*. Since the obtained trace from the method-traversal interpreter ignores the original control flow, we have to restore it. To rebuild the original control flow, in the next phase, the baseline JIT compiler stitches the generated trace. We call this technique *trace stitching*. In the next sections, we will explain method-traversal interpreter and trace stitching, respectively.

[2]Each trace has a linear control flow, but they are compiled as a bridge.

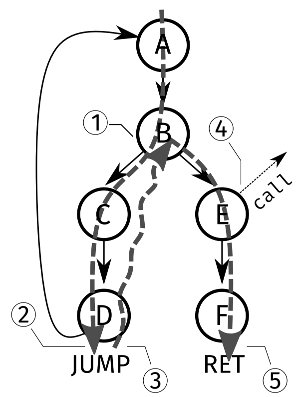

Figure 2: Tracing the entire of a function with method-traversal interpreter.

2.2 Method-traversal Interpreter

The baseline JIT is built on top of a tracing JIT, thus we have to trick RPython's tracer by instrumenting an interpreter definition to cover all paths of a target function. To enable it, we propose method-traversal interpreter. It is a special instrumented interpreter for the baseline JIT, and when applying baseline JIT the tracer follows its execution path. The method-traversal interpreter works as an abstract interpreter because it follows complete control flow graph by exploring both sides of a conditional branch.

The skeleton is shown in Figure 3. All subroutines are decorated by `dont_look_inside` hint, which tells the tracer not to trace the function body. Therefore, a resulting trace has only call instructions to subroutines.

Figure 2 shows how the baseline JIT compiler traces a function body with method-traversal interpreter. The gray-colored dotted line means a generated trace with method-traversal interpreter. Normally, tracing JIT only follows an executed side of the conditional branch. In contrast, the baseline JIT tracer follows the both sides. To enable it, method-traversal interpreter prepares a special stack data structure namely *traverse_stack*. It only stores program counters, so it is marked as *green* and finally removed from a resulting trace.

```python
@dont_look_inside
def tla_ADD(self, pc):
    x, y = self.pop(), self.pop()
    self.push(y.add(x))
    return pc

@dont_look_inside
def tla_CONST_INT(self, pc):
    arg = ord(self.bytecode[pc])
    self.push(W_IntObject(int(arg)))
    return pc + 1

def interp(self, pc, traverse_stack):
    while True:
        jit_merge_point(bytecode=self,bytecode,pc=pc,
                        self=self,
                        traverse_stack=traverse_stack)
        opcode = ord(self.bytecode[pc])
        pc += 1
        if opcode == ADD:
            pc = self.tla_ADD(pc)
        elif opcode == CONST_INT:
            pc = self.tla_CONST_INT(pc)
        elif opcode == JUMP:
            ...
        elif opcode == RET:
            ...
        elif opcode == JUMP_IF:
            ...
```

Figure 3: Skeleton of method-traversal interpreter and subroutines decorated with `dont_look_inside`.

We explain how method-traversal interpreter works by referencing examples. The difference from a normal tracing JIT is the following: (1) conditional branch, (2) back-edge instruction, (3) function call and (4) function return. We explain them as follows.

Conditional branch. Our baseline JIT tracer follows both sides of a conditional branch; firstly, tracing `then` branch, `else` branch next. Finally its control flow is restored by trace stitching (which is explained in Section 2.3).

When tracing (1) in Figure 2, it saves the other direction of a conditional branch to the traverse_stack. Figure 4a shows the implementation of `JUMP_IF` behaving as a conditional branch. You can see that `traverse_stack` saves another directions in lines 8 and 18.

Back-edge instruction. The baseline JIT tracer doesn't follow the back-edge instruction to track all the paths of a target function. When tracing (2), it doesn't follow the destination of JUMP. Next, at (3), it restores saved pc from `traverse_stack` and goes to the other branch (E in the Figure 1).

Seeing the implementation of `JUMP` in Figure 4b, before jumping to somewhere, it checks whether `traverse_stack` or not. If `traverse_stack` is empty, the baseline tracer normally executes `JUMP`. Otherwise, it restores the saved pc from `traverse_stack` and goes to that place. To tell the place of a back-edge instruction, we have to call a pseudo function `cut_here`. It is used in trace-stitching to restore the original control flow.

