Self-Optimizing AST Interpreters

Thomas Würthinger* Andreas Wöß† Lukas Stadler† Gilles Duboscq†
Doug Simon* Christian Wimmer*

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
thomas.wuerthinger@oracle.com woess@ssw.jku.at stadler@ssw.jku.at duboscq@ssw.jku.at
doug.simon@oracle.com christian.wimmer@oracle.com

Abstract

An abstract syntax tree (AST) interpreter is a simple and natural way to implement a programming language. However, it is also considered the slowest approach because of the high overhead of virtual method dispatch. Language implementers therefore define bytecodes to speed up interpretation, at the cost of introducing inflexible and hard to maintain bytecode formats. We present a novel approach to implementing AST interpreters in which the AST is modified during interpretation to incorporate type feedback. This tree rewriting is a general and powerful mechanism to optimize many constructs common in dynamic programming languages. Our system is implemented in Java™ and uses the static typing and primitive data types of Java elegantly to avoid the cost of boxed representations of primitive values in dynamic programming languages.

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

General Terms Algorithms, Languages, Performance

Keywords Java, JavaScript, dynamic languages, virtual machine, language implementation, optimization

1. Introduction

When designing and implementing a new programming language, the first and easiest choice is to implement an abstract syntax tree (AST) interpreter. The interpreter is often embedded in a virtual machine (VM) that provides other services such as memory management. Parsers for context-free grammars process source code in a tree-like fashion, so creating an AST feels natural and can even be automated. Making an AST executable, i.e., writing an AST interpreter, just requires adding an `execute` method to every AST node.

A common argument against AST interpreters is performance: calling the `execute` method for every node requires a virtual method dispatch that is costly, and call sites within the `execute` methods are highly polymorphic and therefore difficult to optimize. For this reason, language implementers define bytecodes to speed up interpretation. Bytecode interpreters are well understood, and many variants with varying dispatch costs have been studied [9, 22]. Bytecodes are a virtual instruction set, i.e, they can be defined freely by language implementers without worrying about hardware constraints. There are two different strategies when defining bytecodes. We believe that both are cumbersome when implementing a new language:

- Language-independent bytecodes aim to provide an instruction set that is suitable for many languages. Java™ bytecodes [17], originally designed primarily for the Java programming language, are now used for many languages. The Common Intermediate Language (CIL) [8], which is used by the .NET system, was specifically designed for multiple languages. Language-independent bytecodes provide only general instructions whose semantics do not match the language semantics exactly. Long bytecode sequences are potentially required to achieve the desired language behavior. For example, Java bytecodes are statically typed, so compiling dynamically typed JavaScript requires frequent type checks. A simple addition in JavaScript requires checking whether the operands are numbers, strings, or arbitrary objects, which all require different handling. Therefore, implementations of JavaScript that run on a Java VM often call a method for a seemingly simple addition.
- Language-specific bytecodes are usually used internally by a VM, often even without a clearly defined specification. They closely resemble the semantics of the programming language, so little can be reused from existing language implementations when implementing a new language. When the language evolves, the bytecodes also have to evolve. This makes language-specific bytecodes cumbersome to define and maintain.

In any case, bytecodes are difficult to modify once they are emitted. Exploiting profiling information, such as types that were observed during execution, is not only useful for just-in-time (JIT) compilers but can also improve the performance of interpreters. However, the compact binary encoding of bytecodes makes this difficult. For example, to replace one generic bytecode (a call to a type-agnostic addition method) with a short sequence of type-specialized bytecodes (a few type guards followed by an integer addition bytecode) requires adjusting offsets in branch instructions that jump over the replaced code.

ASTs are more malleable than bytecodes. It is possible to incorporate profiling information immediately after it is discovered, by rewriting the AST. For example, a tree node for a type-agnostic addition can be replaced with a specialized integer addition node after the first execution of the addition reveals that the operands were integers. This is based on the observation that types are mostly stable, even in dynamically typed languages.

In this paper, we show that while naive interpretation of an AST is slower compared to bytecodes, use of profiling and AST

DLS'12, October 22, 2012, Tucson, Arizona, USA.

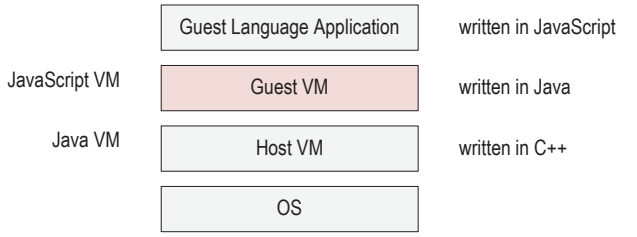

Figure 1. System structure of our guest VM on top of a host VM.

rewriting allow an AST interpreter to achieve speeds comparable to the speed achievable with language-independent bytecodes. We use Java and the Java VM for our implementation, and JavaScript as the interpreted language. However, we are more interested in the general concepts than in the specifics of JavaScript, therefore our prototype does not implement the full language semantics of JavaScript. While we believe that our rewritten type-specialized ASTs are excellent input for a JIT compiler, we leave a detailed investigation of this for future work (see Section 8). In summary, this paper contributes the following:

- The definition of a simple yet efficient AST-interpreter-only execution of a guest language on top of an existing statically typed VM.
- Tree rewriting as a way to optimize and type-specialize AST interpreters at run time.
- An evaluation showing that our AST interpreter achieves performance similar to other, more complicated, implementations that generate language-independent bytecodes.

2. System Structure

The rise of dynamic languages has led to a plethora of new VMs, since a completely new VM is typically developed for every new language. VMs are still mostly monolithic pieces of software. They are developed in C or C++, the languages that they aim to replace. VMs offer many benefits for applications running on top of them, but they mostly do not utilize these benefits for themselves. In contrast, we aim for a layered approach where a *guest VM* is running on top of a *host VM*. The *guest language* is the language that we want to define and write our application in, while the *host language* is the language that we write our guest VM in, i.e., the language that the host VM executes. For our examples, the guest language is JavaScript and the host language is Java, i.e., we have a JavaScript VM implemented in Java that runs on top of a Java VM. Figure 1 summarizes our system structure.

