A Domain-Specific Language for Building Self-Optimizing AST Interpreters

Christian Humer* Christian Wimmer† Christian Wirth† Andreas Wöß* Thomas Würthinger†

*Institute for System Software, Johannes Kepler University Linz, Austria †Oracle Labs

{christian.humer, woess}@ssw.jku.at {christian.wimmer, christian.wirth, thomas.wuerthinger}@oracle.com

Abstract

Self-optimizing AST interpreters dynamically adapt to the provided input for faster execution. This adaptation includes initial tests of the input, changes to AST nodes, and insertion of guards that ensure assumptions still hold. Such specialization and speculation is essential for the performance of dynamic programming languages such as JavaScript. In traditional procedural and object-oriented programming languages it can be tedious to write self-optimizing AST interpreters, as those languages fail to provide constructs that would specifically support that.

This paper introduces a declarative domain-specific language (DSL) that greatly simplifies writing self-optimizing AST interpreters. The DSL supports specialization of operations based on types of the input and other properties. It can then use these specializations directly or chain them to represent the operation with the minimum amount of code possible. The DSL significantly reduces the complexity of expressing specializations for those interpreters. We use it in our high-performance implementation of JavaScript, where 274 language operations have an average of about 4 and a maximum of 190 specializations. In addition, the DSL is used in implementations of Ruby, Python, R, and Smalltalk.

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

General Terms Algorithms, Languages, Performance

Keywords Java, domain-specific languages, dynamic languages, language implementation, self-optimizing programs

1. Introduction

An abstract syntax tree (AST) interpreter that is compiled and optimized statically must include code to handle any allowed input. Those inputs, however, can often be classified into a few frequently occurring categories, and a large number of infrequently occurring corner cases. Even if in one execution the program only receives input of a single category, the complex code for handling all possible inputs has to be included by a compiler. This prohibits potential optimizations that might apply to only one category of inputs. Runtime feedback is necessary to fill this gap. The interpreter can use this to specialize itself to the observed inputs during each run.

Many AST nodes of the interpreter can be specialized independently. Each operation potentially has a large number of specializations. This requires a disciplined approach that is maintainable and understandable. In this paper, we propose using a domain-specific language (DSL) to specify specializing AST interpreter nodes. The DSL is integrated into the programming language used to develop the interpreter, i.e., it is an internal DSL without a separate syntax.

We call the programming language in which the DSL is integrated the *host language*. For our implementation, the host language is Java, and we map DSL elements to Java annotations for classes and methods. Java annotations are an elegant way to attach metadata to Java code that can be processed both during compilation and at run time. We use a Java annotation processor at compile time to generate additional Java code from the annotations.

The DSL greatly reduces the amount of code that the programmer must write for self-optimizing interpreters. Several research groups use the DSL implementation to optimize dynamic language run-times such as JavaScript, Python, Ruby, R, and Smalltalk. In the evaluation we show how we decreased code size from 3,500 to 1,000 lines of code for parts of our JavaScript implementation while maintaining the same semantics and performance. These interpreters, which contain many and complex specializations, can be defined in a simple and structured manner using the DSL. With our approach we are also able to maintain high-performance.

In summary, this paper contributes the following:

- We define an internal DSL using Java annotations for writing self-optimizing AST interpreters that can specialize to a given input. The language allows declaratively specifying the specializations as well as the triggers to switch between and combine specializations.
- We describe performance optimizations that can be applied more easily with the DSL.
- We show how the DSL can be compiled to high-performance Java code that is also suitable for automatic partial evaluation.
- We evaluate the DSL in the context of our high-performance JavaScript implementation.

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. In contrast, our system, called *Truffle* [31, 32], uses 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 implement an application in, while the *host language* is the language that the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

GPCE'14, September 15-16, 2014, Västerås, Sweden
Copyright 2014 ACM 978-1-4503-3161-6/14/09...$15.00
http://dx.doi.org/10.1145/2658761.2658776

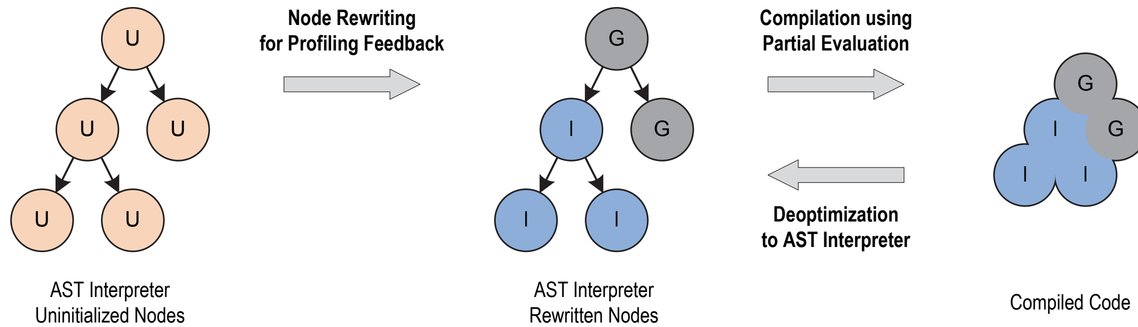

Figure 1. Truffle approach: Self-optimization of the AST interpreter incorporates profiling information into the AST using node rewriting. Compilation using partial evaluation produces high-performance machine code for an AST. Deoptimization reverts execution back to the AST interpreter, so that node rewriting can be performed again.

guest VM is written in, i.e., the language that the host VM executes. For the examples and evaluation in this paper, the guest language is JavaScript and the host language is Java, i.e., we have a guest JavaScript VM implemented in Java.

The guest language implementation is written as an abstract syntax tree (AST) interpreter, which is a simple and intuitive way of describing the semantics of a language. Whether through formal operational semantics or informal specifications, a language's behavior is typically defined by specifying the behavior of its constituent expressions independently. An AST interpreter directly codifies such a language description.

The ability to replace one AST 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 one. 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. The specialized node optimistically makes *assumptions* about its operands for future executions. The typical implementation of the execution semantics (so-called *execute* methods) of a node checks that all assumptions still hold and then performs its specialized operation. If the assumptions do not hold, the node replaces itself with a node that can handle the more generic case. A detailed description of the tree rewriting process and its importance for building high-performance AST interpreters appears elsewhere [31].

