École Polytechnique Fédérale de Lausanne

Operation DSL
Making bytecode interpreters a walk in the park

by Nikola Bebić

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Christian Humer
External Expert

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 10, 2022

Abstract

When writing a programming language interpreter, some implementations opt for a tree-walk interpreter approach, making the implementation much more natural and easier to maintain. However, tree-walk interpreters bring numerous performance-related problems – so most high-performance interpreters transition to a bytecode-based approach. Writing a bytecode interpreter requires significantly more effort since individual bytecode instructions are more low-level than tree nodes. This thesis aims to bring the simplicities of tree-walk interpreters to the bytecode world, explore the optimizations that can be applied, simplify the creation of new bytecode interprets, and transition existing tree-walk interpreters.

We present a new method for specifying bytecode interpreters, abstracting away the stack manipulation, control flow, and instruction stream. By specifying our language as a series of *Operations*, we can generate a highly performant bytecode interpreter, with all the optimizations included, while keeping the implementation complexity closer to that of a simple tree-walk interpreter. We will detail the design of the Operation DSL, analyze the optimizations that can be applied, and shortly detail the current implementation details. Lastly, we will evaluate the DSL's usefulness by comparing the generated bytecode interpreter to a tree-walk and a manually written bytecode interpreter.

Contents

Chapter 1

Introduction

1.1 Motivation

Creating a dynamic language runtime usually includes writing a custom interpreter for the language. As an alternative, the *Truffle framework* [11] allows reusing the existing benefits of the *host GraalVM* runtime and compiler for other, *guest* languages, such as JavaScript, Python and Ruby. That simplifies writing the language interpreters, as the Graal compiler takes care of native performance, and Truffle appropriately prepares the language to be just-in-time compiled through *partial evaluation*.

Most of the Truffle languages are implemented as tree-walk interpreters. Structuring the interpreters in such a way make them even more straightforward, as one needs to specify only the behaviour of individual operations independently of each other. This approach also makes the interpreters more natural to implement, as the code structure is closer to how those operations are specified.

The language implementation constructs these *abstract syntax trees* (ASTs) from the source program. They are either interpreted directly, using tree-walking or fed into a dynamic compiler that produces *GraalVM intermediate representation* graphs [5]. These can then be just-in-time compiled into native machine code. The compilation process uses partial evaluation [12] of the interpreter, so the language semantics does not need to be duplicated between the interpreter and the dynamic compiler. Instead, the dynamically compiled code is obtained as the 1st Futamura projection [7] of the interpreter. This process assumes the program structure is constant and allows Truffle to constant-fold the interpreter, obtaining the compiled function code.

Tree-walk interpreters in Truffle have several disadvantages over bytecode-based ones, such as a larger memory footprint. However, their simplicity and malleability made them a good candidate for implementation [14]. Furthermore, bytecode interpreters make it harder to encode

runtime profiling information (called specializations) necessary for fast interpretation and good dynamic compilation – the bytecode array has to be modified as the program executes, which may involve repositioning the code inside it if instructions change length.

Truffle DSL [8] simplified the process of writing tree-walk interpreters further by handling type profiling and dispatch code. Instead of the language implementors having to handle these manually, the code for keeping track of and validating *specializations* is generated automatically from a Java-embedded DSL. The code generation also opens the door to performing some optimizations that would be prohibitive to write and maintain manually, such as boxing elimination.

However, Truffle tree-walk interpreters have problems that cannot be optimized away, such as slow non-local control flow or the large memory footprint of individual node objects. For this reason, multiple attempts were made to incorporate the bytecode interpreter into dynamic Truffle languages, such as *Graalsqueak* [9] or *GraalPython*. These, however, still use tree nodes internally for handling specialization and profiling, which removes some benefits of bytecode interpreters and loses a lot of optimization potential. Even so, creating and maintaining a bytecode format and translating the source code to the individual instructions is still much more effort than AST interpreters require.

Another approach to writing bytecode interpreters in Truffle would be to disregard the Node-based API completely and implement all the existing functionality (such as specializations, branch profiling, and monomorphization) directly in the bytecode interpreter. That would further complicate the problem mentioned earlier since now specializations, boxing elimination, and profiling all have to be taken care of by hand.

However, just as Truffle DSL simplified creating tree-walk interpreters, creating a domain-specific language for generating bytecode interpreters would remove the burden of manual implementation. Furthermore, since now the DSL controls the dispatch of instructions, we can automate additional optimizations, such as super-instruction detection. That is the route chosen by the *Operation DSL*.

1.2 Goals

This thesis aims to define a domain-specific language for generating bytecode interpreters from a high-level declarative specification of language semantics, similar to tree-walk interpreters. The transition from a tree-walk interpreter to a bytecode one should be as simple as possible in terms of code and conceptual shifts. Additionally, we want to keep the specifications compatible and reusable to ease migration from tree-walk to bytecode-based interpreters and reduce the amount of code needing to be rewritten.

The generated bytecode interpreters should fit the existing Truffle API, which relies heavily

on tree-walk interpretation. They should be able to be dynamically compiled, instrumented, and compiled into native images, just like regular AST nodes can.

Since the language is defined using a more declarative approach, we will focus on some optimizations this enables. Some aim to improve interpreter performance, such as boxing elimination, top-of-stack elimination, and quickening. Others aim to reduce the interpreter's memory footprint, such as reparsing. The optimization process should also be automated and require as little as possible human intervention – we will describe corpus-guided optimization and how it is used to drive certain optimization decisions.

Another essential requirement of the DSL and the generated interpreters is future extensibility. The API should be simple enough that it does not prohibit future optimizations. This requirement, in turn, means that the interpreter's bytecode, stack, and other internal workings must be hidden away from users to allow future changes. It also requires the design of operations to be as simple and atomic as possible to allow later optimizations.

1.3 Overview

The Operation DSL is a part of the Truffle DSL, which is a part of the Truffle Framework, which is a part of the GraalVM project. GraalVM is a Java virtual machine based on the Graal compiler for high-performance Java code execution. Coupled with Truffle and the Polyglot framework, it can be used for high-performance interpretation of many languages [13], including interoperation between them.

Truffle is a language implementation framework that runs on top of GraalVM. It uses exposed Graal APIs to compile implemented guest languages dynamically. Using profiling information, Truffle can produce high-performance native code. If the assumptions made using the profiling information are violated, it can revert to interpretation and recompilation of code. Furthermore, it uses partial evaluation (the first Futamura projection) of the interpreter code to produce the compiled code, meaning that the guest language does not need to implement the compiler manually.

Polyglot is a component of Truffle and GraalVM that allows multiple languages to coexist and interact with each other using the *Interop API*. This allows multiple unrelated languages to be compiled together in one compilation unit, interact with each other's data, and call each other's functions like they were their own.

Truffle DSL is a part of the Truffle framework that automatically generates high-performance profiling, caching, and specialization code for the tree-walk interpreter. It dramatically simplifies writing and maintaining tree-walk interpreters and performs some optimizations that would be prohibitively expensive to implement by hand. It is a Java-embedded DSL using the annotation processor mechanism of the Java compiler. The DSL can generate Java code, taking manually

written Java code as input.

Operation DSL is an extension to the Truffle DSL, adding the ability to create bytecode interpreters. It integrates with specialization, caching, and profiling features of the Truffle Framework and the Truffle DSL but removes the AST Nodes from the implementation, instead relying on bytecode generation and interpretation. It also uses Java annotation processor functionality for code generation.

1.4 Results

The main result of this thesis is the Operation DSL itself. The design is detailed in chapter 3 and chapter 4 of this thesis, including the conceptual design of operations, constraints placed on their definition, design of built-in operations provided by the DSL, and the bytecode format. Not all optimizations planned were implemented – the ones implemented are detailed in chapter 3, while the rest are left as future work.

The success of the goals mentioned earlier was evaluated by implementing and adding a new Python interpreter to GraalPython and comparing it to the existing ones – one based on Node DSL and a manually written bytecode interpreter. The interpreters' performance was also compared, but since the Operation DSL-based interpreter is not yet fully-featured, these are considered preliminary.

Chapter 2

Background

2.1 Tree-walk and bytecode interpreters

The two main ways of implementing language interpreters are *tree-walk interpreters* and *bytecode interpreters*.

Tree-walk interpreters, also called abstract syntax tree (AST) interpreters, are based on walking along the program AST structure and executing each node encountered in the tree. The visit is often performed using recursive virtual calls to the `execute` method that evaluates the entire subtree rooted at that node and returns the result, if any.

Bytecode interpreters instead compile down to *bytecode*: a flat array of bytes representing a program using a virtual instruction set. These instructions are much higher-level than native machine ones (or even JVM) and often map closely to language structures. These instructions get executed using a *bytecode loop*, which does a fast switch-based lookup to identify and execute the current operation before moving on to the next in the stream. Since the bytecode is usually generated per function, the bytecode loop is exited when the function returns or an unhandled exception is raised.

Both tree-walk and bytecode-based interpreters operate on the concept of dispatch – selecting which operation to evaluate next based on the program structure and evaluating it using the results of previous operations. In tree-walk interpreters, this is performed using a dispatch based on the concrete type of the tree node. In bytecode-based interpreters, this dispatch is performed based on the virtual instruction at the current bytecode index.

For both variants, we can *quicken* certain operations. Quickening entails replacing one operation with another, which usually handles a subset of possible values or a more restrictive set of preconditions but performs the operation quicker. In tree-walk interpreters, this is performed by *node replacement*, and in bytecode-based interpreters, by changing the instruction stream,

replacing one instruction with another. In both cases, the target of the dispatch changes. How this is implemented in Operation DSL is detailed in section 3.7.

A significant difference between the tree-walk and bytecode interpreters is in control flow handling. In tree-walk interpreters, control flow is handled implicitly by the implementation: a code in the guest language can be skipped by not calling its corresponding `execute` method; or repeated by calling the `execute` method multiple times in a loop. In bytecode interpreters, special *branch* instructions must be generated that explicitly transfer control within the bytecode stream, either unconditionally or based on a value (*conditional branch*). Loops can be made by branching backwards to already executed code. The branch destinations are encoded as a part of the instruction, using either branch instruction relative or absolute offsets.

Another difference is in value passing. In tree-walk interpreters, values are passed between operations using host-level arguments and return values, which are usually mapped to hardware processor registers. In bytecode-based interpreters, values are passed on heap-allocated arrays, either in the form of *virtual registers* or the *virtual stack*. When using registers, each instruction explicitly encodes the source and destination registers. When using a stack, instructions pop the arguments from it and push the results back. The later approach removes the need to encode the source/destination registers in the instructions. Stack-based virtual machines usually provide register-like storage for *locals*, with dedicated read and write instructions.