This layered approach has several advantages that simplify the implementation of a guest language. The host VM provides many services such as automatic memory management, exception handling, threads, synchronization primitives, and a well-defined memory model that can be leveraged by the guest VM. It is not necessary to implement a complete VM, allowing the language designer to focus on the execution semantics of the language. The guest VM is completely implemented in the high-level host language (in our case Java) and not in a low-level language such as C or C++. The host VM abstracts platform and architecture specific details, so the guest language implementation is automatically portable: our pure-Java implementation of JavaScript runs unmodified on any architecture and operating system where Java is supported. The wide availability of free high quality tools (e.g., Eclipse, NetBeans) further extend the benefits of developing in Java.

Everything in our guest VM is written in Java and expressed using Java concepts. The language parser as well as the AST interpreter with its `execute` methods is Java code that only accesses normal Java objects. This requires that guest language objects and guest language metadata are modeled as Java objects. Additionally, interpreter data structures such as stack frames are Java objects. This approach completely avoids Java bytecode generation. We do not generate Java classes on-the-fly to represent guest language classes, and we do not generate Java bytecodes for guest language methods.

This approach also has some constraints. First, we are limited to the Java object model and memory model, and to the primitive types offered by Java. For example, we cannot employ tagged pointers, a common method to overlay primitive values and references. This is not a problem because we specialize the AST for types, i.e., we can distinguish between primitive values and references at the AST level. Only in rare cases do we have to box primitive values into objects so that we can store them as references.

Second, our native interface is limited to the Java Native Interface (JNI) since this is the only native interface offered by a Java VM. JNI is well specified and completely hides VM implementation details from native code. Therefore, native interfaces of guest languages that expose VM details are difficult to implement. We argue that this is a flaw in the guest language design and should be addressed by the language designer. A native interface that exposes VM internals not only prevents a language implementation in Java, but any new implementation of the language.

3. AST Interpretation

Implementing an AST interpreter for a language is a simple and natural way of describing the semantics of the language. Every operation in the language is encoded in an AST node. A node is responsible for executing its associated operation and returning a result value (if non-void). The execution of a method is done by executing its root AST node. A frame object allows the nodes to access local variables as well as other contextual information necessary for execution, e.g., access to the caller frame.

We model AST nodes in the Java programming language as shown in Figure 2. There is a common abstract node base class with a method for executing the node. A Java object representing the execution frame is a parameter to that method. The frame itself contains an `Object[]` array with the values of the local variables. All values of the language are represented as Java objects, therefore the result value of the `execute` method is of type `Object`. Every node contains a pointer to its parent node in the AST. This allows for a fast replacement of one node with another node.

```
abstract class Node {
  // Executes the operation encoded by this
  // node and returns the result.
  public abstract Object execute(Frame f);

  // Link to the parent node and utility to
  // replace a node in the AST.
  private Node parent;
  protected void replace(Node newNode);
}
```

Figure 2. Definition of the AST node base class.

The control structures of the language are generally implemented using Java control structures. Non-local returns, i.e., control flow from a deeply nested node to an outer node in the AST, are modeled using exceptions. Figure 3 shows how a while loop is implemented. The Java `while` construct is used for the loop. A `ContinueException` and a `BreakException` allow the nodes that represent the *continue* or *break* statements inside the body of the while loop to continue or break the loop.

```
class WhileNode extends Node {
  protected Node condition;
  protected Node body;

  public Object execute(Frame frame) {
    try {
      while (condition.execute(frame)) {
        try {
           body.execute(frame);
        } catch (ContinueException ex) {
           // Continuing in the loop.
        }
      }
    } catch (BreakException ex) {
      // Breaking out of the loop.
    }
    return null;
  }
}
```

Figure 3. Implementation of a `while` node.

4. Tree Rewriting

The ability to replace one node with another is a key aspect of our system. It enables us to improve the executed AST at run time by replacing a node with a more specialized node. Based on profiling feedback from the previous and current input operands, a node is replaced with a specialized node that can perform the operation for these operands faster, but cannot handle all cases. This means that the specialized node makes *assumptions* about its operands. We optimistically assume that the assumptions hold for future executions. The typical implementation of the `execute` method of a node checks that any assumptions still hold and then performs its operation. If the assumptions do not hold, the node replaces itself with a node that can handle the more generic case. The following subsections describe practical examples where we use this replacement technique.

4.1 Operation Specialization

Figure 4 gives an example of a JavaScript function that adds two values. Depending on the actual type of `a` and `b`, the operation performed here can vary between a simple numeric add (integer or double), a string concatenation, or a call to custom JavaScript conversion methods. A static analysis in the context of JavaScript is problematic given that real-world programs rarely allow for a closed world analysis. Even if it could be performed, it is highly unlikely to pay for itself with sufficiently faster interpretation. Therefore, the interpreter must be capable of handling all possible cases.

```
function add(a, b) {
  return a + b;
}
```

Figure 4. JavaScript example.

Instead of only implementing a single node whose `execute` method handles all cases, we split the implementation of the operation into several different nodes with transitions between them. Each of the nodes handles a subset of the aforementioned cases and includes a check on its inputs. If the inputs are not of the predicted type, we replace the node with a different node that can handle the new case appropriately. The transition between the different nodes is unidirectional. The different node classes form a lattice where the bottom node is capable of handling all cases. However, this big and complex method is rarely executed.

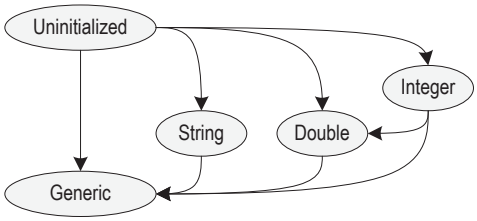

Figure 5. Transitions between different types of add nodes.

In the case of the plus operator in the example, we have five different node types. Figure 5 shows the lattice with the possible type transitions. The top node (*Uninitialized*) is the initial state after parsing where no type information is available. This node has no functional implementation, so a rewrite to another node is mandatory for the first execution. The node *integer* is the desired specialization target because on all current architectures integer operations can be performed faster than floating point computations. Even though the JavaScript language specification does not know an integer type, we can transparently insert it into the lattice for optimization. As long as both arguments are integer numbers and the addition does not overflow, we can use this fast path; otherwise, we convert to the *double* node. The *string* node handles the case of string concatenation. The *generic* node is the only node that can handle custom JavaScript conversion methods. While this is a powerful language concept that allows the addition of any two objects, it is a rare case that does not need to be optimized.

