Optimizing Communicating Event-Loop Languages with Truffle

[Work In Progress Paper]

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

Hanspeter Mössenböck
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Communicating Event-Loop Languages similar to E and AmbientTalk are recently gaining more traction as a subset of actor languages. With the rise of JavaScript, E's notion of *vats* and non-blocking communication based on promises entered the mainstream. For implementations, the combination of dynamic typing, asynchronous message sending, and promise resolution pose new optimization challenges.

This paper discusses these challenges and presents initial experiments for a Newspeak implementation based on the Truffle framework. Our implementation is on average 1.65x slower than Java on a set of 14 benchmarks. Initial optimizations improve the performance of asynchronous messages and reduce the cost of encapsulation on microbenchmarks by about 2x. Parallel actor benchmarks further show that the system scales based on the workload characteristics. Thus, we conclude that Truffle is a promising platform also for communicating event-loop languages.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Concurrent programming structures; D.3.4 [**Processors**]: Optimization

Keywords

Actors, Event-Loops, Concurrency, Optimization, Truffle

1. INTRODUCTION

Communicating event-loop languages such as E [14] and AmbientTalk [16] are appealing for application development because they are free from low-level data races and deadlocks, and their concurrency model is comparably simple. Communicating event loops (CEL) are isolated from each other so that any form of *communication* needs to be done explicitly via message passing instead of implicitly via shared state. This prevents low-level data races and thereby raises the abstraction level of programs. In E, these CELs are called *vats* and contain heaps of objects. In this paper, we simply call them *actors*. As a consequence of the deadlock-free design, synchronization is modeled with asynchronous messages and promises. Consequently, data and synchronization dependencies become explicit to the programmers, which can avoid hidden race conditions and deadlocks.

Recently, the CEL model got adopted by languages used predominantly in shared memory settings. JavaScript [5] add a variation of this model with Web Workers[1] and its support for promises. With this extension, many languages targeting JavaScript VMs as execution platforms also adopt the model, e. g., Dart,[2] ClojureScript,[3] Scala.js,[4] and TypeScript.[5] This trend brings the CEL model from a distributed setting into the realm of shared memory multicore systems ranging from mobile devices to server applications. Hence, an efficient utilization of multicore processors becomes more relevant than distributed messaging performance. For language implementers, this brings new optimization challenges since JavaScript—as language and platform—is dynamically typed. To our knowledge, existing research on the performance of communicating event-loop languages either restricted itself to simple interpreters as was the case for E and AmbientTalk, or used static type systems, e. g., JCoBox [15] used one to optimize message sends. Similarly, languages or frameworks that follow more closely the Hewitt [6] and Agha [1] style of actors focused on type systems to improve performance, e. g., SALSA [4] or Pony.[6]

This work is an initial exploration of the optimization challenges for dynamically-typed communicating event-loop languages for shared-memory multicore systems. We present SOM_{NS}, an implementation of Newspeak[7] [2] based on the Truffle framework, which executes on top of a JVM. Beside a brief sketch of Newspeak's concurrency model, we discuss the optimization challenges for asynchronous execution, ensuring isolation between actors, and promise resolution. We evaluate initial strategies to optimize asynchronous message sends and argument handling to ensure isolation.

*Stefan Marr's research is funded by Oracle Labs.

Submitted as Work-In-Progress Paper to the AGERE'15 Workshop co-located with SPLASH'15, Pittsburgh, PA, USA.

[1] *Web Workers*, W3C, May 2012 `www.w3.org/TR/workers/`
[2] *Dart*, Google `dartlang.org`
[3] *ClojureScript*, `github.com/clojure/clojurescript/`
[4] *Scala.js*, Sébastien Doeraene `scala-js.org`
[5] *TypeScript*, Microsoft `typescriptlang.org`
[6] *Pony*, `ponylang.org`
[7] *Newspeak Programming Language Specification*, Gilad Bracha, ver. 0.095 `http://bracha.org/newspeak-spec.pdf`

2. NEWSPEAK: A DYNAMIC CONCURRENT EVENT-LOOP LANGUAGE

For our exploration we chose Newspeak, a dynamically typed class-based language with actors based on E's communicating event-loop model. For concurrency research, it has the major advantage that it does not have a notion of global or static state. Instead, state has to be passed explicitly following the object capability model [3]. The resulting language is simple and self-consistent, which avoids many special cases while retaining the convenience of classes. Nonetheless, it is similar to widely used object-oriented languages, and thus, represents a wide range of languages used on the Web. Furthermore, Newspeak also has JavaScript and Dart backends to run in browsers and on servers. Our implementation, SOM_{NS}[8] is designed for research on shared-memory concurrency with good performance. For instance, in addition to the CEL model, SOM_{NS} implements Newspeak's `Value` objects, which are deeply immutable and thus can only refer to deeply immutable objects themselves. Hence, it is safe to share such *values* between actors and to allow them to access value objects synchronously.