The specialized ASTs are used as input for a dynamic compiler that translates frequently executed elements of an application to optimized machine code. The dynamic compiler speculates that the AST is stable, i.e., no node rewriting happens after compilation. It starts compiling the interpreter `execute` method of the root node, and recursively inlines the `execute` methods of all children. This is called partial evaluation, because the compilation uses both the `execute` methods and an actual AST as the input. It results in highly optimized and compact machine code because the compiled code only needs to cover the language semantics of the current specialization. Speculation failures are handled using *deoptimization*: the compiled code is discarded and execution continues in the AST interpreter, which performs node rewriting again. The rewritten AST can be compiled again later on. The separation of the language semantics from the optimization system allows language implementers to create a high-performance language implementation by simply writing an AST interpreter. A detailed description of the dynamic compilation appears elsewhere [32].

Manually writing specialization code is tedious: every specialized node requires code that checks its constraints and rewrites itself to a different node if the constraints are not fulfilled. Our implementation of JavaScript includes nodes with more than a hundred specializations. Having a concise syntax for these specializations that avoids all repetitive boilerplate code is crucial. This paper presents the *Truffle DSL* to express specializations.

The definition and implementation of our DSL, called *Truffle DSL*, is available as open source in an OpenJDK project [26]. The OpenJDK project includes the Java annotations as well as the annotation processor described in this paper. However, the ideas and the principles of our DSL are generally applicable and not depending on Truffle. They are a general way to express specializations, and can be used in any language implementation or any interpreter that performs specializations to achieve high performance.

2.1 Node Rewriting

An AST consists of nodes with an arbitrary number of child nodes. AST interpretation traverses the tree in post-order: the children of a node are executed before the node itself. Intuitively, the children can be seen as the arguments of program elements that need to be evaluated before the element itself can be executed.

During execution, a node can replace itself at its parent with a different node. This allows a node to specialize on a subset of the full operation semantics. If its own handling of the operation cannot succeed for the current operands or system environment, a node replaces itself and lets the new node handle the input. The node replacement depends on the following conditions:

Completeness: Although a node may handle only a subset of the semantics, it must provide rewrites for *all* cases that it does not handle itself.

Finiteness: After a finite number of node replacements, the operation must end up in a state that handles the full semantics without further rewrites. In other words, the tree must reach a stable state.

Before a node is executed the first time, its state is not known; we call this state *uninitialized*. If no specialization fits the input values seen, a node that handles the full semantics of an operation is necessary; we call this state *generic*.

Figure 1 illustrates the Truffle approach on an AST with five nodes. When the program starts executing, all nodes are *uninitialized*, as shown on the left hand side of the figure. Upon the first execution of a node, the first specialization that matches the current input values is chosen, and the node is rewritten to this specialization. Only when on some subsequent execution the current specialization no longer matches the new input values, the node is rewritten again. In the worst case, when all specializations match an input, the node rewrites itself to the generic case. When the AST reaches a stable state, compilation by partial evaluation generates

highly optimized machine code for the AST. In case the AST needs to be rewritten after compilation, deoptimization reverts execution back to the interpreter.

2.2 Expressing Specializations

Separating the full semantics of an operation into multiple specializations has two important benefits:

1. It is good software engineering practice to split complex operations into parts that can be implemented, understood, and tested separately.
2. Small and specialized interpreter execution methods allow the dynamic compiler to produce small and fast machine code.

The implementation of a specialization consists of two parts: First, it checks that all inputs are as expected, and performs node rewriting if an input is not as expected. Second, it performs the actual operation. In other words, the actual operation is *guarded* by checks on the inputs.

For example, assume an operation that can be split into three specializations: `I`, `D`, and `S`. The example is inspired by the specializations for the JavaScript addition operation, which is specialized for the types *integer* (`I`), *double* (`D`), and *string* (`S`). Every specialization can be implemented and guarded independently from the others, and together cover the full semantics. The node starts out in the uninitialized state, and the first execution picks `I`, `D`, or `S` based on the input. If the guard of the selected specialization holds for all future executions, the node is *monomorphic*. This is the optimal case, but not guaranteed to hold.

If the guard of the selected specialization does not hold for a future execution, we cannot just move to a different specialization as this could introduce toggling between two or more states, violating our principle of *finiteness*. Instead, we need a *polymorphic* node that can handle multiple input conditions, i.e., a node that combines the semantics of other nodes. Writing all possible combinations manually is infeasible because the number of combinations increases exponentially with the number of specializations. However, it is simple to chain specializations, i.e., automatically combine multiple specializations and execute the specializations' guards until a match is found. The generic case is a chain of all possible specializations.

In our example, assume that the first node rewriting selected specialization `D`, but later on the type of input changes to `S`. Transitioning to specialization `S` would violate finitess – if the input toggles between `D` and `S`, no stable state can be reached. Instead, the input of type `S` triggers a rewrite to the polymorphic node `DS`, which is a chain of the `D` and `S` specializations. It can handle subsequent inputs of type `D` and `S` without node rewriting.

Chaining of specializations is not necessary when one specialization *contains* another, i.e., handles strictly more inputs as another specialization. In this case, the node transitions into the more general state instead and remains in this specialization. In our example, assume that the specialization `D` contains the specialization `I` ("double addition" contains "integer addition"). A node in specialization `I` that receives an input of type `D` is rewritten to specialization `D`, and remains in this specialization even on subsequent input of type `I`.

In summary, node rewriting uses a state machine with *uninitialized* as the initial state and *generic* as the state that covers all possible inputs. The first transition always moves to a monomorphic specialization. States further down are polymorphic chains of specializations. Figure 2 shows all possible states and transitions for our example. From the uninitialized state `U`, the three specializations `I`, `D`, and `S` are reachable. Since `I` is contained in `D`, there is no state that chains `I` with `D`. The generic state is `DS`. It is reached after at most three node rewrites.

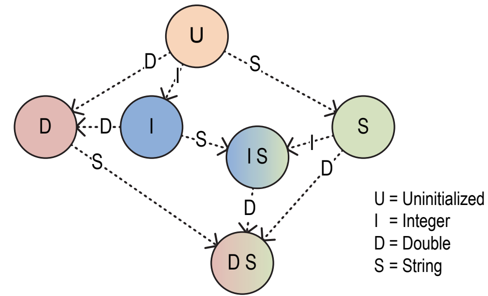

Figure 2. An example node transition graph with all possible states between the uninitialized state `U` and the generic state `DS`.

Writing such a state machine as well as the polymorphic chaining by hand is tedious and error prone. The Truffle DSL provides a declarative way to specify these transitions. Only the monomorphic specializations and their guards need to be written manually. The node transitions and the polymorphic chaining are automatically inferred and optimized. The following sections how the described problem domain can be expressed using Truffle DSL.