The actual number of different node types for an operation and their connection depends heavily on the semantics of the implemented language and its common usage patterns. There are two notable properties in this system:

Correctness: The operation can never be in a wrong state. Every state can handle every possible input corner case by moving to another state. This means that we can at any point in time choose to undo a specialization by replacing the node with a more generic version.

Finiteness: There are no loops in the lattice, thus the number of transitions is finite and there cannot be an endless loop of state transitions. This is important because a state transition implies a run-time overhead and we want the AST to stabilize during the execution of a program.

4.2 Type Decision Chains

The AST rewrite principle can be extended to form type decision chains. A type decision chain is a variation of a polymorphic inline cache [15] implemented via AST node chaining instead of a jump table in generated code. Type decision chains can be used to optimize JavaScript property accesses as well as virtual method calls. In this context, a *type* is expanded to mean the set of properties and methods defined for an object. This is analogous to the use of *maps* in Self [6].

Figure 6 shows a series of AST transformations reflecting the construction of a type decision chain. Similar to the operation specialization described in Section 4.1, the AST contains an uninitialized node in the beginning. This uninitialized node gets the concrete type of the input(s) on which it should perform the operation, e.g., property access or method call. With this concrete type, it creates a specialized AST node that is capable of performing the operation for exactly this type. Guest language types are represented as Java objects, so the type check is a simple object identity comparison. If the type does not match, the specialized AST node dispatches to a subsequent node. This way, the different concrete types that occur at a polymorphic site are encoded in the state

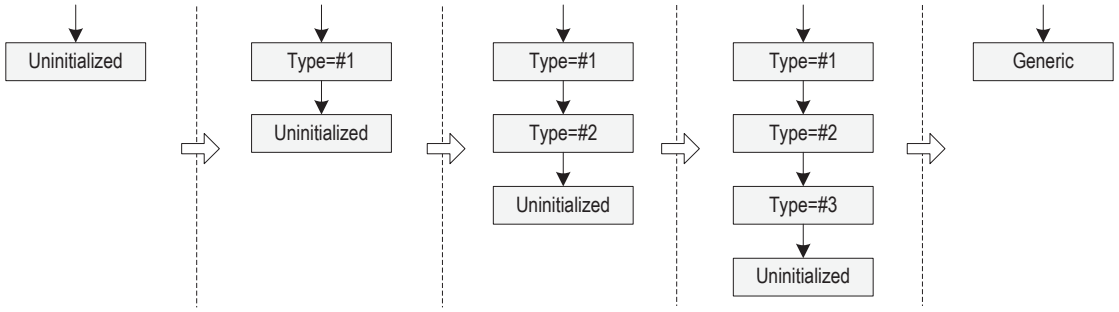

Figure 6. Transformations of the AST for a polymorphic operation.

of the AST. If the number of different types exceeds a certain limit, we declare the call site megamorphic and replace the decision chain with a generic implementation of the operation. Alternately, we can choose to place the generic node at the end of the chain if it is still beneficial to have the specialized versions for the current types in the chain. Finally, at any point in time we can try to re-profile the chain by replacing it with an uninitialized node again.

Decision chains can potentially form trees. This is beneficial if the operation performed for a specific type again needs a polymorphic dispatch. Currently, we use this for efficient implementation of JavaScript's prototype mechanism. Here, the type of the value decides whether the prototype must be accessed for a specific property or not. In case the prototype must be accessed, there is an additional decision chain for different prototype types.

5. Dynamic Data Type Specialization

The main performance problem of the system described so far is the boxing necessary to allow one value to hold both primitive values as well as references. A common solution to this problem is to use value tagging. We present here an alternative approach that we believe is better in terms of performance and simplicity.

5.1 Boxing

Java VMs, and most VMs in general, differentiate between primitive types and reference types. Primitive types directly represent values of varying ranges (integers, doubles, etc.), while reference types are pointers to data structures within the application's heap. The differentiation between primitive and reference types is important for the safety and soundness of the system (no pointer arithmetic), and only reference types need to be visible to garbage collection.

In the Java language, the static type of each value defines whether it is of primitive or reference type, so that the storage type is known beforehand. In JavaScript, however, the type of a value is not known beforehand, so that the system needs to be prepared to process both primitive and reference types.

The most common solution is to allocate storage of reference type, and whenever a primitive value needs to be stored it is wrapped into a small *boxing* object. This boxing object has exactly one field of the primitive type and is used as a proxy that allows the primitive value to be stored in a reference storage.

5.2 Tagging

Boxing imposes considerable overhead, because a new boxing object is created each time a primitive value needs to be stored. To avoid this, VMs sometimes employ *tagging*, which uses a bit-level flag to tell if a stored machine word is a primitive or a reference value. While this allows the system to store primitive values without boxing them, this has several disadvantages:

- While the least significant bit of object references is not needed when objects are aligned, the primitive types do not contain unneeded bits that can be used for object tagging.
- Each time a value is loaded, the system needs to check whether it is a primitive or reference value and remove the flag bits.
- Each time a value is stored, the system needs to set the according flag bits.

5.3 Return Type Specialization

The return value of the `execute` method is declared as being an arbitrary Java `Object` value (see Figure 2). The result of any node has to be boxed even when the node (e.g., an integer add node) knows that it can only produce integer values. We want to avoid that and therefore introduce specialized execute methods (e.g., `executeInt`) as shown in Figure 7. The semantics of those methods are defined in the following way. The caller specifies the primitive type that it desires to get from the callee. If the callee can deliver its result value in that form, it simply returns. If the callee cannot deliver its result value in that form, it throws an `UnexpectedResultException` containing the boxed version of the result. The caller is forced to handle such an exception and act appropriately, e.g., rewrite itself to a version that can handle this type of result.

```
abstract class Node {
  public abstract Object execute(Frame f);
  public abstract int executeInt(Frame f)
          throws UnexpectedResultException;
  public abstract double executeDouble(Frame f)
          throws UnexpectedResultException;
  // ...
}
```

Figure 7. Return type specialized execute methods.