A Self-Optimizing Truffle Interpreter. To achieve performance within small factors of highly optimizing VMs, we built on Truffle [12, 18]. This means, SOM_{NS} is an abstract-syntax-tree (AST) interpreter running on top of a Java Virtual Machine (JVM) with the Graal just-in-time compiler [17]. Truffle and Graal together enables meta-compilation based on self-optimizing ASTs that rewrite themselves at run time to optimize for the characteristics of the executed program and its data. Once a SOM_{NS} method has been executed often enough, Graal generates native code for it by taking the AST and applying partial evaluation, aggressive inlining, as well as classic compiler optimizations. The end result is the compilation of a SOM_{NS} method to native code. With this meta-compilation approach, the compiler is independent from the implemented language, and has been used for example for JavaScript, Python, R, and Ruby. The SOM_{NS} interpreter uses self-optimization to determine types of operations, to optimize the object layout based on types, for polymorphic inline caches [9] to cache method lookups, and other performance relevant issues of dynamic languages. Details are discussed in previous work on SOM [12, 13].

3. OPTIMIZATION CHALLENGES

The first major hurdle for performance is SOM_{NS}' dynamically typed nature, which it shares with JavaScript, Python, Smalltalk, and others. However, the ideas of speculative optimizations and adaptive compilation [8] have been successfully applied to eliminate the cost of dynamic typing and late binding. At times, the results even outperform statically compiled code, because speculative optimizations can make optimistic assumptions leading to faster native code.

For this work however, the main focus is on optimization challenges for event loop languages. Thus, we investigate the performance challenges revolving around asynchronous message sends, ensuring isolation between actors, and the efficient handling of promise resolution.

[8] SOM_{NS} is a derivate of SOM (Simple Object Machine), a family of interpreter implementations: `som-st.github.io`

Asynchronous Execution. Newspeak has four ways to initiate an asynchronous execution. We distinguish between sending of asynchronous messages to (i) far references, (ii) near references, and (iii) promises. Furthermore, we also consider the execution of the (iv) success or failure handlers, i. e., callbacks, registered on promises.

For the sending of asynchronous messages, one challenge is to determine the method to be executed on the receiver side efficiently. In a distributed setting, we assume that the lookup is done every time a received message is processed. Even if one would try to use something like a polymorphic inline cache [9] in an event loop, we assume the event loop to result in megamorphic behavior, because all messages and receiver types are funneled through a single point. Especially for dynamic languages with complex lookup semantics, the repeated lookups represent a significant cost compared to the cost of method invocation in the sequential case. Another issue for reaching peak performance is that information about the calling context is typically lost. For optimizing dynamic languages, this is however highly relevant to enable optimistic type specializations. Imagine a simple method adding numbers, depending on the caller a method might be used exclusively for adding integers, or in another setting exclusively for adding doubles. When an optimizer is able to take such calling-context information into account, it might be able to produce two separate compilations of the method for the different callers, which then can be specialized to either integers or doubles avoiding unnecessary run time checks and value conversions. For handlers registered on promises, lookup is no issue because the handle directly corresponds to the code to be executed. The calling context however might also be an issue if the same handler is used in multiple distinct situations.

Ensuring Isolation Between Actors. The second optimization challenge is to minimize the overhead of guaranteeing isolation between actors. Many pragmatic systems forgo isolation because of performance concerns. Examples include many actor libraries for the JVM [11] including Akka and Jetlang, as well as JCSP[9] and Go,[10] which implement *communicating sequential processes* [7]. SOM_{NS} provides this guarantee because we see it as an essential properties that make CELs useful from an engineering perspective.

