Efficient Implementation of Parametric Polymorphism using Reified Types

by

Matt D'Souza

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

© Matthew D'Souza 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Parametric polymorphism is a language feature that lets programmers define code that behaves independently of the types of values it operates on. Using parametric polymorphism enables code reuse and improves the maintainability of software projects.

The approach that a language implementation uses to support parametric polymorphism can have important performance implications. One such approach, erasure, converts generic code to non-generic code that uses a uniform representation for generic data. Erasure is notorious for introducing primitive boxing and other indirections that harm the performance of generic code. More generally, erasure destroys type information that could be used by the language implementation to optimize generic code.

This thesis presents TastyTruffle, a new interpreter for the Scala language. Whereas the standard Scala implementation executes erased Java Virtual Machine (JVM) bytecode, TastyTruffle interprets TASTy, a different representation that has precise type information. This thesis explores how the type information present in TASTy empowers TastyTruffle to implement generic code more effectively. In particular, TASTy's type information allows TastyTruffle to reify types as objects that can be passed around the interpreter. These reified types are used to support heterogeneous box-free representations of generic values. Reified types also enable TastyTruffle to create specialized, monomorphic copies of generic code that can be easily and reliably optimized by its just-in-time (JIT) compiler.

Empirically, TastyTruffle is competitive with the standard JVM implementation. Both implementations perform similarly on monomorphic workloads, but when generic code is used with multiple types, TastyTruffle consistently outperforms the JVM. TASTy's type information enables TastyTruffle to find additional optimization opportunities that could not be uncovered with erased JVM bytecode alone.

Acknowledgements

First, I want to thank my mom for her unconditional love and encouragement. She has supported me earnestly since the very beginning — even if I never could give her a good explanation of what a "compiler" was.

I especially want to thank my advisor, Ondřej Lhoták, without whom this thesis would not have been possible. Ondřej's second-year compilers course was the spark that ignited my interest in compilers and programming languages. He introduced me to Graal, encouraged me to apply to their internship program, and ultimately took me on as a student for this project. It has been a pleasure learning from him, and I hope we have opportunities in the future to continue working together.

In the course of my studies, I had the opportunity to learn from a lot of wonderful people on the Graal team. Thank you to Alex Prokopec for advising this research project. I'm lucky to have retained maybe half of the ideas he so patiently explained during our meetings. I am also thankful for Gilles Duboscq, Christian Humer, Roland Schatz, and the other team members for their support during my internship. Thank you to Chris Seaton for your generosity and kindness.

Many thanks to Gregor Richards and Peter Buhr for reading this thesis in its entirety and for the valued feedback. I did not intend to write so much, but I hope that it was at least interesting to read. I am also grateful to Patrick Lam for his mentorship throughout my 7+ years at Waterloo.

Though I cannot name them all, I want to thank the people who made grad school bearable. Thank you to James You for being an excellent research partner, in times of success and times of stress. Thank you to all of my labmates in PLG for the Tim Hortons coffees and Grad House beers. Thank you to my friends for putting up with me disappearing into schoolwork for months on end. Thank you to my roommates for the shared dinners, board games, and volleyball. Thanks to Will Shortz for the crosswords.

Finally, I want to thank my wonderful partner and best friend, Grace. I cannot imagine myself finishing this thesis without your love, support, and patience. Thank you for encouraging me when I had setbacks, for taking care of me when I couldn't, and for pretending to believe me when I said I'd be done writing by ~~December~~ ~~January~~ ~~February~~ March.

This work was supported by the National Sciences and Engineering Research Council of Canada.

Table of Contents

List of Figures viii

List of Tables xi

1 Introduction 1
- 1.1 Implementing parametric polymorphism . . . 1
- 1.2 Parametric polymorphism in Scala . . . 2
- 1.3 The problem with erasure . . . 2
- 1.4 Scala's TASTy representation . . . 4
- 1.5 Thesis overview . . . 4
- 1.6 Contributions . . . 5