Because of their complexity and repetitive nature, bytecode interpreters are often targets of automatic generation. Multiple bytecode generators exist, such as VMGen [6] and YARV [10]. These take as input a high-level description of the interpreter and produce the interpreter code as output. Using an interpreter generator makes the interpreter easier to maintain since the generated code is derived directly from the high-level description.

2.2 Java embedded DSLs

Domain specific languages (DSLs), compared to general purpose languages, allow more concise expression of concepts in a particular domain for which they are designed. This means that the structure of DSL code maps nicely to the problem domain; thus, less code is needed to represent certain problem domain concepts.

Embedded DSLs are DSLs that, instead of taking an entire textual source file with a custom syntax as input, are "embedded" into another language, usually a general purpose one. There are multiple ways to embed a language, and they vary based on the capabilities of the host language – either the language has extensible (or general enough) syntax or provides a method of introspecting the code so that the DSL can perform its processing.

Java annotation processor is an interface exposed by the Java compiler that allows custom code to be executed when a particular *annotation* is encountered in the source code. This code

can then inspect the annotated construct, and perform some actions based on that, including generating more Java code. This allows for simple implementation of *embedded DSLs* in Java.

Instead of parsing textual sources, the DSL takes the structure of the annotated Java construct (including any enclosed elements) as input. The semantics of the DSL are then encoded in the Java program structure itself, possibly using other annotations.

The DSL output can then be additional Java source files that will be compiled along with the original sources. Manually written code can depend on the generated code, and the Java compiler will gracefully handle the circular dependencies.

While the annotation processor mechanism is inherent to Java, similar mechanisms can be employed in other languages, such as using annotations in DotNET. Alternatively, meta-programming facilities such as decorators in Python can be employed to either generate code at runtime or statically. Thus, the DSL design presented here is not tied to the Java language and can be ported to other systems or even used with a non-embedded DSL.

2.3 Truffle language implementation framework

In Truffle, the execution of a language is centred around `Nodes` which represent the *abstract syntax tree* (AST) of the language. The nodes then invoke each other through virtual *execute* methods. The nodes are always organized in a tree through a series of *parent-child* relationships, rooted in a *root node*, which usually represents a language-level function. The constructed tree of nodes usually maps closely to the structure of the language itself.

An essential aspect of Truffle is *guest compilation*, the ability to just-in-time compile guest language functions into native code for performance. To do this, Truffle uses *partial evaluation*. For partial evaluation, Truffle assumes specific properties of the AST are constant (e.g. the structure of the AST) and folds them away. This way, all the virtually dispatched child calls can be inlined, and the entire guest-level function becomes a single native function. We can also add additional assumptions to this partial evaluation process, allowing us to specialize the generated code further.

If those assumptions become false at any point (e.g., we assumed addition is integer-only but got floating point numbers), we can *deoptimize* the code. A deoptimization invalidates the compiled code and transfers the control flow to the interpreter, allowing us to handle unexpected cases. It is essential, however, for the language implementation to have an upper limit on the number of times this can happen since Truffle expects the compiled code to stabilize after some time. Otherwise, the program enters a *deoptimization loop*, where the same code gets compiled and deoptimized repeatedly, slowing down compilation.

Truffle heavily relies on the tree structure of the code. Some framework features, such as

source locations and instrumentation, assume the nodes are granular enough to represent individual statements, expressions, and concepts in the guest language. If this is not observed, those features may not work as expected. This requirement complicates the implementation of bytecode interpreters, which contain only a single node for the entire function that internally executes the bytecode loop. The Truffle framework has already been expanded to allow for such interpreters and accommodate existing bytecode interpreters (such as the *Espresso* JVM interpreter), but most of the public-facing APIs assume a node-based interpreter.

2.4 Truffle DSL

The node-based *Truffle DSL* (referred to as *Node DSL*) simplifies the process of writing tree-walk interpreters by automatically handling the tree nodes' specialization, invalidation, and re-specialization. It generates code that implements the node execution from a description of the operation. The language only needs to specify the possible specializations and their preconditions (called *guards*), which the generated code will dynamically dispatch based on the preconditions and argument types. It also enables inline caching to speed up lookup-based operations (such as property accesses and function calls) by caching the results for faster access, assuming the object's dynamic type remains the same.

The code is generated from a Java class containing the specialization methods, while the guards, replacement orders, and similar information are specified using `@Specialization` annotations of those methods. The generated code will extend that class and implement the abstract `execute` method. The generated `execute` method will implement the specialization, dispatch and re-specialization logic.

The parameters of the specialization method define the types of values that the particular specialization will handle. Additional parameters can be annotated with `@Cached` and related annotations to provide *inline caches*. The caches will only be executed once the specialization is first selected, and the value will be reused for further invocations. This value can be used to save the results of expensive computations, such as attribute and method lookups. The guards should still be used to validate that the caches are valid at every subsequent invocation.

Listing 2.1, taken with modifications from [8] demonstrates a simple usage of the DSL. It defines an `AddNode` that contains three specializations: `doInt`, which only handles integers, `doDouble` which handles doubles, and `doString` which handles arbitrary objects, but only if one of them is a string, which is checked using a custom guard. The `doInt` specialization is replaced if the `AritimeticException` is thrown.

Truffle languages already use the DSL extensively to define their nodes, which made the ability to reuse the definitions from Node DSL in Operation DSL a high priority. Reusing the same method structure and annotations would simplify the adoption and allow code reuse during the transition period. In addition, it would allow reusing the existing code generation features of the

```
abstract class AddNode extends Node {
  abstract Object execute(Object left, Object right);

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

  @Specialization
  double doDouble(double left, double right) {
    return left + right;
  }

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

  boolean isString(Object a, Object b) {
    return a instanceof String || b instanceof String;
  }
}
```

Listing 2.1: An `AddNode` defined using the node-based Truffle DSL

Node DSL – guard and type checks, caching, and specialization replacement does not need to be implemented from scratch.

Chapter 3

Design

The Operation DSL is a DSL for describing programming languages, using a high-level description of their *operations*. From this description, it can generate high-performance bytecode interpreters. The interpreters are self-optimizing, statically and at runtime, and can be just-in-time compiled using Truffle partial evaluation.

Operation DSL is composed of three main parts (Figure 3.1): the DSL itself, the generated *code builder* and *bytecode interpreter*, and the runtime support classes distributed as a part of the Truffle Framework. The DSL is the compile-time component which parses the descriptions embedded using annotations in Java and generates the code builder and bytecode interpreter. The code builder is the component that the language implementation uses to describe the functions parsed from the source code and obtain executable functions, which the bytecode interpreter then executes.

The DSL is used to specify language semantics by defining *operations*. These, coupled with some *built-in operations*, can then be used to represent user code. The operations are modelled as n-ary (possibly variadic) functions. For example, a language may decide to model its equality semantics as an `Equals` operation that takes two arguments and returns the result of the equality comparison. More complex aspects of language semantics can be broken down into multiple operations and combined during parsing. Details of defining custom operations are given in section 3.1

We will use structured expressions (S-expressions) to write operations in text. For example, the Python code shown in Listing 3.1 can be represented as the S-expression shown in Listing 3.2. This expression corresponds to the tree structure shown in Figure 3.2.

Some operations, such as `IfThen` and `ConstObject`, are common to many languages and are built into the DSL directly. Some others, such as `Equals` and `Call`, have highly language-specific semantics, and the language implementation must define them.

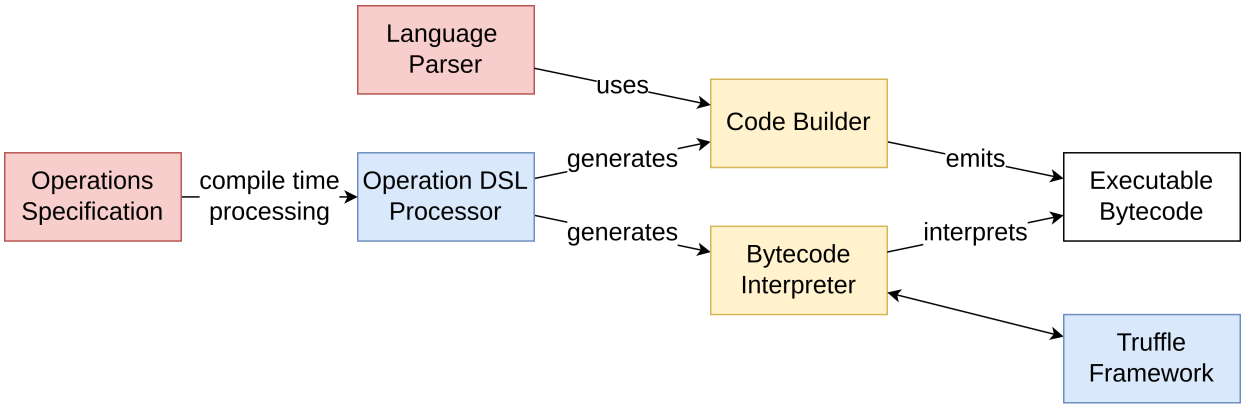

Figure 3.1: General overview of the Operation DSL components. The language implementation provides the red components, the blue components are provided by the Truffle framework and the Operation DSL, and the yellow components are generated at compilation time by the Operation DSL processor.

```
1  if 1 == 2:
2      print("what")
```

Listing 3.1: Example of Python code

```
1  (IfThen
2    (Equals
3      (ConstObject 1)
4      (ConstObject 2))
5    (Block
6      (Call
7        (LoadGlobal "print")
8        (ConstObject "what"))))
```

Listing 3.2: Example Python code converted into Operations, represented as S-expressions

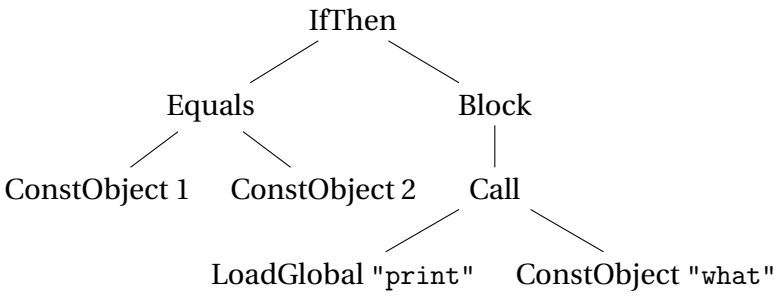

Figure 3.2: The tree-like structure of operations representing the example Python code.

The entry point for the DSL is a single top-level Java class, annotated with a single `@GenerateOperations` annotation. This class will be used as a blueprint to create the bytecode interpreter, and other annotations and members nested within will define the specifics.