3. Truffle DSL

We designed Truffle DSL guided by three main principles:

Declarative: The design of the language is strictly declarative. We want to declare just the specializations and their guards without the complex relationships between them. Ideally a guest language developer can focus on the declaration of fast and simple specializations.

Facilitate Optimization: The DSL makes it easy to write specializations, i.e., it makes it easy to optimize a guest language implementation. The guest language developer can use optimization techniques without having to write much source code.

Interoperability: Although the DSL is designed to be as complete as possible, it is not realistic to cover all use-cases of every guest language AST interpreter. There are always cases that cannot be expressed with the DSL. Therefore, it must be possible to combine manually written Truffle nodes with operations declared using the DSL.

In this section, we describe the basic language elements and illustrate the declarative view of the DSL on AST interpreter nodes. We start with introducing the fundamental elements of the language using the example in Figure 3. The DSL elements are Java annotations on classes and methods, i.e., we use Java language elements to anchor the DSL elements. Classes are used for two different kinds of DSL root elements: type systems and operations.

Java annotations precede class, method, and field definitions, and start with the "`@`" symbol. An annotation can have attributes, represented as key-value pairs where the keys are Java identifiers and the values are constants. Valid types for constants are primitive types, String, class references, other annotations (with attributes), or arrays of any of these types.

Type systems, such as the `GuestLanguageTypes` class in our example, declare an ordered list of types. This list of Java classes is ordered by their concreteness. We will later need this property to estimate the costs of a specialization. The DSL can infer the Java semantics of these types and validate the order for contradictions. For example it is an error if the `Number` type would be declared as a more concrete as the `Integer` type, because in Java `Integer` is a subclass of `Number`. In our example we define four types: `boolean`, `int`, `double` and `String`. None of these types can be considered

```
@TypeSystem(boolean.class, int.class,
            double.class,  String.class)
class GuestLanguageTypes {
 @ImplicitCast double castInt(int value) {
   return value;
 }
}

@TypeSystemReference(GuestLanguageTypes.class)
abstract class BaseNode extends Node {
   abstract Object execute();
}

@NodeChild("child0") ... @NodeChild("childN")
abstract class OperationNode extends BaseNode {
  @Specialization
  Object doSpecialization(Object child0Value, ...,
                          Object childNValue) {
    // Arbitrary Java code.
  }
}
```

Figure 3. Example for the basic language elements of the DSL.

more concrete than the other if we simply look at them from a Java semantics perspective. However types can have different semantics for guest language interpreters. In the examples used in the paper we are going to define `int` as a more concrete version of `double`. In other words they represent the same logical type of the guest language but are represented with different Java types in the interpreter. These relations are modeled using *implicit casts*. They are Java methods annotated with `@ImplicitCast` that take a value of a type in the type system and return a value of another type.

A language implementation usually has one type system, but many operations. Operation roots are modeled using classes that extend the `Node` class. The `Node` class is a part of the Truffle API and represents a node in an AST interpreter implementation. The API of the `Node` class is not relevant for the DSL as a language element, but it is used as convention across Truffle guest languages. The `BaseNode` class in our example links to the type system that is used for the current and all inheriting nodes using the `@TypeSystemReference` annotation. It is convenient but not required to use a base class for operations to share common language elements between nodes. As the execution interface between nodes the `BaseNode` defines one or more abstract `execute` methods. The DSL recognizes such methods by the `execute` prefix and provides an implementation using the operation semantics. In case such a node is written manually these execute methods have to be implemented by hand. In order to enable type check elimination as described in Section 4.1 it is supported to declare multiple execute methods with specialized types. The DSL provides the implementation of these methods automatically.

In our example in Figure 3, the operation node `OperationNode` is derived from the `BaseNode` class. An operation can have an arbitrary number of children which are defined using the `@NodeChild` annotation. The name of such a `@NodeChild` annotation can be declared with the `value` attribute. Even if the name of a child is only used rarely it is considered good practice to give children a name to improve the context of error messages. By default all child nodes have the same base type as the operation node. If a different base type is required it can be specified using the `type` attribute. Each base type must have at least one statically visible `execute` method as well as a referenced type system. Type systems of child nodes must not necessarily match the type systems of the operation.

Figure 4 illustrates the steps that are taken in case a node is executed at run time. The execution of an `OperationNode` is triggered by a parent node *(step 1)*. First, child nodes are executed

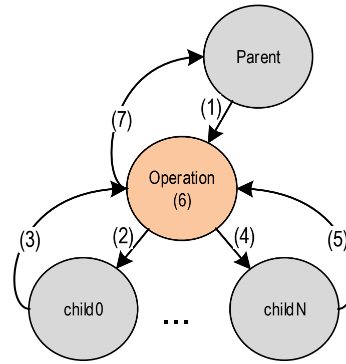

Figure 4. Steps taken when an operation is executed at run time.

and their result values are returned to the operation *(steps 2 to 5)*. The values from the child nodes act as the input values, which are used to match a compatible specialization. Specializations are defined using Java methods annotated by `@Specialization`. Each parameter of such a specialization method represents the value of one particular child in the same order as they are declared. The parameter types are used to restrict the types with which the specialization can be used and therefore restrict its compatibility. In this schematic example the most generic Java type `Object` is used for all parameters and the return value. We call specializations with no restrictions on compatibility *generic*. The implementation of the method body can be done using arbitrary Java code. After a specialization was matched and executed *(step 6)*, the result value of the operation is passed to the parent node *(step 7)*.

3.1 Specializations

This section describes how multiple specializations can be used to optimize operations. A `@Specialization` method represents one monomorphic state of a `Node`. A specialization makes assumptions that are verified using guards. If a guard is invalidated the node re-specializes itself. The DSL provides declarative specifications for different kinds of *guards*:

- *Type guards* optimistically assume the type of a child value. A value that matches the type is cast to its expected type; otherwise the assumption is invalidated by the host language. We model type guards using the parameter type of a specialization method. Type guards are particularly useful to enable type check elimination as described in Section 4.1.
- *Method guards* optimistically assume the return value of a user-defined method to be `true`. Method guards are modelled using Java methods that return a `boolean` value. If the guard method returns `false`, the specialization is no longer applicable and the node is re-specialized. Custom guards are referenced using the `guards` attribute of the `@Specialization` annotation. The implementations of these methods are required to return always the same value for the same combination of input values. This is necessary because they may be executed multiple times or not at all. Multiple guards can be combined using conjunctive normal form (CNF). For example a declaration of the guards attribute as `guards={"guard0 || guard1", "!guard2"}` is equivalent to the boolean expression $(guard0 \lor guard1) \land \neg guard2$.
- *Event guards* trigger re-specialization in case an exception is thrown in the specialization body. The `rewriteOn` attribute of the `@Specialization` annotation can be used to declare a list of such exceptions. Guards of this kind are useful to avoid calculating a value twice when it is used in the guard and its spe-

cialization. A common use-case for such a guard is an overflow check. For overflow checks the result value needs to be computed before the check can be performed. In case an overflow occurs the exception is thrown and the operation is re-specialized. As soon as event guards are triggered another specialization needs to run to complete the operation. Therefore, the specialization must not cause side-effects until the exception is thrown.