2 Background 7
- 2.1 Parametric polymorphism . . . 7
  - 2.1.1 Monomorphization . . . 8
  - 2.1.2 Type erasure . . . 10
  - 2.1.3 Parametric polymorphism in Scala . . . 11
- 2.2 Scala's TASTy representation . . . 13
- 2.3 The Graal compiler . . . 14
  - 2.3.1 Graal IR . . . 16
  - 2.3.2 Optimizations . . . 22
- 2.4 The Truffle ecosystem . . . 27
  - 2.4.1 Truffle interpreters . . . 27
  - 2.4.2 Partial evaluation . . . 30
  - 2.4.3 Self-optimizing ASTs . . . 36
  - 2.4.4 Truffle's object model . . . 39

3 TastyTruffle: A Truffle interpreter for Scala 42
- 3.1 The TastyTruffle AST . . . 42
- 3.2 Definitions . . . 43
  - 3.2.1 Referencing definitions in TastyTruffle . . . 46
- 3.3 Data representation in TastyTruffle . . . 47
  - 3.3.1 Objects . . . 47
  - 3.3.2 `Shapes` . . . 49
  - 3.3.3 Data representation and the AST . . . 51

3.4 Method dispatch . . . 54
3.4.1 Direct calls . . . 54
3.4.2 Indirect calls . . . 56

4 Using reified types in TastyTruffle 60
4.1 Reified types . . . 60
4.1.1 `TypeNodes` . . . 61
4.2 Generic methods . . . 62
4.2.1 Generic locals . . . 63
4.2.2 Generic array accesses . . . 66
4.2.3 Propagating type information . . . 66
4.3 Generic classes . . . 66
4.3.1 Modeling applied generic classes . . . 67
4.3.2 Applying generic classes to type arguments . . . 69
4.3.3 Modeling generic classes in the AST . . . 73
4.3.4 Generic fields . . . 77
4.3.5 Method dispatch . . . 77
4.3.6 Generic parent classes . . . 79

5 Specializing generic code 81
5.1 Motivation . . . 81
5.2 Specializing generic methods . . . 84
5.3 Specializing generic class methods . . . 85
5.4 Static specialization . . . 87
5.4.1 Statically specializing generic methods . . . 88
5.4.2 Statically specializing generic class methods . . . 88

6 Evaluating TastyTruffle 91
6.1 Benchmarks . . . 91
6.2 Setup . . . 91
6.3 Throughput . . . 93
6.3.1 Polymorphic workloads . . . 93
6.3.2 Monomorphic workloads . . . 101
6.3.3 Comparing dynamic and static specialization . . . 105
6.4 Transient behaviour . . . 106
6.4.1 Warmup . . . 106
6.4.2 Compilation . . . 108

7 Related Work 110
7.1 Implementing parametric polymorphism . . . 110
7.2 Working around type erasure on the JVM . . . 111
7.2.1 Static specialization . . . 111
7.2.2 Reifying Scala types . . . 112
7.3 Just-In-Time (JIT) compilation of Scala programs . . . 112
7.4 Truffle interpreters . . . 113

8 Conclusion 114

References 115

APPENDICES 120

A Generic `swap` AST 121

List of Figures

2.1 Definitions for a generic List structure. 8
2.2 A generic C++ List class before and after monomorphization. 9
2.3 A generic Java List class before and after erasure. 10
2.4 Source code for ScalaRunTime.array_apply. 11
2.5 Code using a ClassTag context bound. 12
2.6 Code using an Ordering type class. 12
2.7 Scala source and TASTy for a simple Box class. 14
2.8 High-level view of Graal and the Java ecosystem. 15
2.9 Source code and IR for a simple addFive method. 17
2.10 Source code and IR for a Counter and its count method. 17
2.11 Source code and IR for a max method that tracks the largest value seen. . . 18
2.12 Source code and IR for an iterative factorial method. 20
2.13 Source code and IR for a recursive factorial method. 21
2.14 Source code and IR for a method with dynamically unreachable code. . . . 22
2.15 Definition of a Point class and a hierarchy of Transforms. 22
2.16 Source code and IR for a method transformed by type-checked inlining. . . 24
2.17 Source code and IR for a method transformed by partial escape analysis. . 25
2.18 Source code and IR for a method transformed by type-checked inlining and partial escape analysis. 26
2.19 High-level view of the Truffle ecosystem. 28
2.20 An example AST and the source code for its nodes. 29
2.21 Pseudocode for f(x) = x * 2 (from Figure 2.20) after partial evaluation. . 31
2.22 IR for f(x) = x * 2 after partial evaluation and escape analysis. 32
2.23 Source code for a node with a deoptimization directive. 33
2.24 Source code for a Cache node. 34
2.25 Graal IR for a Cache of x * 2. 35
2.26 Graal IR from Figure 2.25 after dead code elimination. 35
2.27 State machine for a self-optimizing Plus node. 36
2.28 Source code for a self-optimizing Plus node. 37
2.29 Source code for a node that caches deterministic computations. 38
2.30 Modeling an Item with the static object model. 40
2.31 The relation between the generated Item class and its static properties. . . 40
2.32 Modeling a SaleItem with the static object model. 41
2.33 The relation between the generated SaleItem class and its static properties. 41