3.1 Defining custom operations

The *custom operations* are all those that are not built-in and are needed by the language to express its semantics fully. These cover the basic operations over the values, such as arithmetic and logical operations, member access, and function invocation. These were considered too different among languages and could not be included in the DSL.

Like all operations, custom operations are modelled as possibly variadic n-ary functions. In the example from a previous chapter, one such operation is `Equals`, which models the Python equality semantics. It is a binary operation, taking precisely two arguments and returning a result. Another example was the `Call` operations, which is variadic (taking one call target and zero or more call arguments).

One of the goals of Operation DSL was to keep the existing semantics of the Node DSL, such as `@Specialization` and `@Cached`. For this reason, the design of the DSL closely reassembles the way nodes were defined: each operation is defined as a Java class, with methods annotated using `@Specialization` defining the different semantics of the operation depending on types, guards, and profiling information. Additionally, both the Node and Operation DSL must be able to coexist with minimal code duplication, allowing smoother transition and experimentation.

Operations can be defined in two ways: inline inside the main `GenerateOperations` class, or by *proxying* existing Node DSL operations. The latter allows reusing existing implemented language semantics, lowering the transition effort, and allowing code sharing between tree-walk and bytecode interpreters, if they are developed in parallel. The former is practical for operations-specific behaviour or for using benefits of Operation DSL that are unavailable in Node DSL.

Defining operations inline entails creating a nested `static final` class inside the main `GenerateOperations` class and annotating it with `@Operation`. Inside, multiple methods annotated with `@Specialization` can be defined, specifying the semantics of the operation, as with the Node DSL. An example of defining the `Equals` method can be found in Listing 3.3.

When defining operations by proxying existing nodes, one simply needs to add corresponding `@OperationProxy` annotations to the `GenerateOperations` class, as shown in Listing 3.4. The node must still respect all the requirements of the Operation DSL, such as specializations being `static`. This requirement usually means converting all child nodes to `@Cached` parameters and passing all constructor arguments as literal values to the specializations. The specializations will be called directly, so if the node implements any custom execution behaviour, it may not

```java
@GenerateOperations
final class ExampleOperations {
  @Operation
  static final class Equals {
    @Specialization
    static boolean doInt(int lhs, int rhs) {
      return lhs == rhs;
    }
    @Specialization
    static boolean doString(String lhs, String rhs) {
      return lhs.equals(rhs);
    }
    // other specializations
  }
  // other operations
}
```

Listing 3.3: Example definition of the `Equals` operation.

```java
@GenerateOperations
@OperationProxy(AddNode.class)
@OperationProxy(SubtractNode.class)
// ...
final class ExampleOperations {
  // other operations ...
}
```

Listing 3.4: Example of proxying existing nodes to operations.

execute correctly.

Sometimes, a Node instance is required to implement some language semantics, e.g., for obtaining source locations or referencing correct thread-local data in a multi-threaded environment. In this case, the operation can use the `@Bind("this")` parameter to obtain it. The Operation DSL will pass in the bytecode loop node it uses internally. The Node DSL will also understand this parameter and pass the actual Node instance. This way, Operation DSL and Node DSL interpreters can coexist and share code, with the semantics of both being compatible. Similarly, the bytecode index can be obtained to obtain exact source locations.

Variadic operations

Using the inline definition allows for some extensions that the Node DSL does not support. One such extension is the `@Variadic` argument definition. This argument defines a variadic operation

```java
@Operation
static final Call {
  @Specialization(/* ... */)
  static Object doDirect(
    Object callTarget,
    @Variadic Object[] arguments,
    @Cached("...") DirectCallNode callNode
  ) {
    return callNode.call(arguments);
  }
  // ...
}
```

Listing 3.5: Example of defining a variadic `Call` operation, elided for brevity

– it now only requires a minimum number of arguments (the number of arguments before the `@Variadic`) and will merge all the remaining ones into an array. The type of such argument must be a Java array type.

Variadic operations are instrumental in describing semantics which can take a variable number of arguments, such as method invocations or list literals. In the Node DSL, these would have been represented using a `@Children` annotated array field, which had to be manually executed.

The ability to have a single built-in variadic operation that returns all its children as an array was considered. However, such an implementation would likely have needed more instructions and would be hard to optimize later. Thus, it was decided that having explicit variadic arguments was a better choice.

An example of defining a variadic `Call` node, heavily elided for brevity, can be seen in Listing 3.5. The operation can later be used like shown in the example Listing 3.2, with all the arguments after merging the first in a single (possibly empty) `arguments` array argument.

Short-circuiting operations

Short-circuiting is the behaviour of some language elements which do not evaluate all their arguments if one of them satisfies a particular condition. Most commonly this is used for operations such as logic operators `and` and `or` (or `&&` and `||` in C-like languages). These operations return the first value that satisfies a particular condition. In Operation DSL, we provide a special `@ShortCircuitOperation` annotation that can be used to create these.

First, language needs to define a *boolean conversion* operation – a one-argument operation that returns a truth value of its arguments (in regards to the short-circuiting operation).

```
value_1 = evaluate(child_1)
if booleanConvert(value_1) != continueWhen:
  return value_1
# ...
value_{n-1} = evaluate(child_{n-1})
if booleanConvert(value_{n-1}) != continueWhen:
  return value_{n-1}
return evaluate(child_n)
```

Listing 3.6: Pseudocode implementation of a short-circuiting operation, as interpreted by the Operation DSL. `continueWhen` and `booleanConvert` are the arguments of the operation itself.

```
@GenerateOperations
@ShortCircuitOperation(
  name = "And",
  booleanConverter = ToBooleanNode.class,
  continueWhen = true)
final class ExampleOperations {
  // ...
}
```

Listing 3.7: Example of defining a short-circiting `And` operation.

Usually, this is the language's version of the "cast to boolean" operation but does not need to be[1]. Additionally, we define whether a `true` or `false` value should continue execution, or return the last evaluated value. In either case, the last value is returned by default if no other value is successful. A pseudocode implementation of this behaviour can be seen in Listing 3.6. An example of defining the `And` operation can be seen in Listing 3.7.

The defined short-circuiting operations are always variadic (requiring at least one argument). They will be compiled to efficient bytecode, more efficient than if the short-circuiting has been manually desugared to local assignments and conditionals (explained in section 4.1).

3.2 Builtin operations

Even though we allow the guest languages to express a large portion of their semantics through custom operations, there are some actions for which we define *built-in operations*. These would either be impossible to implement as custom operations, give a suboptimal performance, or be very complex to implement. This section will briefly examine those and why we deem them necessary in the built-in operations list.

[1] One example would be when defining a null-coalescing operation the boolean conversion will be a null check.

Control flow

Custom operations are limited when it comes to controlling execution order – they all evaluate their arguments in order, most of them eagerly (except the short-circuiting ones). This restriction simplifies conversion to bytecode instructions but makes it impossible to model structures such as `if` and `while` statements. Additionally, non-local control flow (such as one created by `continue` and `break` statements) is also impossible to model. For this reason, we introduce a set of operations that closely mimic the common structures found in programming languages based on control structures in the Java programming language.

The `if` construct in Java is modeled using the `IfThen` and `IfThenElse` operations (which model the version without and with the `else` branch). These, along with all the other conditional operations, require the "condition" argument to return a primitive boolean, usually requiring the language to emit an additional boolean conversion operation. Additionally, the `Conditional` operation is given as a value-returning version of the `IfThenElse` – used to implement conditional expressions in guest languages.

For loops, a single `While` operation is available. It models a simple `while` loop in Java. More complex loops, such as iterating and counted `for` loops, can be implemented by converting them into a while loop in the code generator.

All of these operations take a single operation as their "body". A `Block` operation can be used for modelling multi-statement blocks, which can take any number of child operations. It takes care to *pop* unused values off of the stack and returns the value of its last child if any.

In addition to these structured control flow operations, some languages require support for unstructured control flow (even if only to support `break`ing out of loops). For this reason, we also have a *label* system that can be used to mark specific locations and branch them from other parts of the code. The location where the label is created is essential, as we do not allow "branching in" into other operations (as this might skip some initialization), only "branching out" or "branching sideways". This restriction should not be a problem for languages that only intend to implement simple control flow operations but might pose a problem for languages that have unrestricted `goto`s. The labels can be defined using `Label` and branched to using `Branch`.

For returning from functions, a `Return` operation must be used. Since, in Truffle, all functions return exactly one value, this operation also requires a value to return. Additionally, support for `yield`-like coroutine operation is planned to simplify implementing coroutines and generators in guest languages.

```
«init»
(While
  «cond»
  (Block
    «body...»
    «step»))
```

Listing 3.8: Desuggaring a simple `for («init»; «cond»; «step») { «body...» }` loop into provided built-in operations. If a language supports coercing values to booleans, an additional boolean conversion is required around the condition.

Locals and arguments

Locals are another primitive in Operation DSL. They are created and cleared automatically and are scoped to enclosing operation. Operations `LoadLocal` and `StoreLocal` are used to interact with them. The locals are represented using opaque `OperationLocal` objects, which allows the code generator to choose how to allocate and organize locals freely. Additionally, it is planned to automatically clear locals once they leave scope.

Arguments to a function can be manually loaded since they are passed in the `Frame` object and can be read directly. However, for convenience, a `LoadArgument` operation is still provided. It also encodes the argument index as an immediate value instead of storing it as a constant like custom operations would need to do.

Constant values

As mentioned, all constants used in a function are stored in a *constants array*. To read values from that array, a `ConstObject` operation must be used. This operation will always return its argument, stored in the constant array during function execution. This operation is the only one that can load arbitrary constants and must be used to pass more complex immediate arguments to custom operations.

Desugaring

Builtin operations do not correspond directly with the semantics of guest languages, but the language parser can perform *desugaring* from language constructs into those provided by the Operation DSL. For example, a `for` loop would be desugared as shown in Listing 3.8. Additional custom operations may need to be defined to support this (e.g., the boolean conversion operation).

Performing control flow using built-ins means that the language does not have to handle the

```
ExampleOperationsBuilder b;
// ...
b.beginIfThen();
  b.beginEquals();
    b.emitConstObject(1);
    b.emitConstObject(2);
  b.endEquals();
  b.beginBlock();
    b.beginCall();
      b.emitLoadGlobal("print");
      b.emitConstObject("what");
    b.endCall();
  b.endBlock();
b.endIfThen();
```

Listing 3.9: Sequence of builder calls (indented for readability) that would produce the operations given in Listing 3.2, implementing the `if 1 == 2: print("what")` example..

associated logic in the bytecode, such as jump offset calculation and branch profiling. These are all handled by the DSL and can be optimized automatically. Additionally, by deliberately forcing the language to desugar the control flow to built-ins, performing some optimizations becomes simpler since the build-in operations are transparent to the optimizer. Alternatively, we would have to treat all control handling operations as black boxes, prohibiting or complicating some optimizations, such as super-instruction detection (which must not cross control flow basic blocks).