Figure 8 shows how a specialized node such as a JavaScript integer add is programmed against this modified framework. The operation expects that both of its inputs return their values as integers. If one of them fails, the node rewrites itself and propagates the unexpected result exception to its caller. The code shown in the figure may seem complex, but in the normal case, the catch blocks are never executed. The actual operations performed are getting the values from the left and right operand and performing an integer addition. There is no boxing, no unboxing, and also no type check involved. In the normal execution path, the inputs are guaranteed to be of the appropriate type.

Note that even if one of the calls to `executeInt` throws an `UnexpectedResultException`, the associated node has already been evaluated. This implies that any side effects have already occurred and the node that replaces the `IntegerAddNode` needs

```
class IntegerAddNode extends BinaryNode {
  public int executeInt(Frame frame)
          throws UnexpectedResultException {
    int a;
    try {
      a = left.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // Rewrite this node and execute the rewritten
      // node using already evaluated left.
    }
    int b;
    try {
      b = right.executeInt(frame);
    } catch (UnexpectedResultException ex) {
      // Rewrite this node and execute the rewritten
      // node using already evaluated left and right.
    }
    // Overflow check omitted for simplicity.
    return a + b;
  }
}
```

Figure 8. Integer add node using return type specialization.

to use the boxed result from the `UnexpectedResultException` the first time it is executed. In case of the binary `IntegerAddNode` the system needs to be aware if only `left` or both `left` and `right` have been executed, because in the first case it needs to use the cached result for `left` and in the second case it needs to use the cached results for both `left` and `right`.

Figure 9 shows how the integer add node would be programmed without type specialization to highlight the difference. In contrast to the code in Figure 8 with return type specialization, there are two type checks, two unboxings, and one boxing in the fastest path.

5.4 Local Variable Specialization

We apply specialization not only to the result values of nodes, but also to local variables. This mechanism is only interesting for dynamic languages where the type of a local variable is not specified. For JavaScript, we distinguish four different states for local variables: uninitialized, integer, double, object. As with the type lattice for node transitions, local variables go through the same possible transitions during their lifetime. Unlike normal nodes, when the type of a local variable changes, every occurrence of the local variable in the AST must be changed atomically. Such a change can only happen when there is a value of a different type being stored into a local variable. The assignment of a value to a local variable is itself specialized. This means that if we have an assignment specialized as an integer assignment of a value to a local variable that is specialized as integer, we can avoid any kind of boxing or unboxing. The assignment calls the `executeInt` method of the node representing the value that should be assigned. Then it stores this integer directly into the frame of the method.

In order to allow primitive and object values for local variables, we use a `Frame` object with an `Object[]` field and a `long[]` field for the local variables in the frame. If the value is primitive (e.g., integer or double), then we directly[1] store the value into the `long[]`, otherwise it is stored in the `Object[]`.

In contrast to a Java VM's mechanism for finding references in native interpreter frames, the references in AST frames are simply the values in the `Object[]` field.

[1] We use utilities such as `Double.doubleToRawLongBits()` to ensure no value conversion occurs when storing a value as a `long`.

```
class IntegerAddNode extends BinaryNode {
  public Object execute(Frame frame) {
    Object a = left.execute(frame);
    if (!(a instanceof Integer)) {
      // Rewrite this node and execute the rewritten
      // node using already evaluated left.
    }
    Object b = right.execute(frame);
    if (!(b instanceof Integer)) {
      // Rewrite this node and execute the rewritten
      // node using already evaluated left and right.
    }
    // Overflow check omitted for simplicity.
    return (Integer) a + (Integer) b;
  }
}
```

Figure 9. Integer add node without return type specialization.

Local variable specialization has some implications in case of recursive method calls. If there is a specialization in a function that has an activation somewhere up on the stack, we need to ensure that upon return to that activation we convert the value of the local variable in the frame for this activation from the previously assumed type to the new type. This conversion is guaranteed to be possible since the types only change according to the type lattice (see for example Figure 5), i.e, types change from more specific to more generic types.

5.5 Field Specialization

For dynamic languages without classes, we use a concept similar to *maps* in Self [6]. Every object has a pointer to an object array that holds the fields of the object. Additionally, there is a pointer to the map of the object: a Java object that describes the layout of this array, i.e., which field is at which index. A store to an unknown field triggers a map transition, where the map of an object changes. This map is used on our polymorphic chains for efficient implementation of polymorphic field accesses (see Section 4.2).

In order to avoid boxing of values when storing to fields, we propose type specialization of the fields. This means that the map not only contains the information at which index a field is stored, but also the primitive type of the field. An object of the dynamic language can contain an additional `long[]` array for those values or also potentially a few `long` Java fields that serve as placeholders for primitive data. A store to a known field that would change the type of the field triggers a map change.

The duplication of the `Object[]` and the `long[]` array can be avoided if the GC is aware of the layout description for guest language objects. However, the overhead is not high for common JavaScript programs. The benefit of avoiding boxing outweighs the overhead of a second array. It is possible to store a fixed number of primitive and reference values in the object itself, which eliminates the need for the extended arrays in most cases (we have not implemented this optimization yet, so for the performance numbers in Section 7.1 all fields are stored in the arrays).

6. Method Inlining

Figure 10 shows an extended version of the JavaScript program with the `add` function. The function is now called twice: the first time the parameters are always numbers, and the second time the parameters are always strings. When applying the operation specialization described in Section 4.1, we end up with a generic version of the add operation. This results in a performance loss, penalizing programmers who factor out common functionality into

```
function foo() {
  return add(1, 2) + add("hello", "world");
}

function add(a, b) {
  return a + b;
}
```

Figure 10. Extended JavaScript example.

helper methods. That is, applying recommended software engineering practices can decrease the performance of the program.

In comparison to dynamic languages, the static typing of Java somewhat mitigates the need for run-time type feedback in terms of achieving good performance. However, use of type profiling is still a major issue for non-primitive types in Java as the information about polymorphic call sites gets less specific and branch probability accuracy decreases as more inlining is applied by the optimizing compiler. Some abstractions used in Java libraries can have their performance severely impacted by this issue. For example, a `foreach` method on a Java collection object that takes a closure as its parameter has a megamorphic call site at the point where the closure is called. This megamorphic call site contains the type of every closure ever given to the `foreach` method. Inlining the `foreach` method into its call site greatly increases the chance that the closure call becomes less polymorphic and may even be monomorphic.