3.1 Source code and TastyTruffle AST for an addOne method. 43

3.2 Source code and TastyTruffle AST for a `factorial` method. 44
3.3 Source code for `Counter` and TastyTruffle AST for its `increment` method. 45
3.4 Source code for the `Representation` enum. 48
3.5 Data definition for the `Field` class. 48
3.6 A `Rectangle` class and its run-time representation in TastyTruffle. . . 48
3.7 A `TranslucentRectangle` class and its run-time representation in TastyTruffle. 49
3.8 Data definition for the `Shape` class. 49
3.9 Method declarations and `Shapes` for the rectangle classes. 50
3.10 Source code for a `ReadLocal` node before and after partial evaluation. . . . 52
3.11 Source code for a `WriteLocal` node. 52
3.12 Source code for an `ArrayInit` node. 54
3.13 Source code for an `ArrayApply` node. 55
3.14 Example code with a direct call. 55
3.15 Source code for a `DirectCall` node. 56
3.16 Example code with an indirect call. 57
3.17 Source code for an `IndirectCall` node. 57
3.18 Pseudocode generated by PE for the indirect call to `bar` in Figure 3.16. . . 58

4.1 A generic `swap` method that updates an array and returns the previous value stored. 61
4.2 Scala pseudocode to construct a TastyTruffle `TypeNode` from a TASTy `TypeRepr`. 62
4.3 Source code for a `MethodTypeParam` node. 63
4.4 Source code and AST for a generic method `foo`. 64
4.5 Source code for a `GenericReadLocal` node. 65
4.6 Pseudocode for a `GenericWriteLocal` node's `execute` method after partial evaluation. 66
4.7 Source code for a `GenericArrayApply` node. 67
4.8 Generic `Box` class. 67
4.9 Data definition for the `Shape` class (reproduced). 68
4.10 Data definition for the `GenericShape` class. 68
4.11 Source code for a `ClassTypeParam` node. 69
4.12 Data definition for the `GenericShapeTemplate` class. 69
4.13 Data definition for the `FieldTemplate` class. 70
4.14 Source code and example usage for the `TypeNodeSpecializer`. 71
4.15 Source code for the `GenericShapeTemplate`'s `apply` method. 72
4.16 Source code for the `GenericShapeTemplate`'s caching mechanism. 74
4.17 Pseudocode for `lookup` (from Figure 4.16) after partial evaluation. 75
4.18 Source code for an `AppliedType` node. 76
4.19 Source code for a `GenericReadField` node. 78
4.20 Example code with generic parent classes. 79

5.1 A generic `swap` method that updates an array and returns the previous value stored (reproduced from Figure 4.1). 81

5.2 Part of the control flow graph for swap. 82
5.3 Part of the control flow graph for swap after tail duplication. 83
5.4 Generic AST for swap after specialization. 85
5.5 Source code for the TypeSwitch node. 86
5.6 Generic Box class (reproduced from Figure 4.8). 87
5.7 A List map method with both class and method type parameters. 87
5.8 Source code for the GenericMethodSpecializer class (Scala). 89
5.9 AST for swap specialized over Int. 89
5.10 Source code for the GenericClassSpecializer class (Scala). 90

