Towards Ahead-of-Time Meta-Compilation of Dynamic Languages With an Extensible Type Analysis

Position Paper

Christoph Aigner
Johannes Kepler University
Linz, Austria
christoph.aigner@jku.at

Stefan Marr
Johannes Kepler University
Linz, Austria
stefan.marr@jku.at

Abstract

Dynamically-typed languages rely on just-in-time (JIT) compilation for execution performance. Meta-compilation systems such as GraalVM's Truffle language implementation framework have reduced the effort needed of enabling JIT compilation to implementing an interpreter. But dynamic languages are increasingly used in scenarios where ahead-of-time (AOT) compilation would be preferable, for instance, for faster startup or to avoid the memory cost of JIT compilation. Therefore, we plan to extend meta-compilation systems to also support AOT compilation.

For successful AOT compilation of dynamically-typed languages, we need an extensive and robust type analysis. In this position paper, we present first ideas for a framework with an extensible core analysis that will enable us to extract type flow semantics from an interpreter implemented in a meta-compilation system. To achieve the precision needed for fast machine code, we will need to include heuristic analyses. For this, we envision a plugin system that allows us to integrate various different heuristics into a singular unified analysis. Combining analyses in this way can produce results that are better than the sum of their parts.

While this is a very ambitious goal, given the complexity of compiling dynamic languages, we believe we can achieve better-than-interpreted performance for programs with *reasonable* behavior. Furthermore, to support the full language semantics we keep a general interpreter as a fallback.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Program analysis*; *Type structures*.

Keywords: ahead-of-time compilation, meta-compilation, static analysis, heuristics, type recovery, dynamic languages

ICOOOLPS '26, Brussels, Belgium

ACM Reference Format:
Christoph Aigner and Stefan Marr. 2026. Towards Ahead-of-Time Meta-Compilation of Dynamic Languages With an Extensible Type Analysis: Position Paper. In *Proceedings of Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '26).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Traditionally, to make execution of dynamically-typed languages such Python, JavaScript, or Ruby fast, language implementers use just-in-time (JIT) compilation. While it often gives good peak performance, the warmup cost of JIT compilation is a problem for short running scripts, unit tests, and frequently updated applications. Furthermore, JIT compilation can cause performance variance, because of its dynamic feedback-driven nature, and has a computational and memory overhead at run time. Thus, for some applications JIT compilation is not ideal. For example, applications deployed in the cloud would often benefit from fast startup, predictable performance, and lower resource use that ahead-of-time (AOT) compilation promises.

Language implementation frameworks such as GraalVM's Truffle framework [11, 26], RPython [6, 7], and Deegen [28] successfully reduced the engineering effort for implementing dynamic languages to that of building an optimizing interpreter. However, they only enable meta-compilation in conjunction with JIT compilation. There is no such approach yet for AOT meta-compilation of dynamic languages. Thus, we propose to extend the Truffle framework so that it can meta-compile programs written in dynamic languages ahead of time, by compiling through their interpreters.

To be able to meta-compile application code ahead of time, we will need a way to perform type analysis on the application code, through the interpreter. Our goal is to apply such analyses to Truffle-based interpreters with only minimal modifications and to leverage the Truffle DSL [11] to extract type flow information. In Truffle, a language implementation can provide so-called "specializations" of operations, which define an operation's semantics for instance, for a specific set of input types, and also enable self-specialization [27], which leads to faster execution. We believe, we can use these

specializations not only to extract basic types, but also for instance for method lookup results to enable type analyses.

Furthermore, as Serrano [24] demonstrates, it is possible to get good compilation results by optimistically compiling dynamic language programs with the assumption that they behave *reasonably* and mostly monomorphic. Given that JIT compilers rely on programs to be mostly monomorphic, and we also see this to be true for very dynamic languages such as Ruby [13], this seems a promising avenue.

To enable the meta-compilation itself, the goal is to leverage Truffle's partial evaluation implementation [26]. The combination of the interpreter with the concrete application code needs to be enriched by our analysis results to fully initialize the interpreter structure for partial evaluation. The resulting system would work similar to its JIT-compiling version. We, however, want to derive the necessary information entirely statically. For *reasonable* programs that is likely to give *better-than-interpreted* performance. For programs that are highly dynamic or megamorphic, we will fall back to interpretation to avoid restricting the language's semantics.