3.3 Operation code builder

Since the bytecode array is deliberately hidden from the language implementation, there needs to be a way to emit them. Emitting them is the role of the *code builder*. The code builder is generated automatically by the Operation DSL and represents the entry point (and only publicly visible generated API) of the DSL.

The code builder API works on the level of operations: the language would translate its syntactical elements into operations and then use the `begin`/`end` calls of the builder to represent them. For example, the operations from Listing 3.2 would be represented using the sequence of calls given in Listing 3.9. The `begin` calls start a particular operation, while the `end` calls end it, after emitting all the "arguments" in order. It is important to nest them correctly, and the builder will raise an exception for wrong nesting. The `emit` calls are used for operations which do not take any arguments, and as such do not need a `begin`/`end` pair. Any "immediate" arguments are given as the arguments to those functions.

The builder does not have a publicly visible constructor. To instead start the building process, a static `create` method must be called, with a *parsing callback*. This callback is a function that takes the builder instance and should perform the necessary calls to the builder to represent the current translation unit (usually an entire source file). After each function is translated, a `publish()` method must be called. It will return the created `OperationNode` instance. However, this function can be called multiple times (as explained in section 3.4), so it should not perform any side effects and must be deterministic in its parsing.

The Operation DSL was made as simple as possible to transition from tree-based to operation-based API. The design of the `begin`/`end` API is intended to be simple to use from the common *AST visitor* pattern that many languages use to produce code. Instead of producing individual nodes for each element, the language should call the `begin` functions while descending the tree and `end` functions while ascending.

Because the API does not use objects to represent operations, the number of allocations while parsing can be drastically reduced, making the parsing process faster. Furthermore, since the Operation DSL has control over the control flow of the function (since all control flow operations are built-in and known), we can perform some static optimizations over the code, such as *dead code elimination*, or finding *super-instructions*.

3.4 Reparsing

One observation is that most metadata (source line numbers, instrumentation information) is often unused during program execution. However, it is still essential to be present when needed. In order to lower memory usage of the runtime code representation, we employ a tactic called *reparsing*.

Instead of saving the metadata, we will only save the parsing callback given to the API during the first node creation. Then, during parsing, we will discard and not initialize any metadata given to the API, so the resulting nodes will not contain any unneeded metadata. If the metadata is requested (e.g., by calling the `getSourceSection` on a node), we will trigger the parsing again, saving the requested metadata for future requests.

All functions created using the same builder instance will be batched together to simplify the reparsing process and share common metadata such as source objects and instrumentation tags. If any function from the batch requests the reparse, they will all be reparsed together. Node batching is also done to reduce repeated work – currently, it is impossible to reparse just one function from the source since functions must be parsed in the same order every time. Different ways to parallelize this process, which may allow us to change the parsed functions' order, were also considered but are currently left as future work.

The language can also configure which metadata gets eagerly stored and which is lazily

reparsed using configuration parameters during the initial parse. This configuration parameter should be used if a language knows it will need particular metadata for its regular execution, skipping an immediate double-parse.

3.5 Boxing elimination

Boxing elimination is an optimization applied to some dynamic languages. Instead of allocating primitive objects (such as integers and floating point numbers) on the heap (also known as *boxing*), we keep them as primitives, reducing the number of allocations. This optimization can be applied to the stack (for stack-based virtual machines) and the locals. This optimization is relatively simple for static languages, as we know ahead of time what types will be on the stack and in locals and can keep the primitive-typed ones unboxed. For dynamic languages, the types can change, and we can observe both primitive and non-primitive values in the same stack slot / local variable at different points during program execution. Any conversion between boxed and unboxed representation of a primitive value takes time, so we want to limit the number of those conversions. In the best case, we would only use unboxed primitives, but if the value would get boxed in the end, we should keep it like that all the way through.

So far, the Node DSL performed boxing elimination by introducing *primitive execute methods*, that returned unboxed primitives. Instead of producing a non-primitive value (or a different kind of primitive), they would raise a `UnexpectedResultException`, triggering a deoptimization. This exception is always followed by a new specialization activating, meaning that maximum the number of deoptimizations remains bounded. If a parent node could potentially expect a primitive argument, it would call into the child's primitive execute method and use that value. If the child raised the exception, it would exclude that child from primitive optimization, meaning that in the future, the generic execute method would be called instead, and no boxing elimination would occur.

One of the guiding principles while designing the Operation DSL was to keep as much as possible of the existing functionality of the Node DSL, so by design, we reuse as much of the existing code generation, modifying it through *hooks*. In this case, this required replacing the child node calls with stack reads. In order to stay consistent with the node terminology, we will still call the instruction that produced a value "child" while the instruction that is consuming it a "parent". There are two main issues here:

- knowing in the "parent" operation whether the child pushed a boxed or an unboxed primitive, and
- knowing in the "child" whether the parent expects a boxed or unboxed value.

Synchronizing this information between the parent and child requires interaction between

possibly distant operations. For this reason we use *child pointers* – the parent instruction stores a relative offset to the child instruction.

In the DSL code generation, a *boxing split* is a group of specializations that all take the same primitive arguments. We say that the boxing split is *active* if only the specializations of that boxing split are active. In that case, primitive arguments can be passed as primitives to it. Otherwise, if no specializations are active at all or active specializations belong to multiple boxing splits, we will pass all arguments as boxed values. From the Operation DSL, we use this to our advantage. Whenever specializations change, we check if a boxing split became active/inactive. If it is activated, we notify the "child" instructions that they should start pushing primitive values. If it is deactivated, we notify them that they should start pushing boxed values again. Since we cannot activate a second boxing split after one has been deactivated, we will never run into a deoptimization cycle here.

Each instruction that takes values off the stack keeps an offset to the instruction that pushed that value – we call this offset the *child pointer*. If multiple instructions could have pushed that value, or if the child is out of range (since only one byte is used for the offset), we will store an invalid offset 0 instead.

When notifying a child instruction that it should start pushing boxed/unboxed values, we will use this child pointer to find out where the child is and what instruction it is. Based on the child opcode, we can perform one of two operations: replace the child opcode completely, based on the primitive type we are performing boxing elimination on, or set or reset a bit in one of the instructions' bitset arguments. If the child pointer is invalid, no boxing elimination will occur, and all values will be passed boxed.

A similar mechanism is used for local boxing elimination as well. The process starts with the `LoadLocal` instruction getting boxing-eliminated using the previously described mechanism. Then, it will mark the local as being boxing eliminated as well. This will then propagate to all the `StoreLocal` instructions once they get executed, which will boxing-eliminate their "child" instructions. The upside of this design is that it does not need any additional logic executed during boxing elimination checks, at the cost of needing multiple program executions until it stabilizes. Nevertheless, since boxing elimination is not needed during compilation, it does not impact compilation stability.

3.6 Corpus-guided optimizations

Some optimizations, such as quickening (section 3.7) and super-instructions, require knowledge about the typical structure of programs written in the language, something we do not get just by defining the possible operations. We need their relative frequencies, common patterns, and similar structural information. One way of solving this problem is providing an API where the guest language implementation can specify this information manually, which is error-prone and

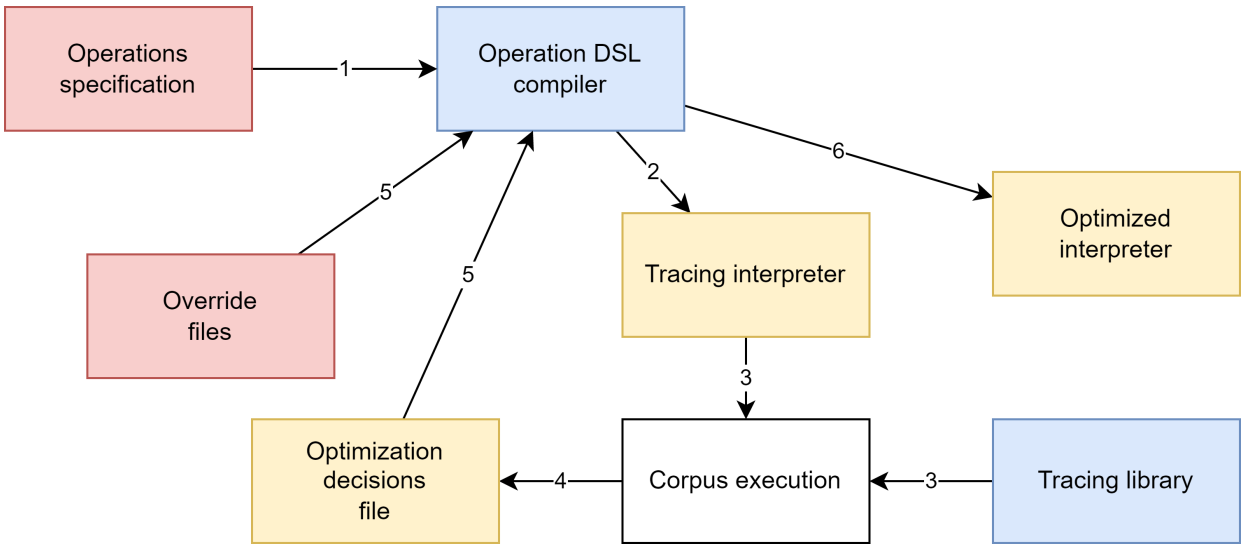

Figure 3.3: Overview of corpus tracing process. The Operations specified by the language are compiled (1) using the Operations DSL, to produce a version of the interpreter with tracing support (2). The corpus is then executed (3), and the tracing library produces a *decisions file*. Along with user provided *override files*, this is used by the Operations DSL (5) to produce an optimized interpreter (6).

complex to create and maintain. Instead, we opted for an alternative solution – *corpus tracing*.

A corpus is a body of code representative of the language in question. It should be a representative sample of all programs written in that language regarding the static and runtime properties of the code, such as code structure, number of operations, and runtime value types. Poor corpus choices, such as micro benchmarks and language test suites, may skew the results and over-optimize for otherwise rare code patterns such as numeric calculations. The methods to choose and evaluate the fitness of a corpus are not examined here and are left for future work.

The corpus is executed using a modified version of the interpreter that, in addition to executing the code, traces the executed instructions, their arguments and internal interpreter state, such as specialization bits (Figure 3.3). This modified version is generated automatically when corresponding compilation flags are enabled and is not required when compiling for release versions of the interpreter – meaning that the entire system has no runtime cost when not tracing.