6.1 Source code for ArrayCopy. 94
6.2 Graal IR subgraph for ArrayCopy on G. 95
6.3 Graal IR subgraph for ArrayCopy on G where the source array is int[]. 97
6.4 Simplified source code for InsertionSort. 98
6.5 Graal IR subgraph for InsertionSort on T_U where ord is the Ordering[Int] instance. 99
6.6 Source code for StdDev. 101
6.7 Graal IR subgraph for ArrayCopy on G when the generic array accessors are used with other types. 102
6.8 Source code for the put method of HashMap. 103
6.9 Graal IR subgraph for the put method of HashMap[Int,Int] on G. . . . 104
6.10 Average throughput over time for Checksum, HashMap, and QuickSort. 107

A.1 Generic AST for swap (from Figure 5.1). 122

List of Tables

6.1 Table of benchmarks. 92
6.2 Throughput of each benchmark. 94
6.3 Throughput of each benchmark using static and dynamic specialization). . 105
6.4 Truffle compilation counts and timing for CHECKSUM, HASHMAP, and QUICKSORT. 109
6.5 Truffle compilation size data for CHECKSUM, HASHMAP, and QUICKSORT. 109

Chapter 1

Introduction

Parametric polymorphism is a powerful abstraction that allows programmers to describe algorithms and data structures independently of the types of values they operate on. For example, an algorithm that copies values from one array into another does not depend on the type of the values stored in the array: it is *generic* over the array's component type. Parametric polymorphism encourages code reuse and improves the maintainability of software projects.

1.1 Implementing parametric polymorphism

Though parametric polymorphism is an elegant feature in the abstract, it introduces some design challenges for the language implementation (i.e., the compiler or interpreter). There exist a variety of different approaches to support generics in language implementations with inherent strengths and limitations. Most of these approaches fall broadly into one of two categories:

Monomorphization (heterogeneous translation) uses different data representations for each unique set of type parameters (an *instantiation*). For each instantiation, a specialized copy of the code that operates on those representations is created. This approach creates arbitrarily many specializations of generic code, which can increase the code size, but the resulting code has precise type information and representations that can be efficiently compiled.

Erasure (homogeneous translation) uses a single uniform data representation. Each generic method is translated to a single method that treats generic data uniformly. Erasure produces less code than monomorphization, but forces generic data to conform to a single representation, which can introduce inefficiencies like primitive boxing. Erasure also loses useful type information that could be used by a just-in-time (JIT) compiler to produce more efficient code at run time.

Despite a language designer's best efforts, parametric polymorphism is often a leaky abstraction. Programmers who care about the run-time characteristics of their programs (e.g., memory usage or throughput) must take the language's implementation of parametric polymorphism into account when writing generic code. For example, in Java programming,

it is conventional wisdom to avoid generic collections over primitive types if performance is important.

1.2 Parametric polymorphism in Scala

The Scala programming language is a statically-typed programming language with a rich notion of types. It is designed to interoperate with Java, and its standard implementation compiles Scala source to Java Virtual Machine (JVM) bytecode. JVM bytecode does not support parametric polymorphism, so generic Scala code is erased to a uniform representation that does not have type parameters.

Consider the simple Scala method below:

```
1 def overwrite[T](array: Array[T], i: Int, value: T): T = {
2   val old = array(i)
3   array(i) = value
4   old
5 }
```

This method overwrites the `i`-th element of `array` with `value` and returns the previous value stored in the array. Since `overwrite` should behave the same way regardless of the types of its arguments (as long as their types agree, of course) it can be written generically over a type parameter `T`.

After erasure, the signature becomes:

```
1 def overwrite(array: Object, i: Int, value: Object): Object = { ... }
```

The method is translated to a representation that effectively "forgets" that it was generic. The representation instead treats any value of generic type `T` as an `Object` (the universal reference supertype). In Scala's implementation of generics, the generic `Array[T]` type also gets erased to the `Object` type.

1.3 The problem with erasure

Erasure can introduce significant overheads at run time. Since generic values use a uniform representation in an erasure scheme, often a language's primitives (e.g., a plain integer value) must be boxed up when used in generic contexts. Boxing and unboxing operations can significantly increase heap allocations and introduce undesired memory indirection.