To reach a level of precision in our analysis where the compiler can produce fast code, we will include speculative heuristics. We will use a plugin system that extends the core analysis with various analyses or heuristics, some of which may need to be language-specific. This will give use a singular analysis for a language, where all plugins share a unified analysis state with the core analysis, which then should form something greater than the sum of its parts.

Language-specific analyses will need to address, for instance, module loading and the extraction of structural information from the code. Since languages such as JavaScript, Python, and Ruby define functions and classes imperatively, our analysis needs to account for the language-specific mechanisms and extract class and type information despite the presence of possible conditional definitions.

In the remainder of this paper, we discuss existing approaches to meta-compilation, compiling dynamic languages, and we outline our proposal for an extensible core analysis that can integrate a wide range of heuristics. We believe these aspects to be the most challenging of this project. We also briefly outline our general research plan.

While we believe this project to be highly ambitious, our approach could significantly lower the effort needed to AOT-compile dynamically-typed languages by utilizing an existing meta-compilation system. At the same time, we are aware of the complexity of dynamic languages and that larger applications tend to use many of the capabilities afforded to them. Though, we believe that it is still realistic to achieve *better-than-interpreted* performance.

2 Existing Meta-Compilation and Dynamic Language Compilation Approaches

Meta-compilation of dynamic languages has become practical thanks to meta-tracing [6, 7] and the combination of self-specialization with partial evaluation [26, 27]. These techniques enable JIT compilation of programs running on interpreters. With run-time feedback, the compilation can optimize the specifically executing program, instead of having to optimize all theoretically possible language behaviors, and reach peak performance competitive with the custom virtual machines. Though, because these approaches use JIT meta-compilation, they can take a substantial amount of time to reach peak performance [14, 16].

An alternative approach is proposed by Xu and Kjolstad [28]. Instead of combining meta-compilation with JIT compilation, they generate a language-specific JIT compiler that goes from a language-specific bytecode to native code at run time. Though, even if this approach gives better warmup behavior, it still has the JIT-compilation overhead when compared to classic AOT-compiled languages.

However, AOT compiling dynamic languages is a major challenge. The most successful approach might be Hopc [24]. Hopc is a classic compiler for ECMAScript instead of being a meta-compiler for an ECMAScript interpreter. It combines a data-flow type analysis, a speculative analysis based on deriving type hints from usage patterns, and a range analysis, to create a speculatively optimized version of a method for good performance. It also compiles a fallback version that ensure that all ECMAScript features are supported.

Most other approaches restrict the language to some subset to enable or simplify compilation [3, 8, 18, 19, 23, 25]. This can simplify type inference for instance by limiting the dynamicity to get stronger guarantees for static analyses.

Other approaches try to achieve results similar to AOT compilation by reusing the results of JIT compilation [17, 21]. While this avoids the need to restrict language semantics and benefits from run-time feedback, it comes with the additional complexity of sharing a database of compiled code and ensuring it is valid for use with the current program.

A standard way to analyze programs is abstract interpretation [20]. In its most general form, one defines a partially-ordered set of sets of constraints, through which a trace of finite length is calculated. The last set of constraints in such a trace is then the result of this analysis. Abstract interpretation has been used for example for static type recovery [12]. In the following section, we will sketch how we hope to use abstract interpretation for ahead-of-time meta-compilation of dynamic language programs.

3 An Extensible Core Analysis

Click and Cooper [9] showed that combining multiple analyses into a single unified one can yield an analysis that is greater than the sum of its parts. Another example for this

is the work of Allen et al. [2], combining type analysis and points to analysis. Similarly, Hopc demonstrated that type analysis combined with optimistic heuristics can successfully AOT-compile ECMAScript [24].

We will combine these ideas to recover types for dynamic languages, but instead of combining multiple rigorous analyses, we will combine one rigorous core analysis with multiple heuristic analyses. This core will be formed by an abstract interpreter that derives its type flow semantics from specializations of operations in a language implementation. Obtaining the types for specializations is interesting, since they provide optimized code for the corresponding semantics.