During program execution, the interpreter will keep track of *state* – the data it needs for optimization decisions, such as the most common instruction sequences (for super-instructions) and common specialization combinations (for quickening). This data is collected and processed immediately, and we only store the aggregated results. This method has the benefit of not creating large trace files but restricts us to only using local heuristics. If more advanced optimizations become required, creating trace files and processing them in a second step may be needed. The state can also be persisted to a file, allowing the corpus to consist of multiple independent

programs run in series.

After every corpus run, the interpreter will automatically write a *decisions file*. This file contains instructions for the compiler on which optimizations should be included when optimizing the interpreter based on the corpus statistics. This file is a JSON file and can be manually inspected, checked in, and version controlled.

The DSL compiler then uses the decision file to drive the optimization step, which emits new or modifies existing instructions. In addition, *decision override files* can be specified, which are manually written files in a similar format to the main decisions file, intended to allow further fine-tuning of characteristics not picked up by the tracer.

3.7 Quickening

Rewriting instructions at runtime to their more specific versions based on runtime profiling feedback is known as *quickening*. This process includes changing the instruction opcode from the generic version to a more specialized one.

The quickening optimization fits nicely in the already present specialization-based dispatch of the Truffle DSL. We define a *quickened instruction* as a version of the instruction with the fixed active specialization. This way, we can skip multiple checks during the instruction dispatch, including the active bitset check, some boxing elimination checks, and any non-included specializations.

During specialization of the generic instruction, we check if the new active specialization set matches any quickened instruction, and if so, replace the opcode of the currently active one. The remainder of the instruction has the same layout and length, so no additional modifications are needed. If the dispatch of the quickened instruction fails, we will fall back to the generic version of the instruction. While executing the generic instruction, it is possible to quicken again to a different quickened variant that accepts more inputs.

As an example, we can take the `Equals` operation defined previously in Listing 3.3. It has two specializations: `doInt` which only compares integers, and `doString` which only compares strings. We will assume that we want to quicken the first one. In that case, our interpreter will have two instructions: a generic one and a quickened integer-only one. When emitting code, only the generic instruction is placed in the bytecode.

We will specialize based on the argument types when executing the generic instruction for the first time. If the `doInt` specialization is chosen, since it is the only active one, we will change the opcode of the instruction to the quickened one. If the `doString` specialization is chosen, since it is not quickened, the opcode will stay generic.

When the quickened variant is executing, and the types do not match (for example, we got strings instead of integers), the instruction will re-specialize. Now, the `doString` specialization will activate. Since now the `doInt` specialization is no longer the only active one, we will change the quickened opcode back to generic, handling both cases.

We can use *corpus-guided optimization* to determine which instructions to quicken and which specializations to include. The most straightforward heuristic takes the corpus's most commonly encountered specialization sets. In the future, more advanced heuristics should be applied, taking re-quickening, boxing splits, and instruction complexity into account.

3.8 Serialization

Another valuable feature of a language is the ability to serialize its constructs to a file and later deserialize them from it. This serialized form can then be used to send them over the network or cache them. Some languages, like Python, use this extensively for caching parsed source files. We also wanted to support this use case and give an option to serialize and deserialize Operation nodes.

The simplest solution would be to have the bytecode itself be the serialized form of the node, plus the serialized forms of all the constants used in it. However, this would expose the bytecode format outside of Operation DSL control and thus would need some form of versioning. Additional problems arise if someone modifies the serialized bytecode – some of the assumptions we make about the bytecode may no longer hold, and validating them at load-time may be slow.

Instead, we opted for the serialized form of the Operation node being the sequence of `begin`/`end` calls that created it, along with the serialized forms of the arguments used during construction. During serialization, a special flag is set in the builder, and the source is reparsed as explained in section 3.4. Instead of producing bytecode, the builder calls are logged to an output stream. Locals and labels are serialized according to their creation order, and constants are serialized using a language-provided serialization callback.

A special parser implementation is used when we want to deserialize the stream. It reads the logged builder calls and replays them in the same order as a regular builder instance. Constants are deserialized using a language-provided deserialization callback, which is expected to behave in reverse to the serialization callback provided to the serializer. Because the same code builder is used during regular parsing, this works with all the previously explained features of the DSL, such as reparsing, and does not expand the API surface that needs to be validated.

Chapter 4

Implementation

This chapter gives some information about the current implementation details of the Operation DSL. These details may evolve as more improvements are made to the generated code, and more supported features are added. We will detail the bytecode format, how it is emitted, and how some Truffle features are supported.

4.1 Bytecode design

In this section, we will describe the overall design of the bytecode format and some decisions that were made during development. We will discuss how branches and constants are handled in the bytecode, how we keep the specialization and inline caches from the DSL in the bytecode itself, and how we handle child pointers and boxing elimination.

The Operation DSL bytecode is a simple stack-based virtual instruction set. Each instruction starts with a two-byte opcode, followed by a variable number of bytes representing the arguments. All instructions pop data from the stack and push the results back onto it. A few instructions are fixed by the Operation DSL itself (instructions performing branches, constant loading, locals handling, ...). In contrast, others are generated from the operation descriptions given by the guest language – each custom operation becomes one or more instructions in the bytecode itself. The overview of instructions produced by the Operation DSL can be seen in Table 4.1.

In the current version of the bytecode, there are two branch instructions – a conditional jump (based on a stack value) and an unconditional one[1]. Both of them have an unsigned absolute destination address as an argument. The conditional branch also has an index into a *condition profile pool*, to allow branch profiling.

[1] Additionally, short-circuiting instructions also have branch offsets, which are explained later.

Instruction Name	Description
`branch`	Performs an unconditional branch to the target (specified as an absolute offset). Only instruction that can perform backwards jumps.
`branch.false`	Pereforms a conditional branch if the top of stack is `false`, otherwise continues from the next instruction.
`pop`	Removes the top of stack value. Used for discarding unused results.
`load.const.«t»`	Pushes a constant value onto the stack. There is a generic `object` variant, and a boxing-eliminated version for primitives.
`load.argument`	Pushes an indexed argument onto the stack. Because arguments are always passed boxed, there are no boxing-eliminated versions.
`store.local.«t»`	Stores the top of stack value into an indexed local. There exinsts an unitnitialized version `uninit`, versions for all primitive types, and a generic boxed version.
`load.local.«t»`	Loads a value from the local, and pushes it onto the stack. Analogous to the `store.local` instruction, there is an uninitialized version, primitive versions, and a generic boxed version.
`throw`	Helper instruction used with `FinallyTry` operations, used for rethrowing the caught exception, which is read from an indexed local.
`return`	Returns the top of stack value from the function.
`instrument.enter`	Notifies the instrumentation that the instrumented tag has been entered.
`instrument.exit`	Notifies the instrumentation that the instrumented tag has been exited. It inspects the top of stack value as the return value of the tag, but does not pop it.
`instrument.leave`	Notifies the instrumentation that the instrumented tag has exited, but not via normal completion (emitted before a branch or return instruction).
`c.«name»`	Custom instructions, defined for each of the custom operations defined by the language.
`c.«name».q.«s»`	Quickened variants of custom instructions. `«s»` reflects which specializations are quickened over.
`sc.«name»`	Short-circuiting instructions, which based on top of stack value, either discard it, or branch to their branch target. Produced from custom short-circuiting operations.

Table 4.1: Overview of bytecode instructions produceed by the Operation DSL. Some of them represent families of related instructions that change into each other based on boxing elimination state – in these `«t»` is replaced by the type produced/expected by the instruction. Some of them are defined by custom operations defined in the DSL – therse incorporate the operation name as `«name»`.

Additionally, the unconditional branch instruction is the only one that performs a backwards jump (the others are only ever emitted as forward jumps). This restriction means that the logic that only happens on backwards jumps (e.g., safepoint and *on-stack replacement* checks) can only be implemented on that instruction.

All the constants used in the function are stored in a *constant pool* – for this, a simple `Object[]` is used. The constant pool is also marked as *compilation constant*, meaning it will fold during compilation. Some instructions read this array and push constants onto the stack, which are emitted from the `ConstObject` operation outlined in the previous section.

The most important aspect of Operation DSL is the ability to provide the same advantages as the regular Node DSL but inline all the data into the bytecode array. This data includes which specializations are active, represented as a bitmask of active and excluded specializations. In the Node DSL, these were represented using integer fields of the generated nodes. Since we do not have a per-instruction node object in the bytecode, these are inlined into the bytecode array – several bytes after the opcode are reserved for the specialization data. These can easily be accessed relative to the current instruction pointer. During specialization, these are modified to reflect the new specialization state, just like in the Node DSL.

Another possibility of the DSL is to use *caching* – storing additional state for each specialization (initialized only upon activation). In the Node DSL, these were again stored in additional fields in the generated node. In the bytecode, we reserve additional space for them in the *constant pool*. Then, any instruction using any caches gets a single index into the first entry in the constant pool (and the other entries, if needed, are indexed relative to that). This usage entails that the constant pool is not entirely constant (as the caches get initialized), but since this initialization always happens on the slow path (behind a *code invalidation*), it does not impact the correctness of code compilation.

An important consideration when dealing with caches is thread safety. If one thread has set the specialization bit but has not yet initialized the inline caches, and the other reads the specialization bit, we may use non-initialized caches. Because of this, we disallowed usage of primitive type caches and `null` values. If we encounter a `null` value, we know that a race condition is in progress and will re-specialize. The re-specialization will then happen under a lock, where no such race conditions can occur. This way, neither specialization bits nor caches need to be read with volatile semantics, which improves multithreaded performance.

As mentioned previously, some operations can be *variadic*, meaning they can take a variable number of arguments. The instructions implementing these operations are also variadic, in that the stack effect of such instruction is not known only from the opcode itself. Instead, they take an additional argument, representing the arity of the operation. This argument is then used to pop the appropriate number of values off the stack and store them in the `@Variadic` argument array, which is passed to the operation.

For the short-circuiting operations, the generated instructions are slightly different. In

```
      «a»
      sc.And       end
      «b»
      sc.And       end
      «c»
  end:
```

Listing 4.1: Example of compiling a short-circuiting operation `a && b && c` to bytecode. The `end` is a label, that the `sc.And` instruction jumps to if the top of stack has a `false` boolean value.

addition to the regular specialization and caching data, they take an additional branch offset. Then, based on the top of stack value, they either

- pop the top value and fall through to the next instruction (if the value satisfies the `continueWhen` test), or
- leave the value on the stack and branch to the target (the end of the entire operation).

All the children, except the last, are followed by one such instruction, meaning that the code such as `a && b && c` would be compiled as shown in Listing 4.1. The design of short-circuiting operations is based on the design of the `JUMP_IF_FALSE_OR_POP` and related instructions in the CPython runtime.