Optimistic assumptions restrict a specialization to a subset of the operation semantics. If two specializations intersect in their operation coverage we have to select one. The best one is usually the one with the lowest costs. There is no reliable way to infer the exact costs of a specialization because the costs are strongly dependent on the underlying run-time system. Therefore the DSL uses a heuristic based on guards. The heuristic assumes that a specialization guarded by a more concrete type guard has usually lower costs than one with a more generic type guard. Also, specializations with a higher number of method or event guards are interpreted as to have lower costs. If this heuristic is not good enough then the costs of a specialization can also be specified manually using the `cost` attribute of the `@Specialization` annotation. Experimental evaluation may provide guest language developers with a good numeric value for this attribute. In most cases, however, specializations are not intersecting and thus the order of specializations is not relevant.

In case a particular set of input values is not supported by any specialization of the operation a fall-back handler is used. The fall-back handler represents all cases that are not covered by a declared specialization of the operation. By default such cases are considered an error and the DSL throws an exception at run time. This exception stores input values, child nodes, and other meta-data to enable the construction of a user-friendly error message. It is also possible to customize the behavior of the fall-back handler using a Java method annotated with `@Fallback`. Such a method has to have a generic signature to be compatible with any combination of input values that might arise. If a fall-back case is triggered it is going to implicitly guard for the assumption that no other specialization is used. Otherwise the fall-back specialization might be executed for a case for which other matching specializations are declared. It is recommended to use the fall-back handler only for unexpected or erroneous cases.

3.2 Example

This section shows the Truffle DSL usage for the example introduced in Section 2.1 and Figure 2. Recall that we define the addition operation for the three types *integer*, *double*, and *String*. They are mapped to the Java primitive types `int` and `double`, as well as the standard library object type `String`. Figure 5 shows the complete source code for the operation.

As a binary node, the addition node has two children, i.e., two `@NodeChild` annotations. We name them `left` and `right`. Note that the parameter names of the `@Specialization` methods are `leftValue` and `rightValue` to keep names consistent, but this is not a requirement of the DSL.

The first specialization `doInt` is applicable when both arguments have the primitive type `int`. This *type guard* is automatically inferred from the declared type `int` of both arguments. As long as the addition does not overflow, the result is also of type `int`. The compiler has special knowledge about the `addExact` method of the JDK and compiles this method to only two machine instructions on the fast path: the addition followed by a *jump on overflow* instruction. Only in the unlikely case of an overflow, an `ArithmeticException` is thrown. Because the exception is an *event guard* (expressed as the `rewriteOn` attribute), overflow triggers node rewriting. Using the `@ImplicitCast` from `int` to

```java
@NodeChild("left") @NodeChild("right")
abstract class AddNode extends BaseNode {

  @Specialization(rewriteOn=ArithmeticException.class)
  int doInt(int leftValue, int rightValue) {
    return Math.addExact(leftValue, rightValue);
  }

  @Specialization
  @Contains("doInt")
  double doDouble(double leftValue, double rightValue) {
    return leftValue + rightValue;
  }

  @Specialization(guards = "isString")
  String doString(Object leftValue, Object rightValue) {
    return leftValue.toString() + rightValue.toString();
  }

  boolean isString(Object leftValue, Object rightValue) {
    return leftValue  instanceof String
        || rightValue instanceof String;
  }
}
```

Figure 5. Example showing the usage of the DSL for an addition.

`double` (see Figure 3), the `int` arguments are converted to `double` so that the `doDouble` specialization can perform the addition.

The specialization `doDouble` is compiled to only one machine instruction, but it is still slower than `doInt` on modern processors because floating point arithmetic requires more clock cycles than integer arithmetic. In addition, the more concrete return type of `doInt` can be used for better specializations of the addition's parent node. However, having both the `doInt` and `doDouble` specialization in a polymorphic state would not have any benefit, so the `doDouble` specialization `@Contains` the `doInt` specialization.

String concatenation should be performed if either the left or the right operand is of type `String`, but not necessarily both. Therefore, the signature of the `toString` specialization cannot use the type `String` for the `left` and `right` argument. The *method guard* `isString` checks the precondition of this specialization.

If no specialization matches the input values, a type error is thrown. It is not necessary to write any code for that by hand. In our example type system, a type error is thrown when attempting to add two `boolean` values. Note that these are not the semantics of JavaScript add, which can perform on any type and never throws a type error. Additional specializations with the appropriate type conversions are necessary to implement the full JavaScript semantics, but the code is too large to serve as an example in this paper. In our implementation the full add operation semantics without nodes for type conversion are described using 10 specializations.

3.3 Conditional Child Execution

Operations that represent control flow in a guest language have to conditionally execute child nodes. For example, in most guest languages a *logical-and* binary operation must ensure that the right child is not executed if the left child has already returned a negative value. To model these relationships using the DSL an additional Java method annotated with `@ShortCircuit` has to be declared. The value attribute of the `@ShortCircuit` annotation references a name of a `@NodeChild` in the enclosing operation. The method receives all child node results that are executed before the short circuited child is executed and returns `true` or `false` whether or not it should be executed. Figure 6 shows how such a *logical-and* operation can be specified using the DSL. The `needsRight` method declares a short circuit method, which specifies that the

```
@NodeChild("left") @NodeChild("right")
abstract class LogicalAndNode extends BaseNode {

  @ShortCircuit("right")
  boolean needsRight(boolean leftValue) {
    return leftValue;
  }

  @Specialization
  boolean doBoolean(boolean leftValue,
                    boolean hasRightValue,
                    boolean rightValue) {
    return hasRightValue ? rightValue : false;
  }
}
```

Figure 6. Example for short circuit evaluation via the `ShortCircuit` annotation in a *logical-and* node.

right child is only executed if the left child returns true. To use this information for the operation, all specializations declare a boolean value that specifies whether or not the child value is computed. In our example the `doBoolean` specialization returns the `right` value if the `right` value is computed and otherwise it can return always `false`. The `@ShortCircuit` annotation can be used for multiple or all node children at the same time. Complex control flow operations like loops are not directly supported by the DSL. However loops are simple to implement with the Truffle API and do not require many specializations.