Abstract interpretation drives the evaluation of the program. It queries the available heuristics if it cannot obtain a sufficiently small set of possible types for a given value. The results of these oracle-style heuristic analysis plugins are then included into the analysis state. Additionally, they are marked as speculative results, so that a type check is inserted when generating the machine code. If this check fails, execution is transferred to the general interpreter.

Listing 1. Simple additions in Python.

```
def doAdd(a, b):
  return a + b
acc = doAdd(1, 2) + unknown
print(acc + 4)
```

To illustrate this approach, let us consider the Python example in Listing 1. Our analysis would traverse this snippet in order of execution flow. We start at the function definition, which informs the analysis that there exists a call target with the name `doAdd`. Then we reach the call of this function with two integers as parameters, proceeding with analyzing `doAdd` under the assumption of two integer parameters. When looking up available specializations for the addition, we find that given integer inputs, it yields an integer result. Therefore, we can conclude that, given integers as parameters, the `doAdd` function again returns an integer. To maximize precision in cases, where the `doAdd` function is called with different types as inputs, we intend on building upon the Cartesian Product Algorithm [1] or its derivatives.

After this call, the analysis hits an addition with an unknown value. Usually rigorous analysis would terminate at this point, since we can not determine a concrete type for the result of the addition. In our case though, we intend on querying heuristic analyses to obtain a type for the unknown value. Since this value is used in an addition with an integer, a general heuristic might conclude that this unknown value is probably also an integer. The abstract interpreter then takes this result, marks it as speculative in the analysis state and is able to proceed, finding `acc` and the parameter passed to the `print` function to be of type integer. Crucially, since the analysis after the heuristics query is done rigorously, no run time type check is needed except for the one ensuring that the unknown value is an integer. Combining these analyses into a singular entity yields a better analysis result than executing them separately.

Furthermore, since we do meta-compilation, we do not analyze a program at the level of the source language directly. Instead, we combine language-agnostic and language-specific analyses that analyze an AST or bytecode through the language's interpreter. For this and the different heuristics, we aim to build a system that allows for easy implementation of additional heuristics, enabling for seamless integration into the larger unified analysis.

A further challenge of dynamic languages is that they can load code, define functions, classes, or types imperatively at any point during the execution of the program. Thus, our abstract interpreter needs to take this into account and perform for instance code loading. Furthermore, since the constructs, scopes, and general concepts are language specific, our abstract interpreter needs to collect the information and structural details of these constructs in an abstract form. Though, fortunately, from our experience implementing dynamic languages in the Truffle framework, they tend to map to similar enough abstractions for a relevant set of languages to allow for a language-agnostic abstract interpretation.

4 The Heuristics Playground

An integral part of this project will be to design an extensible system to enhance the analysis. With it, we can implement various different approaches to heuristic analysis and integrate them into one system, allowing multiple analyses to benefit from one another, as well as furthering the reach of the rigorous core analysis.

These plugin heuristics will not only provide probabilities for types, but also type assertions. This way, the plugin system can also be used to add rigorous analyses to improve the compilation of our dynamic languages. Again, we hope that integrating different approaches into our analysis will yield a result that is greater than the sum of its parts [2, 9, 24].

4.1 Interface Based Heuristic

One language independent but highly effective way to reconstruct missing types is an interface-based heuristic as described by Pluquet et al. [22]. The assumption this analysis is based on is, that the code to compile is correct, i.e., it does not cause a type error, and that all property accesses are valid. First, we need to collect a set of available types and their static interfaces. This can be done alongside the analysis that collects the structural details discussed in Section 3. Then we try to match the set of accessed properties for a given variable to the static interfaces collected so far.

Using the resulting set of possible types, we can further restrict the result of the rigorous analysis. The integration of this analysis in a larger framework also benefits the analysis,

because it could limit the static types to check against using a preliminary result from the core analysis. Integrating an approach approximate interpretation [15] to obtain a solid call-graph or do advanced points-to analysis could further improve this analysis.

4.2 Machine Learning Based Heuristic

One major advantage that JIT compilation has over AOT compilation is, that at run time, the compiler has access to profiling data collected throughout execution to guide optimization. Developers of AOT compilers try to have similar information by integrating a profile run of the program into the compilation pipeline. When running this instrumented build of the target program, profiling data is collected, that is then used to optimize the code further. However, this significantly complicates the build process and also requires the developer to have workloads for the profiling run that are representative of the real world workload.