Function call. To reduce the compilation code size, our baseline JIT compiler doesn't inline a function call. When tracing `CALL` instruction at (4), it doesn't inline the `CALL` but emit only call instruction since subroutines are decorated with `dont_look_inside`.

```python
1  if opcode == JUMP_IF:
2       target = ord(self.bytecode[pc])
3       e = self.pop()
4       if self._is_true(e):
5           if we_are_jitted():
6               pc += 1
7               # save another direction
8               traverse_stack = t_push(pc, traverse_stack)
9           else:
10              if target < pc:
11                 can_enter_jit(bytecode=self.bytecode,pc=target,
12                               self=self,
13                               traverse_stack=traverse_stack)
14              pc = target
15      else:
16          if we_are_jitted():
17              # save another direction
18              traverse_stack = t_push(target, traverse_stack)
19          pc += 1
```

(a) Definition of JUMP_IF.

```python
@dont_look_inside
def cut_here(self, pc):
    "A pseudo function for trace stitching"
    return pc

if opcode == JUMP:
    t = ord(self.bytecode[pc])
    if we_are_jitted():
        if t_is_empty(traverse_stack):
            pc = t
        else:
            pc, traverse_stack = traverse_stack.t_pop()
            pc = cut_here(pc) # call pseudo function
    else:
       if t < pc:
           can_enter_jit(bytecode=self.bytecode,pc=t,
                         tstack=tstack,self=self)
       pc = t
```

(b) Definition of JUMP.

```python
if opcode == RET:
    if we_are_jitted():
        if t_is_empty(traverse_stack):
            return self.tla_RET(pc)
        else:
            pc, traverse_stack = traverse_stack.t_pop()
    else:
        return self.tla_RET(pc)
```

(c) Definition of RET.

Figure 4: Method-traversal interpreter definition.

Function return. When tracing `RET`, firstly, it checks whether `traverse_stack` is empty or not. If `traverse_stack` is not empty, it restores a saved pc and continues to trace. Otherwise, it executes `RET` instruction. The implementation is shown in Figure 4c, and the behavior is almost same to `JUMP`.

2.3 Trace Stitching

The obtained trace by tracing the method-traversal interpreter is a linear execution path, since the tracer was led to track all paths by the interpreter. For correct execution, we propose trace stitching, which is a technique to reconstruct the original control flow.

The left-hand side of Figure 5 shows how trace stitching works, and [1] – [5] are the working flow. Firstly, in [1], the tailor of RPython baseline JIT cuts that point. By doing this, we can handle each branch as a separate trace. At this point, the `cut_here` pseudo function is used for it. `cut_here` works as a mark for the cutting

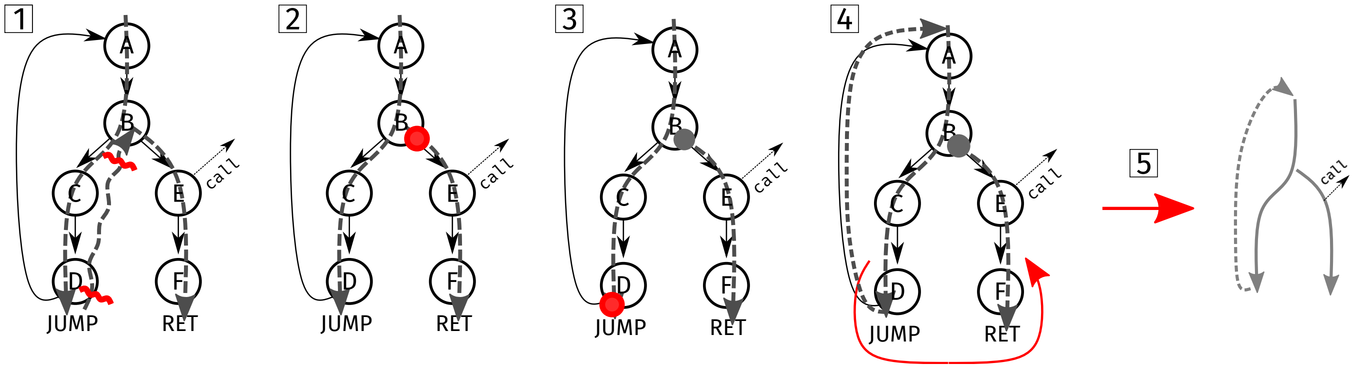

Figure 5: The working flow of trace stitching.

point and helps the tailor to find where to cut. Next, to restore the conditional branch at B, the compiler has to compile it as a bridge. In 2, the tailor generates a label L, and rewrites the destination of a guard failure in B when it fails. Then, in 3, the tailor restores JUMP instruction at the bottom of D. After that, in 4, for the correctness, it copies variables and instructions not in the scope of the branch B – E – F. Finally, in 5 the compiler removes constant or unused variables/instructions and we get the resulting trace tree shown in the right-hand side of Figure 5.

3 RELATED WORK

It is a trace-off relation between compilation time and peak performance. Server-side applications are so long-running applications that slow compilation time is acceptable. However, short-term applications like GUI programs or batch processing programs need better response time, thus a baseline JIT compiler is usually applied for such applications.