The `executeWith` attribute for `@NodeChild` annotations allows a node to pass the result of one child's executable as an input to another child's executable. Figure 7 shown an example. The parent node `P` has two children: child `s` computes an important intermediate result `v` that child `c` also needs for execution. A simple but wrong solution, shown on the left hand side of the figure, is to reference child `s` directly from child `c`. This is probably inefficient, because result `v` is computed twice. In the worst case, when the execution of child `s` has side effects, the second execution returns a different and wrong result.

We solve this problem by passing the result `v` to child `c`, as specified via the attribute `executeWith=s` on child `c`. The references in `executeWith` are limited to children that are declared before, since children are always executed in the order they are defined. The two child nodes of node `P` would declared with `@NodeChild("s")` and `@NodeChild(value="c", executeWith="s")`. In JavaScript we use this pattern to implement calls on element accesses.

We found that the `executeWith` attribute is an essential building block with a diverse set of use cases. It eliminates the need for complex workarounds involving temporary heap objects that store result values required by multiple nodes. In addition, it allows the guest language implementer to split a complex operation into a tree of multiple nodes and pass values around these tree nodes. Every node in the tree can then be specialized separately, which avoids the exponential complexity of specializing a single node in multiple dimensions. A concrete use case for this feature is our implementation the JavaScript code `array[0]()`. In our implementation the outer function call evaluates the value of `array` and passes it to the inner element access.

4. Optimization

We identified several Truffle optimization techniques crucial to interpreter performance. The Truffle DSL enables the guest language developer to exploit these techniques easily. The following sections highlight the most important techniques that we use.

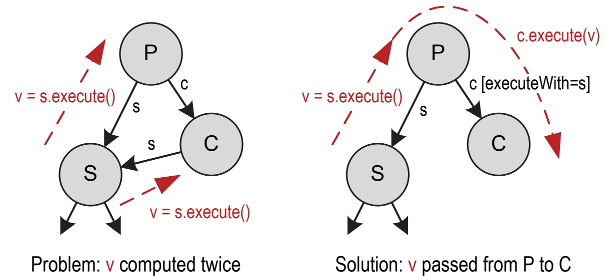

Figure 7. Example for the `executeWith` attribute.

4.1 Type Check Elimination

In order to avoid type checking overhead in the context of the static type system of Java, a node may ask a child node to return its data with a specific static type. To achieve this, it calls an `execute` method on the child node with this static type as the return value. The called node then has the choice to either return its produced value as of this static type, or throw an exception that encapsulates the value as a generic `Object` instance. The parent node must be prepared to catch such an exception and then replace itself with a different node that executes the child with a more generic type.

This system is particularly effective when the involved data has a primitive type, in which case both the type check and boxing overhead is avoided. Instead, a primitive value is directly transmitted from the child to the parent node. In the example of Figure 3, the primitive Java types `int` and `double` can be optimized with a declaration of a `executeInt` and `executeDouble` method in the `BaseNode` class.

The support of the DSL for type check elimination optimization is twofold. First, type specialized execute methods are automatically implemented if they are declared as non-final methods in the operation. Second, if type guards are used for specializations then the DSL tries to use specialized execute methods for monomorphic and polymorphic cases. As soon as a specialized execute method fails once with an unexpected type the DSL ensures that an execute method with a more generic type is used.

4.2 Specialization Reduction

As soon as an operation gets polymorphic it is worthwhile to evaluate if a single more generic specialization leads to better code than building a chain of monomorphic specializations. In Figure 5 we show how the `@Contains` annotation is used to specify that the `doDouble` specialization contains the `doInt` specialization. This way the DSL determines that all *integer* specializations can be removed from the polymorphic chain as soon as the *double* specialization is used once.

The `@Contains` annotation used with a single specialization is already a powerful tool that enables the guest language developer to control polymorphism. Determining when an how rewrites in the polymorphic chain are advantageous is not trivial. For example a more generic specialization might only be beneficial if two or three specializations out of a set of all specializations are chained together. To cope with these cases we reuse the numerical `cost` attribute of the `@Specialization` annotation that we introduced in Section 3.1. The `@Contains` annotation is able to reference multiple specializations. To decide if the DSL replaces these specializations with the more generic version we sum all costs of the contained and used specializations. If this sum is greater than the costs of the more generic specialization we perform the replacement otherwise we leave the polymorphic chain intact. In summary, the `@Contains` annotation together with the `cost` attribute is used

by guest language developers to tune the polymorphic behavior of operations based on empirical measurements.

4.3 Guard Reduction

Guard reduction is another opportunity to optimize polymorphic specialization chains. As the polymorphic chain of specializations is traversed, the type and method guards are executed to determine if the specialization is compatible with a set of input values. We represent these guards with constant boolean values if we derive their value using the previous execution of specializations and their guards. Since it is not feasible for arbitrary Java guard methods to prove that they imply other guard methods, we introduce an `@Implies` annotation to specify that relationship. Like the `guards` attribute in `@Specialization` annotations, the `implies` expression is specified in CNF and can therefore contain any boolean expression. To catch programmer errors in complex expressions, guards are not just replaced with constant values, but also Java assertions are added that check that the eliminated guards hold. Java assertions, if turned off, cause almost no overhead compared to code without assertions.

5. DSL Implementation

The DSL defined in the previous sections must be embedded in a host language. For our implementation, Truffle DSL, the host language is Java. Since we do not want to change the syntax of Java, the DSL is implemented using Java annotations on types and methods. Annotations provide a flexible way to augment Java elements with metadata that can be processed both during compilation and execution. We use compile-time processing to generate additional Java code during compilation using a Java annotation processor [20].

Our code generator is tightly integrated with the source compiler and development environment for the host language. First, the annotated source code is parsed by the compiler. The compiler triggers the code generator if an element containing metadata is encountered. The generator accesses the metadata and the code structure to produce new code. Finally, the compiler also compiles this newly generated code. This enables using the same development tools and integrated development environment (e.g., Eclipse or NetBeans) to browse the generated code. The generated Java code is also available during debugging.