We believe that the core of the problem is the gap between the inlining performed by the optimizing compiler and the way the interpreter executes the program. Therefore, we propose to perform function inlining on the AST level. If we find out that a particular call site (not a method) is hot, we duplicate the AST of the called method and put the copy back into the uninitialized state. This way, the AST of that method is specialized based on its usage patterns from the call site. This allows us to gather context-specific profiling feedback. Figure 11 illustrates this method inlining process, where the big circles represent methods, and the small circles represent AST nodes.

Figure 12 shows the evolution of the AST (simplified) when executing the JavaScript program from Figure 10. In the first version of the AST, the connection between the call sites and the method `add` is through function calls (stage 1). The main difference between this connection and a normal connection between two AST nodes is that the former does not have a parent pointer and also that there may be multiple parents for a function node (in this example two). The nodes that implement a specific method always form a tree whereas the connections between trees form a graph, i.e., the call graph. Cycles in the graph caused by recursion can be dealt with by inlining heuristics, e.g., stop inlining once the AST reaches a certain number of nodes.

There are two plus operations; both are currently in the uninitialized state. After execution of the first `add` function call, the plus operation in the add function moves to the integer state (stage 2). After the second `add` function call, the plus operation moves to the generic state, because the type of the inputs does not match the predicted integer type (stage 3). The plus operation in the `foo` function also moves from uninitialized to generic: we are adding an integer and a string, therefore we cannot specialize (stage 4).

After a predefined number of executions of the `add` call site, we recognize it is hot. In that case, we inline the called method into the caller. The `add` operation node gets duplicated and then replaced by the uninitialized version of that node (stage 5). During the first execution of that node, it gets specialized to be an integer node (stage 6).

When the second call site gets hot, we inline this call site too (stage 7). After the first invocation, the add operation that was

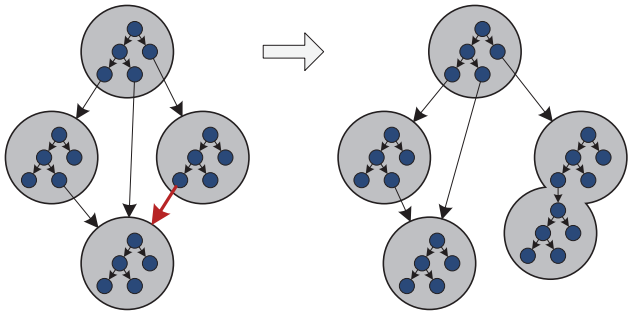

Figure 11. Method inlining in the dynamic call graph.

duplicated during this inlining, is specialized to string (stage 8). We can now execute both invocations of the `add` method specialized on parameter types, which results in a performance gain.

The important thing to note here is that we can apply this program specialization *without* any global analysis. Using only greedy and simple algorithms that operate locally on a node, dynamic specialization of the program achieves faster operations.

7. Evaluation

We implemented an AST rewriting interpreter for JavaScript in Java. Our implementation is pure Java source code that does not need any native code, machine code generation, or Java bytecode generation. The focus is on showing the concept and the potential of AST rewriting rather than a complete JavaScript VM. We have not yet implemented some features of JavaScript, such as `eval()` or regular expressions (see Section 8), but otherwise we adhere to the semantics of JavaScript. We use the V8 benchmark suite [12] for evaluation. Two benchmarks are excluded: *RegExp* (a regular expression benchmark) and *EarleyBoyer* do not yet run on our implementation.

7.1 Performance

To show the performance of our AST interpreter and the impact of AST rewriting, we compare the following configurations:

- *AST interpreter with tree rewriting*: Our AST interpreter with all optimizations described in the previous sections enabled.
- *AST interpreter without tree rewriting*: Our AST interpreter with tree rewriting disabled completely, i.e., the AST is never changed during execution. Still, the AST interpreter is implemented carefully to achieve the best possible performance, and it uses the same the structures to represent JavaScript objects and arrays.
- *AST interpreter rewriting only property access*: Our AST interpreter with tree rewriting only enabled for type decision chains (see Section 4.2).
- *Rhino with Java bytecode generation*: The Rhino JavaScript VM [18] version 1.7R2 with bytecode generation enabled. Similar to our system, Rhino is a JavaScript VM written in Java. It uses an AST interpreter as the first level of execution, and then translates frequently executed JavaScript methods to Java bytecodes. These bytecodes are then translated to optimized native code by the JIT compiler of the Java VM.
- *Rhino interpreter only* The Rhino JavaScript VM with bytecode generation disabled, i.e., using their AST interpreter only.

We run all configurations on the 64-bit Java HotSpot server VM of the JDK 7 update 2. The JIT compiler of the VM translates the AST interpreter methods to optimized machine code. Thereby,

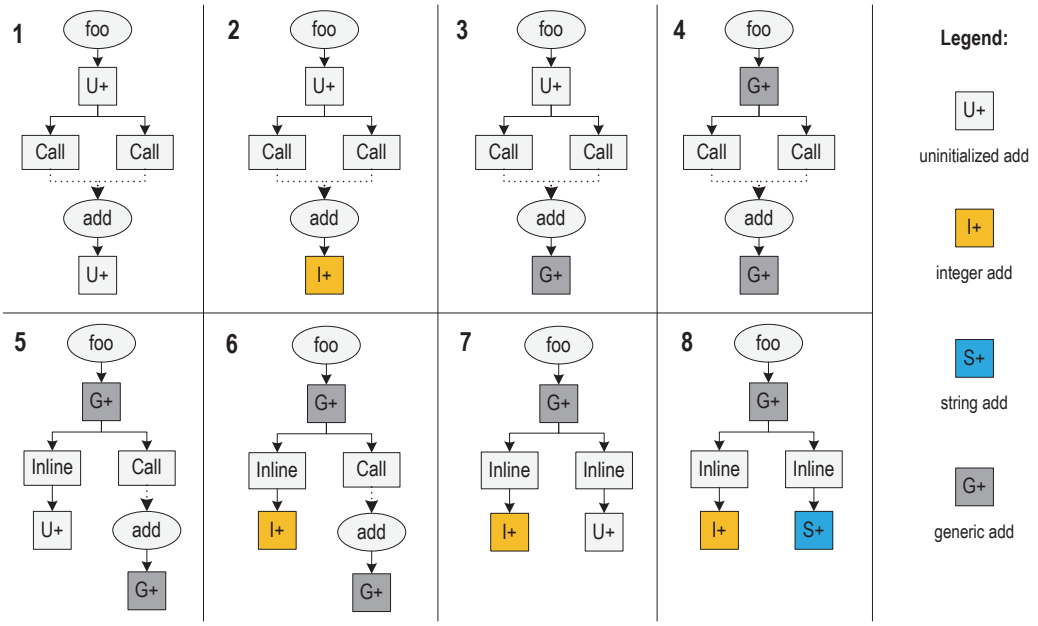

Figure 12. AST evolution for the extended JavaScript add example from Figure 10.