To guarantee isolation, SOM_{NS} needs to ensure that the different types of objects are handled correctly when being passed back and forth between actors. Specifically, mutable objects need to be wrapped in far references so that other actors have no synchronous access. Far references on the other hand need to be checked whether they reference an object that is local to the receiving actor to unwrap them and guarantee that objects owned by an actor are always directly accessible. When promises are passed between actors, SOM_{NS} needs to create a new promise chained to the original one. This is necessary to ensure that asynchronous sends to promises that resolve to value objects are executed on the lexically correct actor. Since value objects do not

[9] *Communicating Sequential Processes for Java (JCSP)*, Peter Welch and Neil Brown, access date: 2015-07-05 `www.cs.kent.ac.uk/projects/ofa/jcsp/`

[10] *The Go Programming Language*, `golang.org`

have owners, we bind promises to actors and resolve the new promise with the original one when passing them between actors. Similar to asynchronous sends, handlers registered on promises need to be scheduled on the correct actor, i. e., the one that registered them. Thus, promises need to be handled differently from other objects passed between actors. For value objects, it needs to be determined efficiently whether they are deeply immutable, so that they can be passed safely as direct references.

For message sending, distinguishing between all these different cases has a negative impact on performance. Thus, finding ways to minimize the number of checks that need to be performed would reduce the cost of guaranteeing isolation. One conceptual benefit of Newspeak, and with it SOM_{NS}, is that the message send semantics do not require copying of objects graphs, which is required, for instance, for message sending between JavaScript Web Workers.

Efficient Promise Resolution. The optimization of promise resolution and scheduling of their messages and handlers is hard because promises can form tree-shaped dependencies, and in this dependency tree, promises from different actors can be involved. Ideally, resolving a promise would only require to schedule a single action on the event loop of the actor owning the promise. In this case, guaranteeing isolation and scheduling the corresponding action on the event loop could be done in code that is straightforwardly compiled to efficient native code. However, when a promise p_A is resolved with a promise p_B, p_A's resolution handlers are not scheduled immediately, instead, they will only be triggered once p_B has been resolved. Similarly, sending an asynchronous message to a promise means that the message is sent to the value to which the promise is eventually resolved. In the general case, this means, the resolution process has to traverse a tree structure of dependent promises and schedule all the handlers and messages registered on these promises on their corresponding event loops. Since the dependent promises can originate from different actors, we also need to check at each point whether the resolved value is properly wrapped to guarantee isolation. Another pitfall with promise resolution is that a naive implementation could easily cause a stack overflow in the implementation, which would cause a crash when resolving long dependency chains of promises.

4. FIRST OPTIMIZATIONS

The previous section outlined some of the challenges for optimizing SOM_{NS}. This section introduces initial optimizations that address them.

Send-site Lookup Caching. To avoid the repeated lookup overhead for asynchronous messages, we rely on SOM_{NS} executing on a shared memory system. This allows us to introduce polymorphic inline caches (PIC) for the send-site of asynchronous messages. Specifically, we use Truffle's notion of a `RootNode`, which essentially correspond to functions. At each send site of an asynchronous message, a root node is constructed that contains a normal SOM_{NS} synchronous message send operation, which already utilizes a PIC. This root node is eventually used when processing the asynchronous message in the event loop. This approach has two major benefits. On the one hand, we achieve specialization based on the send site. Ideally, the send is monomorphic, as most methods call are, so that is requires only a simple identity check of the receiver's class before executing the cached method. Since we reuse the normal synchronous send operation at the receiver site, Truffle also does method splitting and thus, enable the use of profile information based on the specific send site. On the other hand, creating the root node allows us to put these performance critical operations within the scope of Truffle's meta-compilation. This means, when the event loop takes a message for execution, it will eventually call directly into compiled code from the event loop and does not perform any generic operations that cannot be optimized. By constructing the root node that is send-site specific, but executes and performs the caching only in the target event loop, this optimization is applicably to all types of sends in SOM_{NS}. Thus, asynchronous sends to far references, to direct references, and to promises are optimized in the same way.

Compared to normal synchronous method invocation, asynchronous messages have the cost of the message queuing and cannot be inlined into the caller, since this would violate the semantics. Beside that however, we enable the other classic compiler optimizations, which can eliminate the cost of SOM_{NS}' dynamicity.

Guaranteeing Isolation. As discussed before, guaranteeing isolation requires to check at run time whether objects need to be wrapped in far references or have to be near references, make sure that promises are treated correctly to ensure their desired behavior, or to check whether an object is a deeply immutable value object.