Figure 8 summarizes the development process: Our Java annotations are used in the source code (step 1). When the Java compiler is invoked on the source code (step 2), it sees the annotations and calls the annotation processor (our DSL implementation; step 3). The annotation processor iterates over all of our annotations (step 4) and generates Java code for the specialized nodes (step 5). After that the annotation processor notifies the Java compiler about the newly generated code, so it is compiled as well (step 6). The result is the executable code that combines the manually written and the automatically generated Java code (step 7).

5.1 Annotation Processor

Our annotation processor operates as part of the Java compilation process; it generates additional Java code, which is compiled during the same compilation cycle. The processor needs to generate at least one class for each specialization. This means that the amount code scales linearly with the number of specializations that are used. The exact contents of the generated files are out of scope for this paper; Wuerthinger et al. [32] describes the rationale behind the generated classes and methods in more detail.

We evaluated and discarded alternatives to generating Java source code: Instead of processing the annotations at compile time, they could be processed at run time. At run time, it would be possible to directly generate Java bytecode that can be loaded into the Java VM without requiring compilation. The bytecodes would contain the same classes and fields we define in Java code. This approach has two main disadvantages: first, Java bytecode generation is cumbersome and error-prone (even when using supportive libraries such as ASM [7]); second, the generated bytecode cannot be debugged at the source level in standard IDEs such as Eclipse or NetBeans.

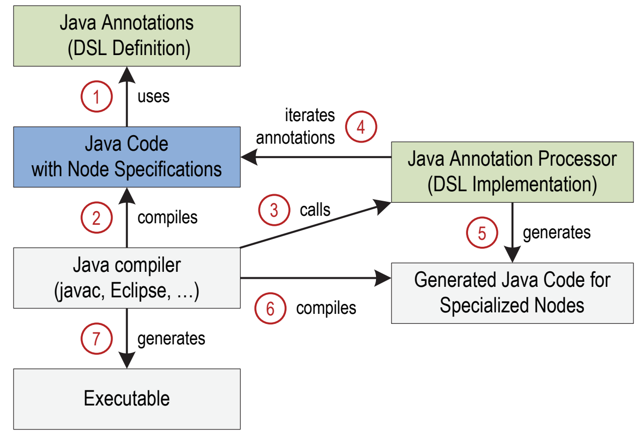

Figure 8. Steps in the development and build process.

Another alternative approach would be to use reflection to call the annotated methods, without needing to generate any code at all. However, performance would suffer since the Java VM cannot optimize reflective method calls as much as normal method calls. In addition, reflective calls are difficult to debug in IDEs.

6. Evaluation

This section evaluates the adoption of Truffle DSL in a Truffle-based JavaScript interpreter. We show that using the DSL instead of hand-written classes a) significantly reduces the amount of code while retaining or even improving execution performance and b) simplifies writing more and better specializations, further boosting performance.

6.1 Initial Conversion to Truffle DSL

We started the adoption of Truffle DSL with a early prototype implementation of JavaScript. This prototype implemented 29 basic JavaScript operations, each of which had up to 6 specializations. Overall, this resulted in 129 classes implementing the functionality of those operations. We have rebuilt these operations using Truffle DSL while preserving the specializations from the original implementation. Figure 9 illustrates the number of types, methods, and lines of code that could be reduced by using Truffle DSL. The number of programmer-written classes could be significantly reduced from 129 to 29 (one for each operation), and the lines of code dropped from more than 3500 to less than 1000.

The runtime performance of JavaScript is crucial to its success. Therefore the adoption of the Truffle DSL must not lead to performance regressions. Figure 10 shows the performance of JavaScript running octane benchmarks[19] before and after the adoption of Truffle DSL in the early prototype. The benchmarks were executed on an Intel Core i7 2620M CPU with 4 cores at 2.70 GHz, 8 GB memory, JDK 1.7.0-13 64Bit (2 GB fixed initial heap), running Windows 7 64 bit. On each benchmark it achieved a speedup compared to the previous hand-written implementation. For the *crypto* benchmark the speedup is even 20%. This speedup was not anticipated from a one-to-one translation and does not stem from more or better specializations. It is achieved mostly through additional

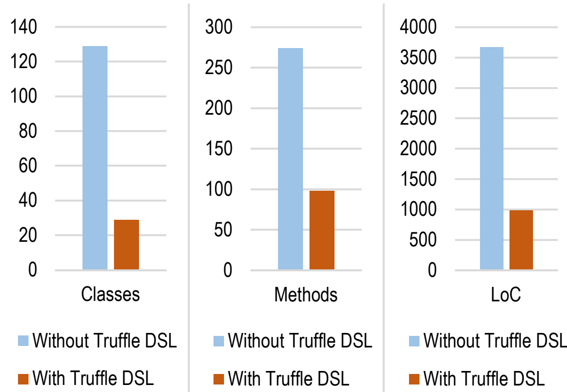

Figure 9. Code metrics for our JavaScript implementation before and after the adoption of Truffle DSL (lower is better).

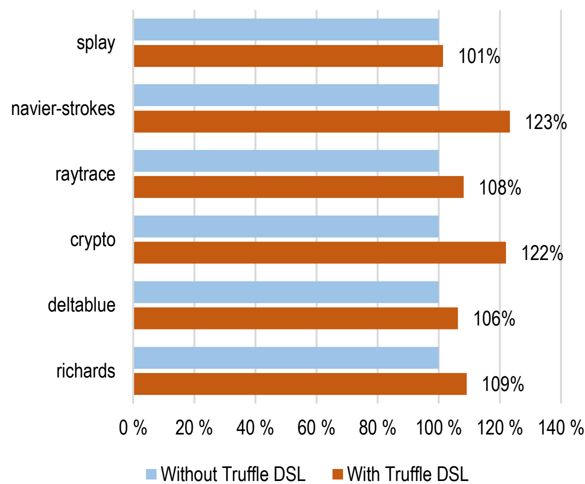

Figure 10. Performance of our JavaScript implementation before and after the one-to-one translation to Truffle DSL (higher is better).

type check elimination, cf. Section 4.1. In the original implementation the programmer had to write the necessary boilerplate code for type check elimination, which was missing in some cases.