Boxing is an issue in the JVM, especially with arrays. In Java, the generic array type `T[]` is erased to `Object[]`, which forces all primitive values to be boxed when they are stored in generic arrays. Scala tries to avoid boxing in generic arrays by defining the primitive array types like `Array[Int]` as subtypes of the generic array type `Array[T]`. Unfortunately, while this approach avoids boxing, it gives generic arrays a heterogeneous representation, which means that the Scala code must dynamically switch over the array's type in order to implement array accesses.

Below is the same `overwrite` method in Scala pseudocode after erasure:[1]

```
1  def overwrite(array: Object, i: Int, value: Object): Object = {
2    val old: Object = array match {
3      case intArray: Array[Int] =>
4          java.lang.Integer.valueOf(intArray(i))
5      case dblArray: Array[Double] =>
6          java.lang.Double.valueOf(dblArray(i))
7      ... // other primitive arrays
8      case objArray: Array[Object] =>
9          objArray(i)
10   }
11   array match {
12     case intArray: Array[Int] =>
13       intArray(i) = if (value == null) 0
14                     else value.asInstanceOf[java.lang.Integer]
15                            .intValue()
16     case dblArray: Array[Double] =>
17       dblArray(i) = if (value == null) 0.0
18                     else value.asInstanceOf[java.lang.Double]
19                            .doubleValue()
20     ... // other primitive arrays
21     case objArray:    Array[Object] => objArray(i) = value
22   }
23   old
24 }
```

Observe in the erased pseudocode how `array`'s type must be checked before it can be read from (lines 2-7) or written to (lines 8-17). This indirection introduces additional overhead to the program. Also observe how accessing array elements still requires primitive values to be boxed (e.g., line 3) since the only type that encapsulates all possible results of an array read is `Object`. While not always the case, optimizing compilers can sometimes elide this boxing if it is provably redundant.

A core idea put forth in this thesis is that, when generic types are erased from a language's representation, the language's implementation is ill-equipped to optimize generic code. When types are erased, the resulting code is often highly dynamic and difficult to optimize. Just-in-time (JIT) compilation can sometimes improve performance, for instance, by eliding box-unbox sequences and redundant type checks, but even the best JITs are driven by imperfect heuristics that sometimes fail.

Erasure is fundamentally a *lossy* conversion that reduces the type information available to the language implementation. For example, in Scala, `Array[T]` is erased to `Object`. Though the `Object` is always some sort of array, this invariant is erased from the JVM bytecode, which makes it difficult for the JVM to efficiently implement array accesses.

Sometimes, dynamic type profiling can help recover this lost type information. However, when these type profiles are polymorphic, they are not as useful. Type profiles are coarse-grained measurements that can only capture first-order information about the types seen at run time. Higher-order information, such as the relationships among generic values,

[1]Erasure occurs during the translation to JVM bytecode; Scala pseudocode is depicted for the sake of presentation. `Int` and `Double` are primitive types, whereas `java.lang.Integer` and `java.lang.Double` are boxed types.

cannot be easily reconstructed by profiling. Consider the following Scala method with two `Array[T]` parameters:

```
1  def copy[T](src: Array[T], dst: Array[T]): Unit = {
2    // copy src array into dst
3  }
```

At the Scala source level, it is obvious that both parameters have the same element type for any given invocation. At the JVM bytecode level, things are not so clear. Suppose the method is called with integer and double arrays. Type profiles record that the parameters took on types `Array[Int]` and `Array[Double]`, and the JIT can use this information during code generation. However, the fact that `src` is `Array[Int]` *if and only if* `dst` is `Array[Int]` is completely opaque. The JVM must often perform additional analysis to figure out these sorts of relationships, or otherwise generate inefficient code.

1.4 Scala's TASTy representation

The Scala 3 compiler introduced a new intermediate representation for Scala programs called TASTy. TASTy is a compact binary representation used by the Scala compiler to represent Scala programs. In TASTy, each definition and expression is annotated with precise type information computed by the compiler. The TASTy representation is lossless: unlike JVM bytecode, wherein generic types from Scala source code are erased, TASTy preserves all generic type information. The availability of precise types in TASTy makes it a valuable alternative to JVM bytecode as an interpretation target.

1.5 Thesis overview

The goal of this thesis is to show that making generic type information accessible empowers a language implementation to use more optimal data representations, reduce polymorphism, and ultimately generate more efficient code.