One problem encountered while translating the Node DSL concepts to bytecode was the lack of a parent-child relationship between the operations. This relationship was important when implementing boxing elimination (section 3.5). For this reason, explicit "child" pointers are added to each instruction for each value they pop off the stack. They point to the instruction that initially pushed that value (if this were a tree-walk interpreter, it would be its child node). These are not always unique, so this is performed on a best-effort basis – if the operation is not unique (such as after conditional branches), the child pointer is left empty, and boxing elimination is not performed.

Bytecode format details

As mentioned previously, the generated bytecode layout is an implementation detail and can be changed since it is both emitted and used by the generated code only. However, we will still provide an overview of the bytecode layout as it is implemented currently.

The bytecode is, contrary to the name, represented as a sequence of 16-bit `shorts`[2]. Each

[2] Initial versions used a `byte` array for storage, but as most internal structures are 16-bit, we decided to change to a `short` array for the performance benefit, at a small cost of packing efficiency. This change allowed faster, aligned reads and better partial evaluation.

instruction in the bytecode starts with a 16-bit opcode, followed by several immediate arguments. Currently, each instruction contains a fixed number of arguments which can be:

- Local variable indices, for reading and writing to them, using a 16-bit index.
- Branch offsets, using a 16-bit absolute index into the instruction array (limiting the code size to 128 KB).
- Base index into the constants and child array. Instructions that require multiple constants or child nodes encode only the index of the first one, and others are indexed relative to that.
- "Child" instruction indices, for boxing elimination purposes. For space-saving purposes, these are encoded using unsigned 8-bit offsets relative to the current instruction, and two of them are packed into a single word. If the target instruction is more than `0xFF` words away, a 0 is written, and boxing elimination is not performed.
- State bitset, separated into 16-bit words in case the bitset is longer.
- Index into the condition profile array for conditional branch instructions.

4.2 Bytecode generation

The Operation Builder is the only exposed API that can be used to generate the bytecode. The program is specified as a series of `begin`/`end` calls to this builder, representing the operations.

In order to speed up the builder, the bytecode is generated as the calls are made, with as little additional object allocation as possible. The operations are structured so that no lookahead is needed to emit the code, so no backtracking or code moving is required. All code is either emitted during the `begin` call, during the `end` call or before/after a `begin`/`end` call of one of the children. For example, the `IfThenElse` can be constructed like so:

- After the `end` call of a first child, emit a `branch.false` instruction.
- After the `end` call of the second child, emit a `branch` instruction, and make the previous `branch.false` point right after it.
- After the `end` call of the third child, make the previous `branch` point to the current bytecode index.

There is additional validation of the number of children and whether a child operation is expected to produce a value or not. If a child is expected to produce a value (e.g., in a condition of a `IfThen` operation) and the operation specified was not a value-producing one, then an

exception is thrown since otherwise, the stack layout could get corrupted. If a child was not expected to produce a value but did, a `pop` instruction is emitted to clear the stack. Some operations take the value-producing properties of their children (e.g., `Block` is value producing if its last child is value producing).

4.3 Exception handling

An important aspect of language implementation is exception handling. For this, the `TryCatch` operation is used, which executes the first argument inside an *exception handler* context. If any exception is thrown during execution, control transfers to the second argument after storing the caught exception in a variable for processing.

During bytecode generation, `TryCatch` marks the start and end bytecode indices of its first child and the start bytecode index of the handler. Additionally, it marks the stack depth that has to be reset (since the exception may have occurred at a deeper stack position) and the local in which the exception has to be stored. If no excepetion happens, the control will jump over the handler using a `branch` instruction. The generated code can be seen in Listing 4.2.

```
1       «protected child»
2       branch        end
3
4   ; on exception, branch here
5       «handler»
6   end:
```

Listing 4.2: The bytecode generated by a `TryCatch` operation

To implement this in the interpreter, we use a single Java `try ... catch` block, catching all Truffle exceptions. If an exception is caught, a linear scan through the current function's *exception handlers* list is used to find the appropriate exception handler, based on the current bytecode index and the handler's start and end indices. The exception is then stored in the local designated by the handler, the stack is cleared to the depth required, and control transfers to the handler.

4.4 Handling `finally` blocks

The `FinallyTry` operation allows simple implementation of constructs similar to `try ... finally` in Java (e.g., Python `with`). These require the exit handler to be called regardless of how the body execution ends (normally, with an explicit control flow exit or through an exception). Two approaches were considered when implementing this:

- having a subroutine call and return instructions, similar to `jsr` and `ret` in JVM. Such an implementation would allow the exit handler code to be emitted once, and all exit points would call it as a subroutine, or
- emitting the exit handler multiple times, once at each exit point and in the exception handler.

In the end, it was decided to go with the second option, as the first one would either result in the inferior performance of compiled code or would end up duplicating the handler at every subroutine call at the compiler level (as is currently done with `jsr` instructions). The second approach can sometimes result in exponential code duplication. However, this was considered acceptable.

The order of arguments to a `FinallyTry` operation was chosen to simplify the code generation and remove the need to move the bytecodes once emitted. First, the "finally" block is translated completely. Then the "try" block is translated, inserting the bytecode of the "finally" at every exit point (any `Return`, `Branch`, or similar operation), and all their internal labels are repositioned (since we are using absolute branch offsets). This way, the code generation remains linear, without the need to move the code around during generation (only the handler code is copied).

In order to facilitate exit handling, a stack of open operations has to be maintained. Each non-local control flow operation (a branch or return) checks if this stack contains any `FinallyTry` operations, and if so, emits their exit handler code before emitting its own. A similar mechanism was later reused to implement instrumentation (explained in section 4.6), which must also be notified of exits.

Additionally, an exception handler is created, as if a `TryCatch` was used, that catches all exceptions, performs the "finally" handler, and rethrows them. Since some languages have different logic for exceptional and non-exceptional exits, a separate `FinallyTryNoExcept` operation is available, which performs the previously described exit-handling logic but without creating the exception handler.

4.5 Source locations

Truffle supports reporting current source location within functions for use in exception tracebacks, program introspection, and instrumentation. In Node-based languages, each tree node is assigned a single source location and those that do not have one inherit from their parent node. Such an implementation works reasonably well if the nodes are granular enough. In the case of bytecode interpreters, entire functions are represented using a single node, so workarounds are required to give more precise source locations.

To achieve this, we use the current bytecode index and the possibility of obtaining the source

location based on it. Additional support from the existing exception system is required to store and use this value when reporting stack trace source locations.

Source locations are only stored in the node if the current builder configuration requires them (e.g., during a reparse, section 3.4). Special operations `Source` and `SourceSection` are used to specify the source, and position within that source. These can be nested to represent nested source constructs – the current location will always be the innermost enclosing one. All source sections are stored along with their starting bytecode index, and during a source location request, a scan of this array is performed to resolve a bytecode index to a source location.

4.6 Instrumentation

Just like source locations, instrumentation is a feature of Truffle that allows building more complex tools such as debuggers on top of polyglot languages. To do this, it exposes a simple API that allows attaching listeners to tree nodes to monitor their entry and exit. For this reason, nodes can be *tagged* using instrumentation tags, representing their role in the tree (e.g., statement, expression, function body, ...). Just like source locations, this does not map straight to a bytecode-based interpreter, as there are no nodes to which instrumentation tags can bee attached. Instead, Truffle provides an API that allows a language to *materialize* the instrumented nodes. In tree-based languages, this is used to expand possibly simplified nodes (such as supernodes, or constant folded expressions), but here we can use it to create a completely independent tree of *instrument nodes* (called the "instrument tree"). The Truffle framework can then instrument this tree.

Just like with nodes, we use tags to specify the instrumentation behaviour. For this, a `Tag` operation is used, which takes the tag class as an argument, and "wraps" the contained operations in it. These tags will be ignored during regular parsing, but during an instrumentation reparse, they will be expanded.

During tag expansion, additional instructions are inserted into the bytecode that call into the instrument tree. These mark entry into and exit out of the particular tag. For exits, the process similar to `FinallyTry` is used to ensure that all possible control flow exits are covered. During exception processing, the instruments are also invoked to process the exception. Because these instructions never modify the stack, they can be skipped over once the instrumentation is detached.

Currently, only a proof-of-concept of the instrumentation system is implemented, which does not correctly work with the other features mentioned. For this reason, only an overview is given here. Furthermore, more work is needed on the Truffle API side to support bytecode-based instrumentation as first-class citizens.

Chapter 5

Case studies

In this chapter, we will demonstrate Operation DSL's usage to implement certain aspects of a dynamic language. We will cover some aspects of language implementation using the DSL by analyzing simplified parts of the Python `for` loop semantics implementation.

5.1 Defining operations

Some examples of operation definitions were given in previous chapters (Listing 3.3, Listing 3.7). In order to support an iterator loop in Python, we will need an operation that, given an iterator, returns both whether it has any more elements and, if it has any, the element in question. To return the next element, we will use a `LocalSetter` argument, and the actual return value of the operation will be a boolean `true` if it has any elements, and `false` otherwise.

This operation can be specified as shown in Listing 5.1. The `doIntegerIterator` specialization handles integer-returning iterators. The specialization requires a specialized integer iterator type `PIntegerIterator` (line 6). In addition it receives the implicit frame argument (line 5), and a local reference `LocalSetter output` (line 7). If the iterator is empty it immediately returns `false` (lines 8-10), otherwise it reads the next element, stores it in the frame using the `LocalSetter`, and returns `true` (lines 11-12).

The `doIterator` specialization handles all other objects, with the help of the `GetNextNode`. The helper node is executed on the object, which returns the next iterator element (line 25), which is stored into the local (line 26), and `true` result returned (line 27). If executing thee `GetNextNode` throws an exception signaling the end of iteration, we store the `null` into the local, and return `false` (lines 28-33).

```
@Operation
static final class ForIterate {
  @Specialization
  static boolean doIntegerIterator(
      VirtualFrame frame,
      PIntegerIterator iterator,
      LocalSetter output) {
    if (!iterator.hasNext()) {
      return false;
    }
    output.setInt(frame, iterator.next());
    return true;
  }

  // long, double and Object iterators ...

  @Specialization
  static boolean doIterator(
      VirtualFrame frame,
      Object object,
      LocalSetter output,
      @Cached GetNextNode next) {
    try {
      // get the next element using a helper node
      Object value = next.execute(frame, object);
      output.setObject(frame, value);
      return true;
    } catch (StopIterationException e) {
      // catch the StopIteration exception,
      // used to signal exhausted iterators
      output.setObject(frame, null);
      return false;
    }
  }
}
```

Listing 5.1: Definition of `ForIterate` operation, elided for brevity.

```
1  static OperationNodes parseSource(Source source) {
2    // perform the actual parsing
3    ParseTree tree = parse(source);
4    return ExampleOperationsBuilder.create(
5      OperationConfig.DEFAULT,
6      builder -> {
7        Visitor visitor = new Visitor(source, builder);
8        tree.accept(visitor); // start the recursive visit
9      }
10   );
11 }
```

Listing 5.2: The Operation DSL builder entry point, using a parse tree visitor pattern.