it performs aggressive optimizations such as global code motion and scheduling, graph-coloring register allocation, array-bounds-check elimination, loop invariant code motion, loop unrolling, and escape analysis. Additionally, it uses profiling information to, e.g., inline the most frequently called method of a polymorphic call site. This helps to optimize a few call sites of the `execute()` AST interpreter method, but still a lot of virtual method calls remain. When the Rhino VM generates Java bytecodes, these bytecodes are optimized the same way by the Java HotSpot VM. However, complicated bytecode sequences have to be emitted to simulate JavaScript semantics with typed Java bytecodes, leading to sub-optimal machine code being generated.

The benchmarks were executed on a two socket, dual core AMD Opteron 2214 with 2.2 GHz, a total number of 4 cores, and 4 GB main memory. The OS is Oracle Enterprise Linux, version 2.6.18. The reported numbers are the average of 10 executions.

Figure 13 shows the performance results. The numbers are speedups relative to our AST interpreter with all optimizations enabled, i.e., higher means better. Tree rewriting is an important optimization that greatly improves performance of the interpreter. For example, it leads to an 11x speedup for the *DeltaBlue* benchmark. The most important rewriting is the optimization of property and array accesses using type decision chains. This optimization speculates that types are stable, i.e., that objects of the same shape are used when performing a property lookup repeatedly even when the language itself is dynamically typed. This is a well-known observation that was already used to optimize languages such as Self [6]. In our tree rewriting interpreter, this optimization fits naturally into the general rewriting framework.

Rhino in its interpreter-only mode is in general much slower than our non-rewriting interpreter. This shows that our careful implementation style of the interpreter, combined with a good object model to represent JavaScript objects, can make a difference and lead to a nearly 2x speedup. With all of our implementations enabled, our interpreter is nearly 4x faster than the Rhino interpreter. This speed comes close to the speed of Rhino when generating Java bytecodes, making the fully optimized configuration of Rhino only 40% faster than our interpreter.

Interpreters are much slower than the code generated by optimizing JIT compilers. To show the performance possibilities for JavaScript, we compare our interpreter to the V8 JavaScript VM [11], one of the best performing JavaScript VMs with a JIT compiler developed and optimized solely for JavaScript. In comparison to our AST interpreter with tree rewriting, the V8 VM is 6x (benchmark *Splay*) to 62x (benchmark *Richards*) faster, with a mean of 28x. The future work in Section 8 shows how our rewritten AST can be the input for an optimizing compiler.

The results show that a carefully implemented AST interpreter with AST rewriting comes close in performance to much more complicated systems generating Java bytecodes, which are compiled to optimized native code by the Java VM. Performing high-level and dynamic language specific optimizations on the AST has more impact than the low-level optimizations that the Java VM can perform on Java bytecodes. The most important tree rewriting optimization, type decision chains, is difficult to implement in Rhino with bytecode generation because bytecodes cannot be rewritten once they are generated, i.e., it is not possible to speculate on type stability. Therefore, Rhino's bytecode generation would not profit from, e.g., our choice of object representation. In summary, generating Java bytecodes makes the implementation much more complex, but does not lead to a significant performance benefit.

7.2 Rewriting

Figure 14 shows how many nodes of certain kinds of operations are rewritten. We report the number of nodes that are rewritten at least once (columns '>0'), as well as the number of nodes that are rewritten more than once(columns '>1'). This shows that most nodes are rewritten only once, but then remain stable and unchanged for the rest of the execution. This implies that the types are stable, i.e., an arithmetic node that is rewritten once to type *Integer* often remains at this type. The first two rows are arithmetic operations and comparisons, while the following three rows are variable assignments and loads. Loads from global variables are accesses from the JavaScript global object, which we rewrite to special nodes so that we can optimize them more than normal property accesses. The rows for property access and array access

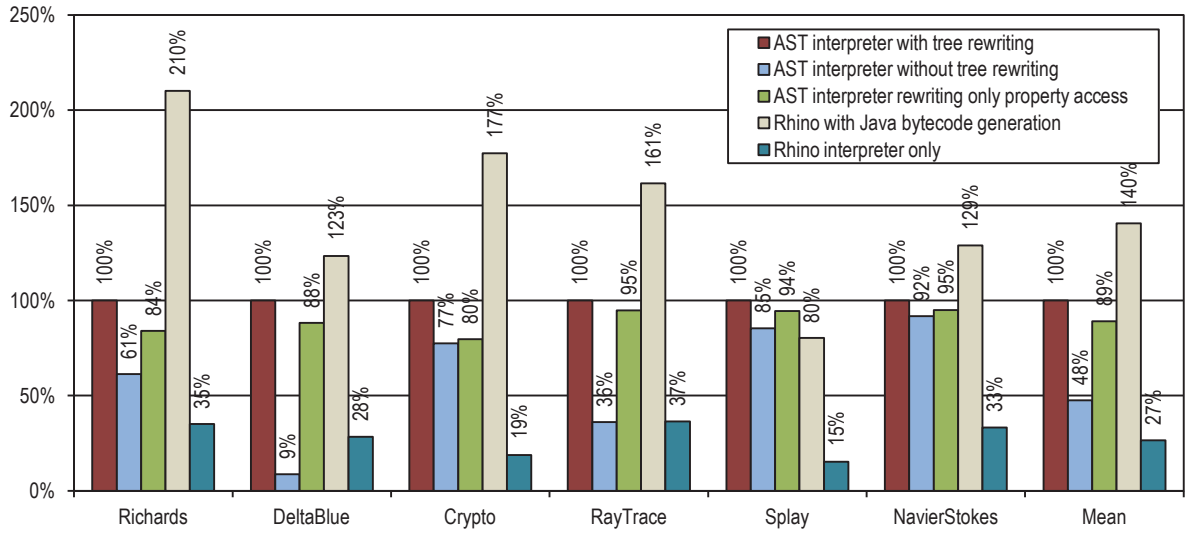

Figure 13. Performance impact of our AST rewriting, and comparison with the Rhino JavaScript VM (higher is better).