To start, this thesis introduces TastyTruffle, an interpreter for TASTy written in the Truffle framework (Chapter 3). Truffle is a framework that reduces the effort required to create high-performance language implementations. A language interpreter written in Truffle is converted to a compiler for the language at run time using partial evaluation and a JIT compiler.

Since TastyTruffle interprets TASTy instead of JVM bytecode, it has full information about generic types. This type information gives TastyTruffle more opportunities to efficiently implement generic code.

Firstly, TastyTruffle can use the type information in TASTy to *reify* types at run time (Chapter 4). Using reified types, TastyTruffle allows generic programs to dynamically switch between different data representations that avoid boxing. TastyTruffle's implementation of `overwrite` directly uses the type parameter `T` to perform type-sensitive operations at run time (pseudocode depicted):

```
1  def overwrite(T: ReifiedType, array: Array[T], i: Int, value: T): T={
```

```
2   val old = T match {
3     case Int => array.asInstanceOf[Array[Int]](i)
4     case Double => array.asInstanceOf[Array[Double]](i))
5     ...
6     case _ => array(i)
7   }
8   T match {
9     case Int => array.asInstanceOf[Array[Int]](i) = value
10    case Double => array.asInstanceOf[Array[Double]](i) = value
11    ...
12    case _: array(i) = value
13  }
14  old
15 }
```

Since TastyTruffle has precise type information, when primitives flow into generic contexts, they can be stored in their natural representations without boxing. For example, when `T` is `Int`, `value` is stored as an `Int` and can be copied directly into the `Array[Int]` without unboxing (line 9).

When TastyTruffle knows which methods and classes are generic, it can also dynamically *specialize* generic code (Chapter 5). Specialization transforms polymorphic code to monomorphic code that can be more easily optimized by TastyTruffle's JIT compiler, which allows it to achieve performance more reliably. Since specialization happens dynamically, TastyTruffle mitigates the static code size penalty often associated with monomorphization.

Evaluated on a variety of small generic programs, TastyTruffle achieves comparable and often better peak performance than a regular JVM implementation (Chapter 6). These gains come with a modest increase in compilation pressure and warm-up time. With inspection of the compiler graphs, it is apparent that using TASTy as an interpretation target allows TastyTruffle to find optimization opportunities that could not be inferred with JVM bytecode.

1.6 Contributions

TastyTruffle was developed in collaboration with James (Jack) You. Both of us contributed substantially to various aspects of its implementation, and as a result, there is no clean delineation between our work. There are, however, some differences between the contents of this thesis and Jack's thesis [50].

Jack's thesis introduces the TastyTruffle interpreter. It provides significant detail about the TastyTruffle AST and how the TASTy representation can be converted to a TastyTruffle AST. Chapter 3 of this thesis also introduces TastyTruffle, but it does not have the same depth of detail.

Jack's thesis extends the basic TastyTruffle interpreter to support generic code. This extension relies both on reified types and static specialization. Chapter 4 of this thesis extends TastyTruffle to reify types *without* specialization. This unspecialized extension serves as a comparison point and helps validate the necessity of specialization. Support for unspecialized execution also enables TastyTruffle to specialize its ASTs

dynamically rather than rely on static transformations. Chapter 5 presents a dynamic specialization scheme in addition to the static scheme presented in Jack's thesis.

Finally, Jack's thesis evaluates TastyTruffle on a series of `List` and `ArrayBuffer` microbenchmarks. Chapter 6 of this thesis evaluates TastyTruffle on a wider range of larger benchmark programs. A larger portion of the evaluation is spent investigating patterns in the compiled code.

Chapter 2

Background

This chapter provides background information to understand the motivation and implementation of TastyTruffle. It first discusses how Scala and other languages implement parametric polymorphism (Section 2.1). Then, it introduces TASTy IR, the intermediate format interpreted by TastyTruffle (Section 2.2). Finally, it explains how the Graal compiler and Truffle framework enable high-performance language implementations (Sections 2.3 and 2.4). These technologies are complex, so this chapter aspires to convey a thorough but concise understanding of how they work.

2.1 Parametric polymorphism

Static type systems allow programmers to precisely describe the types of values that flow through their code. However, sometimes code behaves independently of some of its values, and their concrete types are unimportant to the description of the code. In such scenarios, the code is *generic*, and it is desirable for the code to work the same way regardless of the types of values it encounters.