6.2 Current use of Truffle DSL

Figure 11 shows detailed code metrics of the Truffle JavaScript implementation at the time of writing. After the adoption study described above was finished, we evolved the prototype to a complete implementation of JavaScript. The metrics are grouped into language operations and builtin functions. Language operations are constructs like *add*, *call*, or *read*. Builtin functions are core API functions like *Math.abs*, *Array.push* or *parseInt*. The implementation contains 274 node classes with a total of 1,024 specializations. In total, these nodes are declared using 16,833 lines of code which generate 157,404 additional lines of code using the annotation processor. The JavaScript implementation uses nodes with up to 190 specializations. We argue that implementing them manually, i.e., writing 190 complete classes with rewriting logic instead of 190 annotated methods, is overly tedious. With the DSL it is possible to quickly experiment with new specializations and their attributes to test their impact on performance. While achieving good performance the generated code increases the memory footprint compared to a reflective implementation of the DSL. The amount of code that is generated is similar to the amount of a manually written implementation. We expect the specialization count to still rise as the Truffle JavaScript implementation becomes fully optimized. However, we try to achieve best peak performance with the lowest number of specializations possible.

	Language Operations	Builtin Functions	Total
Nodes	68	206	274
with $s = 1$	8	109	117
with $s = 2$	15	45	60
with $2 < s \leq 5$	20	46	66
with $s > 5$	25	6	31
s: *number of specializations per node*			
Specializations	615	409	1,024
Average per node	9.04	1.99	3.74
Maximum per node	190	12	190
Truffle DSL Java Code			
Classes	70	257	327
Methods	1,090	1,073	2,163
Lines of Code	7,914	8,919	16,833
Generated Java Code			
Classes	994	1,118	2,112
Methods	8,700	7,042	15,742
Lines of Code	100,919	56,485	157,404

Figure 11. Code metrics of our current JavaScript implementation (with more specializations than the original adoption of the Truffle DSL).

Figure 12 shows the current peak performance evaluation of the Truffle JavaScript engine compared with the version 3.26.31 of the V8 engine [18] using various Octane benchmarks [19]. It shows that the DSL has been useful when building a JavaScript engine that is competitive performance-wise with other state of the art engines. The benchmarks were executed on an Intel Xeon E5-2690 v1 with 8 cores, 2 threads per core, at 2.90 GHz, 96 GB memory, running Ubuntu Linux 12.04.3 (kernel version 3.5.0). The Java heap size was fixed to 64 GByte for all benchmarks and all benchmark files were in an in-memory file system. All results for Truffle JS are based on revision `73a0c8e14cd1` of Truffle[1].

Truffle DSL is currently adopted by at least five Truffle language runtimes. Besides the JavaScript implementation described in this paper, Truffle DSL is used in the open-source Python implementation ZipPy [33], the open-source R implementation FastR [15], in a Smalltalk interpreter [4], and in an experimental interpreter for JRuby [9]. They show on different use-cases similarly promising results as Truffle JavaScript. Therefore, we believe that Truffle DSL is beneficial for more than one guest language.

We are currently looking into other language paradigms e.g., functional programming, that may provide us with new challenges. This may require additions to or modifications of the DSL.

7. Related Work

In the following we present work related to implementing an optimizing AST interpreter based on a DSL. Work related to Truffle has been presented in earlier papers regarding Truffle [31, 32].

Specializing Program Execution Previous research has shown how specializing program execution based on dynamic properties

[1] http://hg.openjdk.java.net/graal/graal/rev/73a0c8e14cd1

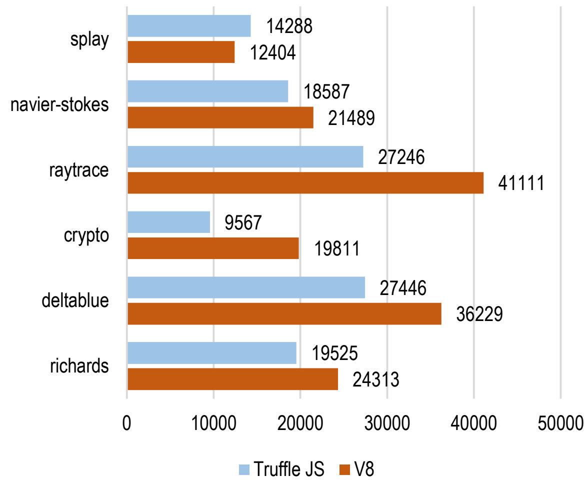

Figure 12. Truffle JS peak performance comparison with the V8 engine using octane benchmarks (higher is better).

of the system can improve system performance. For instance, in a work by McNamee et al. [24] this technique is used to specialize operating system code depending on so-called specialization predicates. For bytecodes, techniques like *quickening* [8] allow rewriting the bytecodes to improve execution performance.

To simplify implementation of a specializing system, approaches like the one presented by Volanschi et al. [29] have been developed to declaratively specify the specialization process. A similar approach is taken in this paper, where a DSL allows expressing under what conditions AST nodes are replaced. To that end, we use predicates as described in Ernst et al. [14], implemented as type checks on parameters and guard functions.

Dynamic Language Compilers: Dynamically typed languages like JavaScript and Ruby benefit greatly from specializing code for the types actually observed during execution. The original idea stems from an optimizing compiler for the language SELF [11]. The TraceMonkey project for Mozilla's SpiderMonkey JavaScript interpreter reports a speedup of 2x-20x depending on the benchmark [16]. They combine trace compilation with type specialization to achieve that speedup. In contrast to our approach, their approach is implemented in the host language of the interpreter, without support by a DSL.

Using DSLs to Improve Software Development: Domain-specific languages can support the implementation of an interpreter or compiler for yet another language. JColtrane is an annotation-based approach to write a SAX-style XML parser. The authors report on a lower complexity for writing a SAX-style parser, but significantly less efficient code (almost three times slower than the regular SAX) [25].

Scala is language particularly suitable as host language for DSLs. Its multi-paradigm approach and features such as implicits, structural types, or closures make it a powerful language for designing internal DSLs [17]. Using parser combinators, Scala also supports the definition of external DSLs. A technique called language virtualization [10] has been suggested to solve the problem of applying domain-specific optimizations in the context of parallel execution. It provides a heterogeneous model for parallel programming based on DSLs implemented in one host language, e.g., Scala. A prerequisite for that approach is lightweight modular stagging [28]. This technique provides the power of multi-stage programming languages, but can be implemented in a lightweight manner as a library, uses only types instead of annotations to distinguish between binding times, provides optimizations like common subexpression elimination or value numbering, and finally enables further optimizations like language virtualization.

Language-oriented programming [30] is an approach to use DSLs to optimize the software development process. For instance, Cedalion [23] is a host language that can be used to specify internal DSLs. Cedalion features static validation and projectional editing. It uses internal DSLs, but evades some of the drawbacks, e.g., interoperability between languages or a lack of tooling.