To avoid this complexity, we will investigate using profile-guided optimization (PGO) for our AOT meta-compilation as a stepping stone to develop machine-learning-guided optimizations beyond type inference. We could then replace profiling data by a specially trained ML model, similar to the work of Cugurovic et al. [10]. This would allow us to apply ML-guided optimizations to our dynamic languages.

4.3 Reasoning Across Opaque Calls

A common problem in languages such as Python and Ruby is that many applications use language extensions implemented in for instance C. With our plugin system, we could support plugins that also provide information about such extensions, if necessary perhaps even specialized to a specific extension. Such plugins would give us the benefits utilized for instance by cross-module quickening [4] or proposed for PyPy [5]. This would allow us to either extract type information perhaps from debug symbols or provided by the developers of native extensions to feed additional information to our analysis to help reason across this boundary.

This approach is not necessarily limited to native calls, but also applicable to any library implemented in the target language. Our analysis could significantly cut back on execution time if library calls were pruned out of its scope, making the call opaque to the framework similar to native calls, and instead replaced by type guarantees of an annotation plugin for the given library version.

5 Research Plan

As as first step towards realizing this full analysis environment we propose developing the purely rigorous core analysis in the context of a meta-compilation system such as GraalVM's Truffle framework. To achieve executable results, this will be done using an interpreter for a known-to-be AOT-compilable language such as WebAssembly. Throughout this work we will develop a way to automatically extract type flow semantics from an interpreter implementation that runs the abstract interpretation on a concrete program. Finally, we will need to initialize the Truffle interpreter based on the analysis results to enable partial evaluation and compilation of the target program so that we can compile to native code using the existing Truffle system.

Having figured out the core of our analysis, we intend on implementing basic heuristics to try and compile dynamic languages. For this, we intend to tackle compiling Python ahead of time again staying in the context of meta-compilation systems. A key mechanism will be the ability to fall back to a general interpreter for the case of missing or incorrect type information. This fallback will also be crucial to be able to execute any code written in a dynamically-typed language, since we do not expect to be able to fully AOT-compile all programs written in a given language.

Finally, we intend to explore the proposed plugin system and experiment with specialized heuristics to improve performance and increase the part of the program we are able to compile ahead of time.

At this point, our performance aims are conservative and we hope to reach *better-than-interpreted* performance. To achieve this, we focus on compiling a subset of the target language consisting of programs with *reasonable* behavior and make these fast in an initial step. Our main objective is to drop the expensive overhead of warmup in JIT-based systems, while retaining as much performance as possible. We believe that reaching this conservative goal would be beneficial since it could reduce the needed computational resources, for instance saving memory compared to a JIT compiler, while possibly having faster start up.

6 Conclusion

Languages such as Python and JavaScript are getting more and more used in environments where fast startup and low resource use are key, for example in the cloud. Therefore, we argue that AOT compilation of dynamic languages is becoming more desirable as an alternative to JIT compilation.

Our goal is to adapt Truffle's language implementation framework to support ahead-of-time meta-compilation for dynamic languages. This will lower the engineering barrier for language implementers to provide AOT compilation for dynamic languages. We believe this is achievable by extending the Truffle framework with support for abstract interpretation and a plugin-based analysis framework that allows us to combine a rigorous analysis with various optimistic heuristics and other analyses for instance for calls into native extensions. With this extensible system, we believe, that we can make the benefits of AOT compilation available for dynamic languages, which so far rely mainly on JIT compilation for performance.

References

[1] Ole Agesen. 1995. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European conference on object-oriented programming*. Springer, 2–26.