Programming languages support generic code using *parametric polymorphism*. With parametric polymorphism, programmers can annotate values and functions with type parameters, which serve as placeholder types. Then, when some code *instantiates* a generic definition—that is, it uses the definition with concrete type arguments—the compiler can type-check it by replacing the type parameters with the concrete types.

Consider a `List` data structure consisting of two fields: a `data` element, and a possibly-empty `tail` list. The code that describes how to manipulate `List`s—to create a singleton `List`, prepend an element, read `data`, etc.—does not change whether `data` is an integer, string, pointer, or something else. Thus, `List` can be parameterized over a type parameter `T`, (written as `List<T>`). The type annotations of the fields and functions that operate on `List<T>`s use `T` to indicate that they behave generically. Figure 2.1 provides sample definitions for a generic `List` in an imaginary language.

At a call site like `makeList<Int>(42)`, the type-checker understands that `T` is `Int` and thus that the result has the concrete type `List<Int>`. The types of the functions that operate on this `List<Int>` can also be inferred. For example, the type-checker can infer that calling `getData` on a `List<Int>` will yield an `Int`. Thus, the generic definition allows

```
1 structure List<T> {
2   data: T
3   tail: List<T>
4 }
5 function makeList<T>(data: T): List<T> { ... }
6 function prepend<T>(data: T, list: List<T>): List<T> { ... }
7 function getData<T>(list: List<T>): T { ... }
8 function getTail<T>(list: List<T>): List<T> { ... }
```

Figure 2.1: Definitions for a generic `List` structure.

code reuse while preserving the safety guarantees of static typing.

Parametric polymorphism is a valuable language feature, enabling code reuse and a richer notion of types. However, deciding to support parametric polymorphism in a language raises a difficult question: how should it be *implemented*? Historically there have been two main strategies to implement parametric polymorphism, monomorphization and type erasure. These strategies can be seen as two ends of a spectrum that trade off code size and performance.

2.1.1 Monomorphization

Monomorphization is the process of specializing generic code to concrete types. Each time a generic definition is instantiated with a new set of concrete types, the compiler creates a new version of the definition. Each version is called a *specialization*. A specialization's type parameters are replaced with concrete types, and so the new definition is *monomorphic* and can be implemented as usual.

Several languages, including C++ and Rust, use monomorphization to implement parametric polymorphism. The previous `List` example, implemented using C++ templates, is presented in Figure 2.2. The compiler determines that `List` is instantiated with types `int` and `char`, so it instantiates two specializations with `T` replaced by `int` and `char` respectively. At each use-site of the `List` definition (e.g., `List<int>`), the compiler resolves it to the corresponding specialization (e.g., `List_int`).

Monomorphization gives a compiler full information about the types of values. It can accurately determine the layout of objects and local variables, which is important in low-level languages like C++ and Rust. This extra knowledge enables more optimizations within the compiler.

Knowing the concrete value of a type parameter also means the compiler can resolve operations performed on generic values. For example, in C++ it is possible to write `x + y` for two values declared with generic type `T`, because the compiler can statically check for the appropriate `operator+` definition each time a template is instantiated.

The tradeoff of monomorphization is code size. Each unique set of types used to instantiate a generic definition creates a new specialization. These specializations have distinct, incompatible representations, so code that operates on generic definitions must be duplicated.

```
1  template<typename T>
2  class List {
3    T data;
4    List<T>* tail;
5
6    List(T data, List<T>* tail) { ... }
7    List<T>* prepend(T data) { return new List<T>(data, this); }
8    T getData() { return data; }
9    List<T>* getTail() { return tail; }
10 };
11
12 List<int>* intList = (new List<int>(42, nullptr))
13     ->prepend(123);
14 List<char>* charList = (new List<char>('i', nullptr))
15     ->prepend('h');
```

(a) Before monomorphization.

```
1  class List_int {
2    int data;
3    List_int* tail;
4
5    List_int(int data, List_int* tail) { ... }
6    List_int* prepend(int data) {
7        return new List_int(data, this); }
8    int getData() { return data; }
9    List_int* getTail() { return tail; }
10 };
11 class List_char {
12   ...
13 };
14
15 List_int* intList = (new List_int(42, nullptr))->prepend(123);
16 List_char* charList = (new List_char('i', nullptr))
17     ->prepend('h');
```

(b) After monomorphization.