To efficiently check deep immutability of values, we rely on Newspeak's semantics. All objects of classes that include the `Value` mixin are supposed to refer only to deeply immutable objects themselves. Since object constructors can however execute arbitrary code, we chose to check at the end of a constructor whether all fields of a value object contain only value objects themselves. For these checks, we assume that objects are usually going to be initialized with the same types of objects for each field. Thus, we enable specialization of the value check for each field separately, which in the ideal case means that per field a simple type check or read of a flag is sufficient to determine whether the constructed object is a legal value object. Like all of the optimizations discussed here, this optimization relies on a lexical stability of program behavior, which for a majority of programs is given or can be reached by method splitting based on the usage context. With the correctness check on construction, value objects can be recognized based on a flag that is set in these objects without having to traverse the object graph.

To ensure isolation, we optimize asynchronous sends to far references, handler registration and asynchronous sends to promises that are already resolved, as well as explicit promise resolution. For all these cases there is a concrete lexical element in the program, and consequently the AST can contain a node that can specialize itself based on the observed values. For the asynchronous sends, this means for each argument

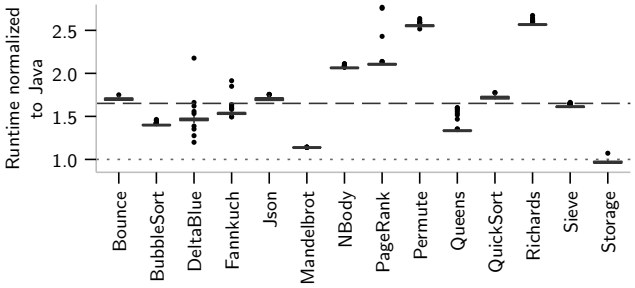

Figure 1: Peak performance comparison between SOM_{NS} and Java on classic sequential benchmarks. The dotted line is the Java performance based on the Graal compiler. The dashed line is the geometric mean over all SOM_{NS} benchmarks.

to the message send, a wrapper node is inserted into the AST that can specialize itself to one of the various cases that need to be handled. The assumption is that checking a guard such as the type of an object and whether sender and receiver actor are different is a faster operation than having to process all different cases repeatedly.

For the case that a handler is registered on an already resolved promise, or an asynchronous message send is performed, the same optimization applies and the operations are specialized directly in the AST corresponding to the interaction with the promise. However, for unresolved promises, this only applies to the arguments of the asynchronous message send. For the value to which the promise is resolved, we cannot use the same specialization. Since promises are normal objects, they can also be resolved explicitly by calling the `resolve()` method on the corresponding resolver object. For this specific case, the specialization is again applicable since it can be done as part of the AST element that does the call to the `resolve()` method. For the resolution of the promise that is the result of an asynchronous message send, this optimization applies as well. As discussed before, for each asynchronous send, we construct a root node that contains the actual synchronous send, i. e., method invocation done on the receiver side. We use this root node also to perform the promise resolution with the return value of the method invocation. Here, the send-site based specialization again provides the necessary context for the specialization.

5. PRELIMINARY RESULTS

For each benchmark, we measured 100 consecutive iterations within the same VM after 150 warmup iterations. The results represent SOM_{NS} peak performance. The benchmarks are executed on a system with two quad-core Intel Xeons E5520 processors at 2.26 GHz with 8 GB of memory and runs Ubuntu Linux with kernel 3.13, and Java 1.8.0_60.

To give an intuition of SOM_{NS}' performance, we compare it with Java. Since Truffle relies on the Graal compiler, we chose to also use Graal for the Java benchmarks to avoid cross comparison between compilers. On this benchmark set, Graal is about 10.9% slower than HotSpot's C2 compiler, which we consider more than acceptable. The results in fig. 1 show that SOM_{NS} is 1.65x (min. −3%, max. 2.6x) slower than Java on our set of benchmarks.

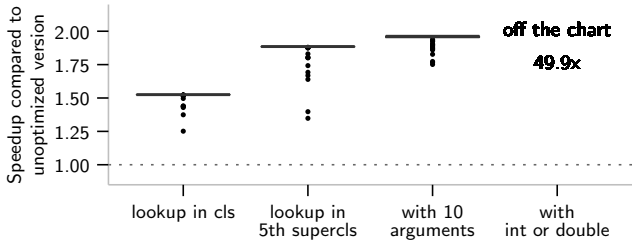

Figure 2: Speedup of asynchronous send operation compared to unoptimized SOM_{NS}. Microbenchmarks focus on lookup caching, ensuring of isolation, and preservation of calling context.