5.2 Parsing and desugaring

Like in Node DSL, parsing the source code is left to the language. The Operation DSL starts with an already parsed and possibly semantically analyzed program. The main entry point is the `Builder#create` method, which takes a configuration parameter, and the *builder callback*.

We use a visitor pattern over the language parse tree in the example in Listing 5.2. We will start from the parsed source (line 3), and invoke the builder `create` method. We will use the `DEFAULT` parsing configuration which strips most of the metadata from the nodes. The builder callback (lines 6-9) creates the visitor, and starts the tree visit.

In Listing 5.3 we show how functions are emitted. First, we visit each statement in the function in order, emitting the operations for it (lines 5-7). Then, we end the function body with an implicit `return null` (lines 10-12). Finally, we end the function using the `publish()` function (line 15).

In Listing 5.4 we show how a Python-like `for` loop can be implemented using Operation DSL. Lines 1-2 show a general structure of the `for` loop, with two placeholders: *«iterable»* and *«body»*. Lines 4-8 how a `for` loop would be desugared to a `while` in Java. The `GetIterator` function (line 4) creates an iterator from an iterable object. The `ForIterate` (line 6) is the equivalent of the operation we defined earlier, which traverses the iterator. The `&value` is the by-reference variable passing, something Java does not have natively. The third block (lines 10-14) shows the same code but decomposed into individual operations.

In Listing 5.5 the code that visits the `for` loop parse tree, and emits the corresponding operations is shown. Lines 3-4 define the temporary locals `iter` and `value`. Lines 6-10 emit the operations `(StoreLocal iter (GetIterator «iterable»))`. The local variable to store into is passed as the argument to the `begin` method of the corresponding operation. Lines 12-24 emit the operations for the `(While ...)`. The lines 14-16 emit the condition `(ForIterate &value`

```
private ExampleOperationsBuilder builder;

void visitFunction(Function node) {
  // execute all statements
  for (Statement statement : node.statements) {
    statement.accept(this);
  }

  // functions end with an implicit return
  builder.beginReturn();
  builder.emitConstObject(Null.INSTANCE);
  builder.endReturn();

  // create the node
  OperationNode node = builder.publish();
}
```

Listing 5.3: The Visitor implementation, with the `visitFunction` function. This is the entry point for parsing each function.

```
for value in «iterable»:
  «body...»

Object iter = GetIterator(«iterable»);
Object value;
while (ForIterate(iter, &value)) {
  «body...»
}

(StoreLocal iter (GetIterator «iterable»))
(While
  (ForIterate (LoadLocal iter) &value)
  (Block
    «body...»))
```

Listing 5.4: The desugaring of an iterator `for` loop to a `while`, using a `ForIterate` operation defined earlier. The first block is the original `for` loop in Python-like syntax, the second is the desugaring in a Java-like syntax, and the third is the operations in S-expression form.

```
1  void visitFor(For node) {
2    // create the temporaries iter and value
3    OperationLocal iter = builder.createLocal();
4    OperationLocal value = builder.createLocal();
5
6    builder.beginStoreLocal(iter);
7      builder.beginGetIterator();
8        node.iterable.accept(this);
9      builder.endGetIterator();
10   builder.endStoreLocal();
11
12   builder.beginWhile();
13
14     builder.beginForIterate(value);
15       builder.emitLoadLocal(iter);
16     builder.endForIterate();
17
18     builder.beginBlock();
19       for (Statement statement : node.body) {
20         statement.accept(this);
21       }
22     builder.endBlock();
23
24   builder.endWhile();
25 }
```

Listing 5.5: The implementation of the `for` visitor function, that emits the desugared operations. The code is indented according to the begin/end structure for readability.

`(LoadLocal iter))`. The lines 18-22 emit the body of the while loop. The body is a single `Block` operation, containing all the statements of the `for` loop.

Chapter 6

Evaluation

The Operation DSL was used to implement an interpreter for Python inside the GraalPython runtime. This implementation was compared to the Node-based and manually written bytecode interpreters.

Currently, the Operation DSL Python interpreter is not fully compliant with the other two interpreters, mostly in introspection, generator, and exception handling. However, it can successfully run many benchmarks from the Python test suite, including bootstrapping the benchmark harness itself. We will therefore consider the three interpreters similar in complexity since the bulk of the language functions as expected.

6.1 Specification complexity

The most important reason for using Operation DSL is to simplify the process of creating bytecode interpreters. To evaluate the success of this goal, we will compare the specification complexity of the three Python interpreters. As a metric, we will consider the total number of operations needed to specify the language behaviour and the number of significant lines of code as a rough complexity measurement. For operations, in the case of the AST interpreter, we will count the nodes used directly.

Figure 6.1 shows the complexity measurements of different interpreters. For the *significant lines of code* (SLoC) metric, only non-empty, non-comment lines of code were considered. For the *specification SLoC* only nodes directly used by the interpreters are considered. We excluded the indirectly used helper nodes, as all three interpreters equally use these. For the *operation count* metric, we counted the number of different nodes in the case of the AST interpreter, the number of instructions in the case of the bytecode interpreter, and the number of custom operations in the case of the Operation DSL interpreter. The *parser backend SLoC* metric compares the

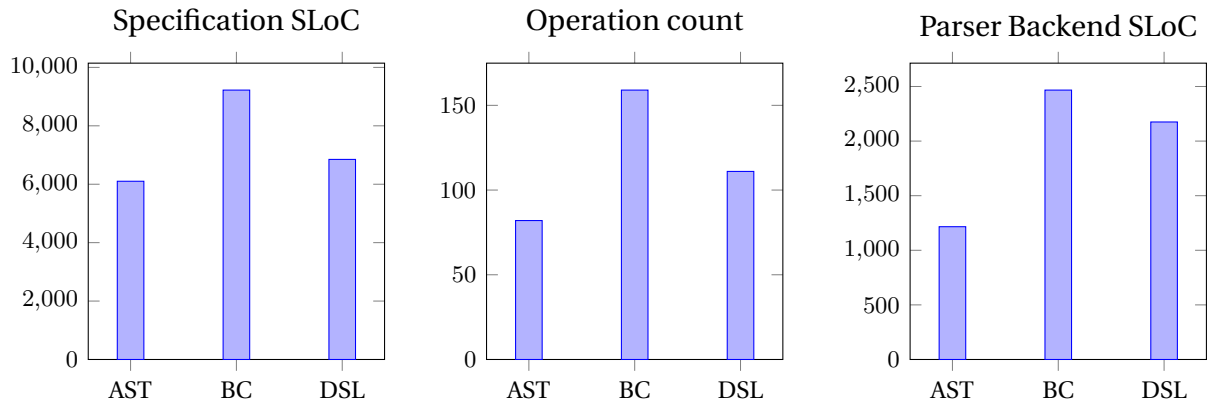

Figure 6.1: Comparison of tree-based (AST), manually written bytecode (BC), and Operation DSL generated (DSL) Python interpreters, based on code complexity metrics. Lower is better.

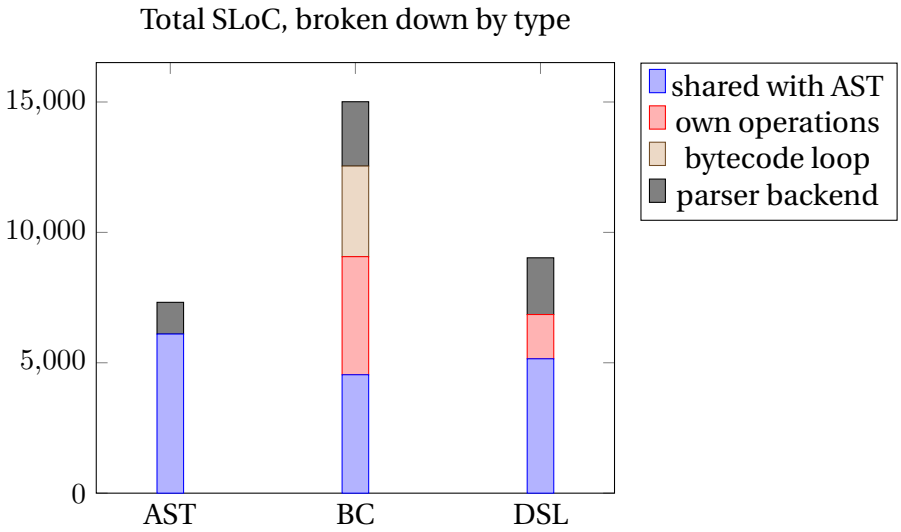

Figure 6.2: Total SLoC needed to implement the three interpreters, broken down by component.

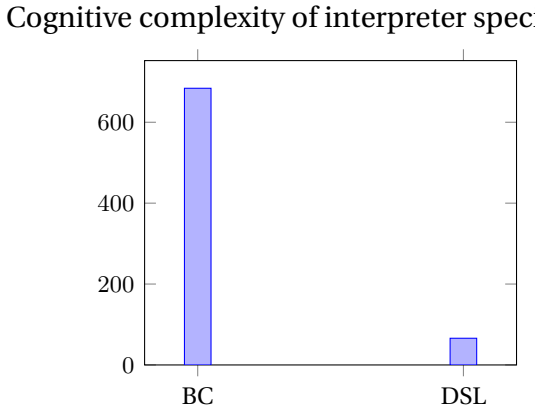

Figure 6.3: Total values of the *cognitive complexity* metric for the specifications of the two different interpreters (excluding the nodes shared with the AST).

complexity of generating the interpreter from a parsed syntax tree – all three take the same syntax tree structure as input and generate the executable nodes.

Figure 6.3 further divides the SLoC metric by type of the code implemented: nodes shared with the AST, operations defined by that interpreter only, and the parser backend. Additionally, the bytecode interpreter contains the bytecode loop code as a separate item.

Furthermore, using the *cognitive complexity* [3] metric, the complexity of the bytecode and DSL specification was analyzed, excluding the nodes shared with the AST. The manually written bytecode interpreter contains 10.36 times more complex code, even though the code analyzed is only 4.74 times longer.

The specification complexity is in general higher in case of Operation DSL than the AST interpreter. The additional complexity can be attributed to the requirement to separate high-level operations defined in the AST interpreter into multiple, lower-level operations.