	Richards		DeltaBlue		Crypto		RayTrace		Splay		NavierStokes	
	>0	>1	>0	>1	>0	>1	>0	>1	>0	>1	>0	>1
Binary Op	63	1	96	1	446	9	190	22	64	4	230	2
Unary Op	0	0	1	0	5	0	18	2	0	0	7	0
Assignment	217	0	286	1	769	0	287	8	155	1	253	0
Load Local Var	358	12	568	10	1600	25	737	40	340	14	842	8
Load Global Var	139	0	206	0	474	1	221	0	81	0	71	0
Property Access	390	2	897	3	829	0	912	3	338	0	126	0
Array Access	15	0	11	1	160	1	14	0	10	0	106	0
Method Inlining	24	12	117	28	20	9	74	49	27	5	1	0

Figure 14. Number of AST nodes that are rewritten (columns '>0') and rewritten more than once (columns '>1').

show the number of rewrites that occur for type decision chains. Finally, the last row shows the number of methods that are inlined at the AST level.

Figure 15 presents details about the type specialization for arithmetic operations. The *Static* rows show the number of nodes for the respective operation generated during parsing, i.e., the number of operators in the source code. The remaining rows contain dynamic execution counts. Every operation starts in the *Uninitialized* state and is rewritten upon first execution. An *Uninitialized* count that is lower than the corresponding *Static* count implies that some operations are never executed (dead code). Unlike static analysis, we waste no resources identifying the type of never executed operations.

Although JavaScript uses the `double` type to represent numbers at the language level, it is possible to use integer operations as long as no overflow occurs. The execution counts of the *Integer* rows show that using integer arithmetic is a beneficial optimization: 3 out of our 6 benchmarks can be type-specialized to operate almost exclusively on integer numbers. Integer arithmetic is much faster than floating point arithmetic on all current architectures.

The *Generic* rows show operations that have mixed types. Our current implementation reaches this state if the left input is integer and the right input is double, or vice versa. Therefore, the execution counts of this state are high for some benchmarks. To avoid reaching this state, we would have to insert explicit type conversion nodes that convert one input from input to double and then specialize the arithmetic operation to the type double.

	Richards	DeltaBlue	Crypto	RayTrace	Splay	NavierStokes
Addition						
Static	30	25	163	96	23	113
Uninitialized	14	17	100	32	20	83
Integer	654,972	358,140	18,610,735	27,318	8,049	11,364,308
Double	0	0	857	4,802,357	39,306	89,660,148
Generic	1	38,526	1,998	14,848	892,155	1,622,590
Subtraction						
Static	2	4	150	18	4	24
Uninitialized	2	5	107	12	4	23
Integer	127,999	29,820	3,848,603	2	821,437	197,905
Double	127	127	7	3,308,470	63	6,291,481
Generic	0	0	11	153,980	0	9,437,178
Multiply						
Static	3	6	49	50	3	81
Uninitialized	3	6	32	42	3	67
Integer	0	318,333	12,429,155	55,060	0	5,001,179
Double	0	0	34	7,947,147	0	36,707,056
Generic	0	0	1,192	379,002	0	22,594,234
Divide						
Static	4	5	12	15	4	7
Uninitialized	3	5	9	11	4	5
Integer	0	4,222	1	0	0	0
Double	0	0	220	82,431	13,119	126
Generic	1	1	24	1,297,978	1	1

Figure 15. Execution counts for arithmetic operations.

8. Future Work

Our prototype implementation is not yet a complete JavaScript VM, since we focus on the concept and potential of AST rewriting rather language completeness. Two parts missing are `eval()` and regular expressions. The `eval()` statement takes a string as a parameter and executes it as if it were source code inserted at this position. Since the string can be constructed at run time, this allows dynamic code generation. A recursive call of our parser and interpreter would be a straightforward implementation of `eval()`. It parses and executes the string from scratch at every execution, i.e., without saving any type information that was collected by AST rewriting. This approach would be sufficient if the repeated execution of a specific `eval()` statement had a different string parameter every time. However, studies have shown that `eval()` is misused by many developers, e.g., by supplying a constant parameter [20].

Even for valid use cases of `eval()`, we expect that the input parameter is *stable* in the same sense that types are stable: in most cases, a limited amount of different strings is provided.

By combining the idea of type decision chains and method inlining on the AST, we can aggressively optimize such use cases:

- When the `eval()` is executed the first time, the provided source code string is parsed into an AST and executed. We assume we will see this string again, so the tree is rewritten to a string comparison that checks for equality.
- If the string comparison succeeds, the AST cached from the previous execution is executed.
- If a different string is evaluated, the new string is parsed again, and another string comparison node is appended.
- Up to a certain maximum number, the ASTs of strings are cached. These trees are specialized and optimized by multiple executions, so all the optimizations described in this paper are also performed for dynamically generated source code.
- Only if too many different parameters are encountered, the strings need to be parsed every time the `eval()` is called.

For regular expressions, we can use the regular expression library provided by the standard Java library, again assuming that the regular expression itself is stable. This means we can cache one or a few regular expressions to avoid parsing the regular expression every time it is executed.

To achieve the best possible peak performance, frequently executed parts of an application have to be compiled to optimized machine code. In such a mixed-mode execution system, the interpreter is the first level of execution, and JIT compiled code is the second level. Our AST interpreter rewrites and specializes its input, i.e., it collects profiling information during execution. It eliminates the need for a separate profiling phase and allows flexible and language specific profiling; compared to profiling generated bytecode which can only benefit from the existing profiling implemented for these bytecodes. Therefore, we believe that the rewritten ASTs are the ideal input for a JIT compiler. However, we do not want to write a JIT compiler for every language; instead, we want one compiler that is language agnostic. We can achieve that by using partial evaluation: the `execute()` methods of a whole AST are inlined into one compilation unit, thereby assuming that the tree is stable. This compilation unit is then optimized by the JIT compiler. If there is a control path that would change the AST, we remove it from the compiled code and instead replace it with a runtime call that triggers deoptimization [16], i.e., the optimized machine code is discarded and execution continues in the AST interpreter. The AST interpreter rewrites the tree, which can then be compiled and optimized again. This way, we are able to create optimized machine code for an AST that only contains the fast path for every node. An existing implementation of this partial evaluation is PyPy [21]. However, they do not rewrite the AST during interpretation, but instead rely on a trace-based JIT compiler [2, 3] to exploit type stability.