[2] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. 2015. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, Anders Møller and Mayur Naik (Eds.). ACM, 13–18. doi:10.1145/2771284.2771287

[3] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. ACM, 105–116. doi:10.1145/3357390.3361032

[4] Felix Berlakovich and Stefan Brunthaler. 2024. Cross Module Quickening - The Curious Case of C Extensions. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, Vienna, Austria, September 16-20, 2024 (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:29. doi:10.4230/LIPICS.ECOOP.2024.6

[5] Maxwell Bernstein and Carl Friedrich Bolz-Tereick. 2024. Dr Wenowdis: Specializing Dynamic Language C Extensions using Type Information. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (Copenhagen, Denmark) *(SOAP 2024)*. ACM, 1–8. doi:10.1145/3652588.3663316

[6] 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) *(ICOOOLPS '09)*. ACM, 18–25. doi:10.1145/1565824.1565827

[7] Carl Friedrich Bolz and Laurence Tratt. 2013. The Impact of Meta-Tracing on VM Design and Implementation. *Science of Computer Programming* (2013). doi:10.1016/j.scico.2013.02.001

[8] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, 410–429. doi:10.1145/2983990.2984017

[9] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (1995), 181–196. doi:10.1145/201059.201061

[10] Milan Cugurovic, Milena Vujosevic-Janicic, Vojin Jovanovic, and Thomas Würthinger. 2024. GraalSP: Polyglot, efficient, and robust machine learning-based static profiler. *J. Syst. Softw.* 213 (2024), 112058. doi:10.1016/J.JSS.2024.112058

[11] Christian Humer. 2016. *Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters.* Technical Report. Johannes Kepler University Linz.

[12] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[13] Sophie Kaleba, Octave Larose, Richard Jones, and Stefan Marr. 2022. Who You Gonna Call: Analyzing the Run-time Call-Site Behavior of Ruby Applications. In *Proceedings of the 18th Symposium on Dynamic Languages* (Auckland, New Zealand) *(DLS'22)*. ACM, 14. doi:10.1145/3563834.3567538

[14] Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. 2023. AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 233 (Oct. 2023), 29 pages. doi:10.1145/3622808

[15] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1165–1188. doi:10.1145/3656424

[16] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '15)*. ACM, 821–839. doi:10.1145/2814270.2814275

[17] Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. 2023. Reusing Just-in-Time Compiled Code. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 263 (Oct. 2023), 22 pages. doi:10.1145/3622839

[18] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2023. An Executable Semantics for Faster Development of Optimizing Python Compilers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering* (Cascais, Portugal) *(SLE 2023)*. ACM, 15–28. doi:10.1145/3623476.3623529

[19] Fumika Mochizuki, Tetsuro Yamazaki, and Shigeru Chiba. 2025. BlueScript: A Disaggregated Virtual Machine for Microcontrollers. *The Art, Science, and Engineering of Programming* 10, 21 (15 Oct. 2025), 27. doi:10.22152/programming-journal.org/2025/10/21

[20] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. doi:10.1007/978-3-662-03811-6

[21] Andrej Pečimúth, David Leopoldseder, and Petr Tůma. 2024. An Analysis of Compiled Code Reusability in Dynamic Compilation. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (Pasadena, CA, USA) *(VMIL '24)*. ACM, 43–53. doi:10.1145/3689490.3690406

[22] Frédéric Pluquet, Antoine Marot, and Roel Wuyts. 2009. Fast type reconstruction for dynamically typed programming languages. In *Proceedings of the 5th Symposium on Dynamic Languages, DLS 2009, October 26, 2010, Orlando, Florida, USA*, James Noble (Ed.). ACM, 69–78. doi:10.1145/1640134.1640145

[23] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. 2013. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 217–232. doi:10.1145/2509136.2509536

[24] Manuel Serrano. 2018. JavaScript AOT compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*, Tim Felgentreff (Ed.). ACM, 50–63. doi:10.1145/3276945.3276950

[25] Ariya Shajii, Gabriel Ramirez, Haris Smajlović, Jessica Ray, Bonnie Berger, Saman Amarasinghe, and Ibrahim Numanagić. 2023. Codon: A Compiler for High-Performance Pythonic Applications and DSLs. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction* (Montréal, QC, Canada) *(CC 2023)*. ACM, 191–202. doi:10.1145/3578360.3580275

[26] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI'17)*. ACM, 662–676. doi:10.1145/3062341.3062381

[27] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium* (Tucson, Arizona, USA) *(DLS'12)*. 73–82. doi:10.1145/2384577.2384587

[28] Haoran Xu and Fredrik Kjolstad. 2026. Deegen: A JIT-Capable VM Generator for Dynamic Languages. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1, Article 138 (April 2026), 29 pages. doi:10.1145/3798246