The operation count is also higher for the DSL than the AST since some simple functions, such as reading the variadic arguments, must be implemented as a separate operation. The bytecode interpreter has a higher operation count than DSL since it has to manually define instructions for local and stack manipulation as well as quickened variants, while those are generated automatically by the Operation DSL.

It should be noted that the DSL also generates some additional features not counted in the bytecode interpreter implementation, such as serialization and instrumentation support. Further, bytecode interpreter still uses nodes internally to handle specialization logic, removing the memory footprint benefits that bytecode interpreters can offer.

6.2 Existing DSL specification reuse

The second goal of the Operation DSL was to reuse as much as possible of the existing AST-style interpreter specification and convert it into bytecode operations using `@OperationProxy`.

To evaluate this goal, we can first look at SimpleLanguage, the example language implementation in Truffle. Out of 16 operations, 15 were reusable, by only changing the specialization methods from `protected` to `public static`, and adding a `@Bind("this")` argument where needed. The only non-reusable operation was the function call operation `SLInvoke` which takes variadic arguments and cannot be directly reused. It was instead rewritten, leading to slight code duplication.

This goal is less successful in Python implementation. Out of 111 operations, 36 could be reused. The remaining 75 can be split into the following categories:

- 29 are simple operations that would have no direct analogies in the AST world or would be helper functions/snippets of code. These are value constructors (e.g. `MakeList`), cell loads/stores (`LoadCell`), and argument handlers (`GetVariableArguments`).
- 9 are more complex operations that also have no equivalent in the AST world, or their behaviour is a part of a larger non-proxiable node. One such example is the `ForIterate` operation explained in chapter 5.
- 18 are AST nodes that could not be adequately proxied since they use non-static state. Instead, a wrapper operation that calls a cached node instance was created.
- 19 are reimplemented versions of Python comparison and in-place arithmetic nodes. While these do not use static state, modifying them to comply with the Operation DSL requirements breaks the AST interpreter, so they were reimplemented instead.

A general pattern was observed during implementation – if an operation used all the features of the DSL (e.g., helper node caching), instead of implementing them manually (by, e.g., null checking and deoptimizing), the node would be reusable with minor modifications. Otherwise, the heavy reliance on the node state would make the node hard to reuse directly, requiring a wrapper operation or a complete re-implementation.

Chapter 7

Related Work

7.1 Bytecode interpreter generation

Due to the repetitive nature of their code, bytecode interpreters are often automatically generated by tools from instruction descriptions. One such generator is *VMGen* [6]. It takes as input a description of each instruction (in the form of the stack effect, immediate arguments, and the C code that performs it) and generates the runtime engine, compiler, optimizer, and profiler.

The compiler is similar to the operation builder of the Operations DSL in that it is a wrapper used for generating the bytecode (there is, however, no concept of nesting, so only a single `gen_` function is provided per instruction, instead of `begin`/`end`). The stack and locals are exposed to the language implementors, and the generated code does not validate stack consistency. However, this allows some behaviour prohibited by Operation DSL, such as pushing multiple values from a single instruction.

VMGen generates highly optimized interpreter code: the bytecode is *directly threaded*. The instructions themselves point to the code implementing them, meaning that a dispatch is simply an indirect branch. Additionally, the code for fetching the next instruction is optimized to allow modern processor prefetching. The first optimization is impossible in Java, as there are no function pointers (alternatives using functional interfaces/lambdas are considered). The second optimization is handled by the Graal host compiler, which can reorder independent pieces of code.

Similar mechanisms inspired by VMGen are employed by other bytecode-based interpreters such as Ruby YARV [10].

7.2 Inline caching and quickening

Inline caches are commonly used to speed up bytecode interpreter performance. Instead of performing an expensive lookup every time an instruction is executed (e.g. a property descriptor or a call target), we can perform it once and cache it *inline*, that is in the instruction argument itself. Caching requires an additional check that the precondition still holds (e.g. the operand types are the same) but assuming it still does, it dramatically speeds up the computation. This method has been used to speed up Smalltalk performance by caching call targets [4] and later implemented in other languages. Later, this was also combined with quickening in *INCA* [1] to further speed up the dispatch for primitive instructions by replacing an indirect cache call with direct implementation.

The DSL handles the inline caches through `@Cached` arguments. These are used to cache call targets, property descriptors, and similar, and *guards* are used to validate the preconditions. Since we cannot store object pointers directly with the bytecode, we keep them in a separate array and store the offset in the bytecode.

Implementation of quickening (section 3.7) which creates instructions that work only on a subset of specializations, together with caching explained previously, gives similar results to *INCA* – we end up with special instructions that operate on the most common values. Additionally, boxing elimination of the stack (section 3.5) gives us the similar effects to machine-level instructions in [2].

Chapter 8

Future work

In this section, we will briefly discuss the planned improvements to the Operation DSL and further work that can be done on it.

Super instruction detection A simple super-instruction detection prototype has been implemented but never thoroughly evaluated. A more advanced super-instruction detector should be implemented, which would also take care of branches and specialization. The design of the DSL, with its simple instructions, heavily relies on the existence of a super-instruction finder to optimize away common instruction sequences. This detector could also work on the level of operations, by generating tree-rewriting rules and merging instruction which would otherwise be separated by unknown code.

Top-of-stack elimination This simple optimization stores the topmost stack element in a host-level local variable instead of in the guest-level frame. This optimization makes the reads and writes to the topmost stack slot faster but requires a spill if an instruction does not access it.

Dead code elimination This optimization can remove instructions that will never be evaluated by keeping track of reachable code paths. Because the operation structure provides more information than individual instructions, this analysis can be performed on the level of operations and during build-time, removing the need for a post-processing pass.

Constant folding Like the previous one, this is a common optimization performed by languages that evaluates constant parts of expressions at parse-time instead of evaluating them (possibly multiple times) at runtime. The DSL can perform this automatically during build time, marking certain operations (or individual specializations) as constant-foldable. Then, if all arguments

are constants and the specialization matches, the individual instructions are replaced with a single constant load.

Node flattening Creating small, helper nodes is commonplace in Truffle. Other nodes can then reuse these as if they were functions while keeping the benefits of specializations by declaring them as `@Cached` arguments. However, using them in Operation DSL means that the advantages of bytecode interpreters are partly lost since the Node objects must still be constructed for them. Instead, the state of the primary node and all the helper nodes (including their helper nodes, transitively) could be flattened into a single bitset stored in the bytecode array.

Statically typed languages Currently, Operation DSL assumes a dynamically typed language. A statically typed language can be implemented on top of it by performing the static type analysis and then discarding the result. Such an approach is inefficient since the static type information can be used to inform a lot of otherwise runtime-inferred decisions. By having mechanisms to provide this information to the DSL during build-time, it could generate better runtime code – e.g. by preemptively enabling specializations based on static types.

Single assignment locals Locals only assigned once (such as synthetic temporary variables) can be optimized further. They could be stored directly on the stack instead of having to be copied into the locals. A read would be a stack-relative load; upon leaving the scope, they could be popped. Local boxing elimination could also be simplified in those cases since there is only one setter.

Support for bytecode-first languages Currently, Operation DSL is aimed at implementing source-based languages. However, a good use case would be for Truffle languages that have a canonical bytecode form (e.g., Smalltalk [9]) or are bytecodes themselves (JVM, LLVM). These are often stack-based, just like the generated bytecode is, but usually have fewer constraints over the structure of the bytecode. Also, it may not always be possible to convert them to a tree-like structure of Operations. Thus a new API must be made for them. Allowing the bytecode-based languages to use Operation DSL features would simplify their implementation and maintenance and possibly introduce features like boxing elimination and constant folding.

Compiler optimizations The Graal compiler is currently tuned to compile node-based interpreters. Optimizations that are beneficial to bytecode interpreters should be explored, in order to match the performance of state of the art interpreters.

Chapter 9

Conclusion

This thesis presents an implementation of Operation DSL, a generator for high-performance bytecode interpreters, which abstracts away the individual instructions, control flow, and stack and local manipulation, to allow for simpler, cleaner, and more performant interpreter implementations. It strives to make writing performant bytecode interpreters as simple as writing tree-walk ones while allowing performing even more optimizations over the code.

We present already performed optimizations, and some planned to be implemented. Specializations and quickening allow the interpreter to be highly performant and produce fast native code when compiled. Boxing elimination removes unnecessary allocations in dynamic code while still being performant if handling non-primitive objects becomes required. Corpus-guided optimizations allow automatic generation of optimization decisions (such as quickened instructions) based on a representative body of code.

The generated interpreter is simple to transition to, as demonstrated by implementing an experimental Python interpreter, and offers all the benefits of the Node DSL, with the addition of automatic bytecode generation and validation, branch profiling, and exception handling. It is already performant, with still room for improvement.

Bibliography

[1] Stefan Brunthaler. “Inline caching meets quickening”. In: *European Conference on Object-Oriented Programming*. Springer. 2010, pp. 429–451.

[2] Stefan Brunthaler. “Multi-level Quickening: The Key to Interpreter Performance”. 2012.

[3] G. Ann Campbell. *Cognitive Complexity - a new way of measuring understandability*. Tech. rep. Version 1.6. SonarSource SA, 2021.

[4] L Peter Deutsch and Allan M Schiffman. “Efficient implementation of the Smalltalk-80 system”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1984, pp. 297–302.

[5] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. “Graal IR: An extensible declarative intermediate representation”. In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 2013.

[6] M Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. “vmgen—a generator of efficient virtual machine interpreters”. In: *Software: Practice and Experience* 32.3 (2002), pp. 265–294.

[7] Yoshihiko Futamura. “Partial computation of programs”. In: *RIMS Symposia on Software Science and Engineering*. Springer. 1983, pp. 1–35.

[8] Christian Humer. “Truffle DSL: A DSL for Building Self-Optimizing AST Interpreters”. MA thesis. Johannes Kepler Universität Linz, 2016.

[9] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. “Graalsqueak: A fast smalltalk bytecode interpreter written in an ast interpreter framework”. In: *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. 2018, pp. 30–35.

[10] Koichi Sasada. “YARV: yet another RubyVM: innovating the ruby interpreter”. In: *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, pp. 158–159.

[11] Christian Wimmer and Thomas Würthinger. “Truffle: a self-optimizing runtime system”. In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 2012, pp. 13–14.

[12] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. “Practical partial evaluation for high-performance dynamic language runtimes”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 662–676.

[13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. “One VM to rule them all”. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, pp. 187–204.

[14] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. “Self-optimizing AST interpreters”. In: *Proceedings of the 8th Symposium on Dynamic Languages*. 2012, pp. 73–82.