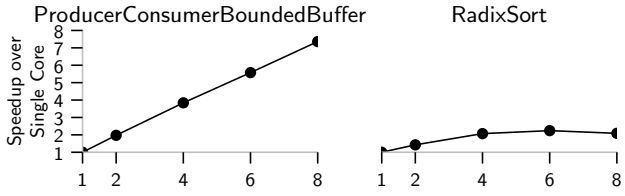

Figure 3: Results for two Savina benchmarks [10] to demonstrate scalability on multiple cores.

The microbenchmarks compare SOM_{NS} with and without the discussed optimizations. As depicted in fig. 2, the caching of lookups and the optimizations to reduce the run-time checks for the argument handling give a speedup of 1.5x to 2x on these microbenchmarks. The preservation of the calling context to enable optimizations can give even more speedup of 49.9x, which unfortunately does not fit onto the chart. As an initial verification that the parallel execution leads to speedup, we chose two of the Savina benchmarks [10] that have potential for parallelism. Figure 3 indicates that an increased number of actors indeed increases performance.

6. CONCLUSION

This first exploration investigates optimization challenges for communicating event-loop languages. With SOM_{NS}, a Newspeak implementation based on Truffle, we show that they can be implemented efficiently with Truffle. SOM_{NS} is only 1.65x slower than Java. Furthermore, we show that send-site caching reduces the lookup overhead, cost of ensuring isolation, and enables the use of the calling context for optimization. On microbenchmarks, we see speedups of 1.5x to 2x, while the use of the calling context for optimization can give a speedup of 49.9x. Finally, we also show that SOM_{NS} can realize parallel speedup on two benchmarks. While these are only preliminary results, some of the ideas are applicable to other types of languages. Since asynchronous message reception rarely leads to monomorphic behavior, send-site-based optimizations could also be beneficial for statically typed languages.

Nonetheless, much work remains to be done. For instance, we do not yet have a solution of handling complex promise dependencies efficiently and we did not yet verify the benefit of these optimizations on larger actor programs. We however hope, SOM_{NS} is an interesting platform for future research not only of optimization techniques but also for safe concurrent programming models beyond classic actors.

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_20.

[3] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3):143–155, Mar. 1966. ISSN 0001-0782. doi: 10.1145/365230.365252.

[4] T. Desell and C. A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In G. Agha, A. Igarashi, N. Kobayashi, H. Masuhara, S. Matsuoka, E. Shibayama, and K. Taura, editors, *Concurrent Objects and Beyond*, volume 8665 of *LNCS*, pages 144–166. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44470-2. doi: 10.1007/978-3-662-44471-9_7.

[5] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015.

[6] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[7] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. ISSN 0001-0782. doi: 10.1145/359576.359585.

[8] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114.

[9] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *LNCS*, pages 21–38. Springer, 1991. ISBN 3-540-54262-0. doi: 10.1007/BFb0057013.

[10] S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 67–80. ACM, 2014. ISBN 978-1-4503-2189-1. doi: 10.1145/2687357.2687368.

[11] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596658.

[12] S. Marr and S. Ducasse. 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, 2015. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660194.

[13] S. Marr, T. Pape, and W. De Meuter. Are We There Yet? Simple Language Implementation Techniques for the 21st Century. *IEEE Software*, 31(5):60–67, September 2014. ISSN 0740-7459. doi: 10.1109/MS.2014.98.

[14] M. S. Miller, E. D. Tribble, and J. Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005. doi: 10.1007/11580850_12.

[15] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing Active Objects to Concurrent Components. In *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *LNCS*, pages 275–299, Berlin, 2010. Springer. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_13.

[16] T. Van Cutsem, E. Gonzalez Boix, C. Scholliers, A. Lombide Carreton, D. Harnie, K. Pinte, and W. De Meuter. AmbientTalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40(3–4):112–136, 2014. ISSN 1477-8424. doi: 10.1016/j.cl.2014.05.002.

[17] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'13, pages 187–204. ACM, 2013. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509581.

[18] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587.