9. Related Work

Williams et al. applied type specialization to the Lua VM interpreter [23]. They use a dynamic intermediate representation (DIR) that replaces the standard Lua bytecode. Each DIR node encodes the opcode of a specialized instruction. Operations whose result type can change are represented with a *type-directed* node that includes a table of target nodes, one entry per each of the 9 types in Lua. The interpreter uses the result type of a node of such an operation to find and dispatch to a type-specialized target node. This is effectively a fixed-size polymorphic inline cache between the specialized DIR nodes. The specialized operations avoid the need for type checking the first input operand of instructions. It is not clear if/how the technique avoids type checking subsequent input operands. The DIR interpreter also propagates types through calls and returns although does not support inlining. The DIR interpreter achieves an average speedup of 1.3x over the standard Lua bytecode interpreter even though the dispatch overhead for DIR is greater than that of bytecode dispatch.

As part of their survey on the design of efficient interpreters [9], Ertl and Gregg offer advice on how to increase the efficiency of an interpreter, especially taking into account the impact of various branch prediction schemes have on indirect dispatch in an interpreter. They advise writing a threaded code interpreter [1] and combining common sequences of instructions into *superinstructions* [19]. Both of these enhancements significantly complicate the design of the interpreter. In contrast, we believe that the simple, tree rewriting AST interpreter we present can not only achieve reasonable interpretation performance but can be used with a special form of JIT compilation (see Section 8) to provide excellent overall program performance.

Brunthaler implemented quickening for a Python bytecode interpreter [4, 5]. Frequently executed bytecodes are re-written to more specific, type-specialized versions. This introduces type feedback for, e.g., arithmetic operations, similar to our system. However, we argue that bytecodes are harder and less flexible to rewrite because a bytecode sequence has to be overwritten with a new sequence of the exact same length. This required Brunthaler to change the original bytecode representation of the Python interpreter.

Our system works on the assumption that the value types in a program are mostly stable, much like those of statically typed languages. Gal et al. show with their trace based compiler for JavaScript that this assumption is well founded [10]. In an attempt to improve the type information available to a dynamic compiler, Hackett et al. added a hybrid type analysis to the JaegerMonkey JIT compiler used in Firefox [13]. Their analysis uses static type inference backed by dynamic checks for cases not accounted for by the static analysis. When such a check fails, the inferred type information is updated, and any code that is subsequently invalidated must be immediately discarded and potentially recompiled (e.g., if it is currently executing). This hybrid type analysis substantially increases compile time (2.5x on the SunSpider benchmark) with only a modest speedup factor of 1.27 on the same benchmark. In our system, which already uses dynamic type feedback, such static type analysis would be far too expensive and provide little or no benefit.

The need for more context-sensitive inlining in a Java VM supporting dynamic languages is highlighted by Click [7] and Häubl [14]. Our use of AST inlining at hot call sites is designed to address exactly this issue.

10. Conclusions

We have shown that through the use of profiling and AST rewriting, an AST interpreter can achieve speeds comparable to language-independent bytecodes. A significant advantage of such a design is its simplicity in comparison to dynamic language systems that use bytecode generation. Additionally, the layered structure of our system allows us to develop the interpreter in pure Java, which offers a whole raft of language and tooling advantages over lower level languages such as C and C++. We believe that our system provides a solid basis for providing overall excellent performance due to the type information that can be made available to a JIT compiler. Such a JIT compiler would use a special form of partial evaluation by inlining all the `execute()` methods of a whole AST into one compilation unit.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs, as well as the Institute for System Software at the Johannes Kepler University Linz, for their support and contributions.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] J. R. Bell. Threaded code. *Communications ACM*, 16(6):370–372, 1973.

[2] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM Press, 2009. doi: 10.1145/1565824.1565827.

[3] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 9:1–9:8. ACM Press, 2011. doi: 10.1145/2069172.2069181.

[4] S. Brunthaler. Efficient interpretation using quickening. In *Proceedings of the Dynamic Languages Symposium*, pages 1–14. ACM Press, 2010. doi: 10.1145/1869631.1869633.

[5] S. Brunthaler. Inline caching meets quickening. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 429–451. Springer-Verlag, 2010. doi: 10.1007/978-3-642-14107-2_21.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 49–70. ACM Press, 1989. doi: 10.1145/74877.74884.

[7] C. Click. Fixing the inlining problem, 2011. URL http://www.azulsystems.com/blog/cliff/2011-04-04-fixing-the-inlining-problem.

[8] ECMA. Standard ECMA-335: Common language infrastructure (CLI), 2012. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[9] M. A. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5, 2003.

[10] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 465–478. ACM Press, 2009.

[11] Google. V8 JavaScript engine, 2012. URL http://code.google.com/p/v8/.

[12] Google. V8 benchmark suite, 2012. URL http://v8.googlecode.com/svn/data/benchmarks/current/run.html.

[13] B. Hackett and S. Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 239–250. ACM Press, 2012. doi: 10.1145/2254064.2254094.

[14] C. Häubl, C. Wimmer, and H. Mössenböck. Evaluation of trace inlining heuristics for Java. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1871–1876. ACM Press, 2012. doi: 10.1145/2245276.2232084.

[15] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.

[16] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992. doi: 10.1145/143095.143114.

[17] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*, 2012. URL http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf.

[18] Mozilla. Rhino JavaScript VM, 2012. URL http://www.mozilla.org/rhino/.

[19] T. A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 322–332. ACM Press, 1995. doi: 10.1145/199448.199526.

[20] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 52–78. Springer-Verlag, 2011. doi: 10.1007/978-3-642-22655-7_4.

[21] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 944–953. ACM Press, 2006. doi: 10.1145/1176617.1176753.

[22] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization*, 4(4):2:1–2:36, Jan. 2008. doi: 10.1145/1328195.1328197.

[23] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 278–287. ACM Press, 2010. doi: 10.1145/1772954.1772993.