Other similar language workbenches exist, e.g., the Meta Programming System (MPS) [2] or the Spoofax system [21]. In contrast to Cedalion, they target external DSLs but provide techniques for interoperability between the DSLs. While MPS comes with its own IDE, Spoofax integrates with the Eclipse IDE. In contrast to other approaches, Spoofax allows dynamic loading, i.e., development of the language and development with the language in the same IDE, a feature also supported by our system.

Expressing DSLs as Java Annotations: Java annotations and the Java annotation processor as defined in JSR 269 [20] are used by several projects to implement internal DSLs.

In Dietl et al. [12], the Checker Framework [27] is used to create type checks for a host language with a DSL. In particular, using a DSL expressed as Java annotations, the type checks for Java source code are enhanced by several non-standard methods like nullness checks, fake enumeration checkers, or canonicalizing checkers. The authors argue that such a DSL represents a sweet spot for program analysis, balancing expressiveness and comprehensibility.

The Project Lombok [1] uses Java annotations to remove boilerplate code from Java projects. For instance, the annotation `@Getter` can be added to a field to automatically generate a getter function in the background. With only a few annotations, large amounts of boilerplate code can be avoided. More advanced features, e.g., providing extension methods for Java, are still experimental.

The juast library [22] uses an JSR 269 annotation processor to strengthen code quality. Their paper remains unclear about what specifications the user has to provide to achieve that goal.

Frameworks for DSLs: Xtext [3], part of the Eclipse project, is a framework for defining external DSLs in close cooperation with the Eclipse IDE. The language can itself be enriched by using additional DSLs. Bettini [5] presents the DSL XTypeS for defining a type system for languages written in the Xtext system. While the type system within Xtext is usually implemented in Java, using a DSL allows simpler, easier to understand formalisms that are stronger related to the actual formalization of the language. XTypeS allows defining rules that act on the elements of the AST of a program written in an Xtext language. The rules are compiled to Java classes and are automatically executed by the Xtext framework based on their method signatures. A further improvement of that approach is the DSL Xsemantics [6]. It provides a richer syntax for rules, has more advanced features, e.g., closures, and targets all kinds of rules, not only a type system.

Another DSL based on Xtext is the expression language Xbase [13]. The language is statically typed and tightly integrated with the Java type system. It can be used in any Xtext-defined language to specify the behavior via expressions in a concise way.

8. Conclusions

In this paper we presented a DSL that greatly simplifies writing self-optimizing interpreters. Using the declared monomorphic specializations, a state machine for all node rewriting transitions can be derived. The DSL enables applying simple and intuitive optimization techniques that would otherwise be infeasible to implement. The DSL also allows the guest language developer to focus

on experimenting with different approaches. In addition to demonstrating that the DSL can significantly simplify the implementation of self-optimizing interpreters, we also presented measurements to show the efficiency of the generated source code. We hope to see even more adoption of our DSL to other dynamic language implementations as well as other self-optimizing interpreters.

Acknowledgments

We thank the members of the Institute for System Software at the Johannes Kepler University Linz, and the Virtual Machine Research Group at Oracle Labs for their support and contributions. We especially thank Peter Kessler, Michael Van De Vanter, and Adam Welc for feedback on this paper.

The authors from Johannes Kepler University are funded in part by a research grant from Oracle.

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

References

[1] Project Lombok. URL http://projectlombok.org/.

[2] Meta programming system. URL http://www.jetbrains.com/mps.

[3] Xtext. URL http://www.eclipse.org/Xtext/.

[4] TruffleSOM: A file-based Smalltalk implementation built with truffle., 2014. URL https://github.com/smarr/TruffleSOM.

[5] L. Bettini. A DSL for writing type systems for Xtext languages. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, pages 31–40. ACM Press, 2011.

[6] L. Bettini. Implementing Java-like languages in Xtext with Xsemantics. In *Proceedings of the ACM Symposium on Applied Computing*, pages 1559–1564. ACM Press, 2013.

[7] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002.

[8] S. Brunthaler. Efficient interpretation using quickening. In *Proceedings of the Dynamic Languages Symposium*, pages 1–14. ACM Press, 2010.

[9] M. L. V. D. V. C. Seaton and M. Haupt. Debugging at full speed. In *Proceedings of the 8th Workshop on Dynamic Languages and Applications (DYLA)*, 2014.

[10] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language virtualization for heterogeneous parallel computing. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 835–847. ACM Press, 2010.

[11] 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.

[12] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. W. Schiller. Building and using pluggable type-checkers. In *Proceedings of the International Conference on Software Engineering*, pages 681–690. ACM Press, 2011.

[13] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: implementing domain-specific languages for Java. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 112–121. ACM Press, 2012.

[14] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 186–211, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL http://dl.acm.org/citation.cfm?id=646155.679688.

[15] FastR. FastR is an implementation of the R language in Java atop Truffle and Graal., 2014. URL https://bitbucket.org/allr/fastr.

[16] 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.

[17] D. Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 9781935182450.

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

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

[20] JSR 269. JSR 269: Pluggable Annotation Processing API, 2011. URL http://www.jcp.org/en/jsr/detail?id=269.

[21] L. C. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 444–463. ACM Press, 2010.

[22] A. Klepinin and A. Melentyev. Integration of semantic verifiers into Java language compilers. *Automatic Control and Computer Sciences*, 45(7):408–412, 2011. .

[23] D. H. Lorenz and B. Rosenan. Cedalion: a language for language oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 733–752. ACM Press, 2011.

[24] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Trans. Comput. Syst.*, 19(2):217–251, May 2001. ISSN 0734-2071.

[25] R. Nuccitelli, E. Guerra, and C. Fernandes. Parsing XML documents in Java using annotations. In *XML, Associated Technologies and Applications*, 2010.

[26] OpenJDK. Graal project, 2013. URL http://openjdk.java.net/projects/graal.

[27] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 201–212. ACM Press, 2008.

[28] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 127–136. ACM Press, 2010.

[29] E. N. Volanschi, C. Consel, G. Muller, and C. Cowan. Declarative specialization of object-oriented programs. *SIGPLAN Not.*, 32(10): 286–300, Oct. 1997. ISSN 0362-1340.

[30] M. P. Ward. Language oriented programming. *Software-Concepts and Tools*, 15:147–161, 1995.

[31] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th symposium on Dynamic languages*, DLS '12, pages 73–82. ACM Press, 2012.

[32] 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 Onward!* ACM Press, 2013.

[33] W. Zhang, P. Larsen, S. Brunthaler, and M. Franz. Accelerating iterators in optimizing ast interpreters. In *Submitted to OOPSLA*, 2014.