The Java HotSpot™ VM has two JIT compilers; the server compiler [17] and the client compiler [11]. The server compiler is a highly optimizing compiler and tuned to gain a much faster peak-time performance with lower compilation speed. On the other hand, the client compiler is a JIT compiler designed for low startup time and small memory footprint.

Firefox baseline compiler [20] is a warm-up compiler used in IonMonkey JavaScript JIT compiler [13]. Firefox's baseline JIT is designed to work as an intermediate layer between interpretation and highly optimizing JIT compilation. Firefox used different JIT compilers, JaegerMonkey and IonMonkey, depending on a situation, but it had several significant issues. For example, the calling convention in the two compilers are different. Moreover, JaegerMonkey itself is indeed much complex. Firefox's baseline JIT compiler is designed and created to ease such a situation. Its baseline JIT compiler is implemented simpler than other compilers but achieves from 10 to 100x faster than interpretation.

Liftoff [6] is a baseline JIT compiler for V8 and WebAssembly. V8 has already a JIT compiler namely TurboFan, but its compilation process is complicated and it consumes longer compilation time. Liftoff makes code quality secondary in order to get a faster startup time. In this way, it separates itself from the existing TurboFan compiler.

4 CONCLUSION AND FUTURE WORK

In this paper, we proposed the ideas of a threaded-code-based RPython's baseline JIT compiler and how to implement it. The essential technique is trace tailoring, which consists of the method-traversal interpreter and the trace stitching. Method-traversal interpreter is an interpreter design that tricks the trace to follow all paths of a target function. Trace stitching rebuilds a trace tree from a resulting trace generated from a method-traversal interpreter, aiming to restore the original control flow.

Currently, we designed the method-traversal interpreter on a tiny language and created a compiler that can emit a trace tree which contains only call instructions to subroutines. Our next task is implementing a JIT backend to emit machine code. After implementing the backend, we will confirm the performance of our baseline JIT. By comparing with the original tracing JIT in RPython, we will see how much startup time and memory footprint can be reduced. Finally, we will verify the effectiveness of our baseline JIT on production-level applications. Given this context, we will implement it on Python with the PyPy interpreter to run production-level benchmarks.

ACKNOWLEDGEMENT

We would like to thank the reviewers of the ICOOOLPS 2021 workshop for their valuable comments. This work was supported by JSPS KAKENHI grant number 21J10682 and JST ACT-X grant number JPMJAX2003.

REFERENCES

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.* https://doi.org/10.1145/349299.349303 arXiv:1003.4074

[2] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (June 1973), 370–372. https://doi.org/10.1145/362248.362270

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy). ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[4] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies* (Prague, Czech Republic) *(IWST '16)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/2991041.2991062

[5] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[6] Google. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. https://v8.dev/blog/liftoff

[7] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. 2011. Improving the Performance of Trace-based Systems by False Loop Filtering. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/1950365.1950412

[8] Ruochen Huang, Hidehiko Masuhara, and Tomoyuki Aotani. 2016. Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler. In *International Symposium on Trends in Functional Programming*. Springer, 44–58.

[9] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A trace-based Java JIT Compiler Retrofitted from a Method-based Compiler. *Proceedings - International Symposium on Code Generation and Optimization*, 246–256. https://doi.org/10.1109/CGO.2011.5764692

[10] Yusuke Izawa and Hidehiko Masuhara. 2020. Amalgamating Different JIT Compilations in a Meta-Tracing JIT Compiler Framework. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (Virtual, USA) *(DLS '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3426422.3426977

[11] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (May 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

[12] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA '15)*. ACM, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

[13] Mozilla. 2016. IonMonkey, the Next Generation JavaScript JIT for SpiderMonkey. https://wiki.mozilla.org/IonMonkey

[14] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR '19)*. ACM, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[15] Oracle Lab. 2013. A high performance implementation of the Ruby programming language. https://github.com/oracle/truffleruby

[16] Oracle Labs. 2018. Graal/Truffle-based implementation of Python. https://github.com/graalvm/graalpython

[17] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java Hotspot™ Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1* (Monterey, California) *(JVM '01)*. USENIX Association, USA, 1.

[18] Topaz Project. 2013. A high performance ruby, written in RPython. http://docs.topazruby.com/en/latest/

[19] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '87)*. Association for Computing Machinery, New York, NY, USA, 227–242. https://doi.org/10.1145/38765.38828

[20] Kannan Vijayan. 2013. The Baseline Compiler Has Landed. https://blog.mozilla.org/javascript/2013/04/05/the-baseline-compiler-has-landed/ Mozilla.

[21] Thomas Würthinger, Andreas Wöundefined, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587