Figure 2.2: A generic C++ `List` class before and after monomorphization.

```
1  class List<T> {
2    T data;
3    List<T> tail;
4
5    List(T data, List<T> tail) { ... }
6    List<T> prepend(T data) { return new List<T>(data, this); }
7    T getData() { return data; }
8    List<T> getTail() { return tail; }
9  }
10
11 List<Foo> fooList = new List<Foo>(new Foo(), null);
12 Foo myFoo = fooList.getData();
13 List<Integer> intList = new List<Integer>(42, null);
14 int myInt = intList.getData();
```

(a) Before erasure.

```
1  class List {
2    Object data;
3    List tail;
4
5    List(Object data, List tail) { ... }
6    List prepend(Object data) { return new List(data, this); }
7    Object getData() { return data; }
8    List getTail() { return tail; }
9  }
10
11 List fooList = new List(new Foo(), null);
12 Foo myFoo = (Foo) fooList.getData();
13 List intList = new List(Integer.valueOf(42), null);
14 int myInt = ((Integer) intList.getData()).intValue();
```

(b) After erasure.

Figure 2.3: A generic Java `List` class before and after erasure.

2.1.2 Type erasure

Type erasure is another approach to implement parametric polymorphism. Instead of creating different specializations for each generic instantiation, the compiler produces a single implementation with a uniform representation. In this representation, each type parameter is replaced with some universal supertype; the genericity gets *erased* from the generated code. All client code uses the same erased representation.

Erasure is the approach taken by languages like Java and TypeScript. A generic Java `List` is depicted in Figure 2.3. During compilation, the type parameters inside the generic definition are replaced with `Object`, the supertype for all heap-allocated values, and the definition itself becomes non-generic (i.e., `List<T>` becomes `List`).

The compiler inserts type casts to ensure client code is still compatible with the erased representation. For example, suppose `List` is applied to some type `Foo`. The erased return type of `getData` is `Object`, so the compiler must cast the result to `Foo` to ensure the correct

```
1  def array_apply(xs: AnyRef, idx: Int): Any = {
2    xs match {
3      case x: Array[AnyRef]  => x(idx).asInstanceOf[Any]
4      case x: Array[Int]     => x(idx).asInstanceOf[Any]
5      case x: Array[Double]  => x(idx).asInstanceOf[Any]
6      case x: Array[Long]    => x(idx).asInstanceOf[Any]
7      ... // other primitive array cases
8      case null => throw new NullPointerException
9    }
10 }
```

Figure 2.4: Source code for `ScalaRunTime.array_apply`.

type is returned (line 12).

The situation for primitive values is more problematic. Primitive types like `int` or `double` are not subtypes of `Object`, so they cannot be used in generic contexts. Instead, Java inserts code to convert primitives to their *boxed* equivalents; for example, `42` is auto-boxed with the function `Integer.valueOf(42)` (line 13). When primitive values are obtained from generic contexts, they must also be unboxed by calling the boxed accessor (e.g., `intValue` on line 14).

Erasure was chosen for Java to maintain backward-compatibility. Using an erased representation allowed generic Java code to interface safely with older pre-generic Java code and the Java Virtual Machine (JVM), but today it serves as a pain point for performance. In aggregate, autoboxing introduces a lot of heap-allocation and indirection that can be avoided if primitives are type-compatible with erased representations.

2.1.3 Parametric polymorphism in Scala

Scala is designed to interface with Java code and run on the Java Virtual Machine (JVM). As a consequence, it uses the same erasure semantics to implement parametric polymorphism. Scala tries to work around the limitations of erasure in a few ways.

Arrays

Scala uses a different approach from Java to support generic arrays. Rather than require a single erased representation for `Array[T]`, Scala allows arrays of primitives (e.g., `Array[Int]`) to be passed into generic contexts. To support these different representations uniformly, the compiler replaces generic array accesses with calls to runtime accessor methods that handle all possible representations. For example, generic array reads are replaced with calls to `ScalaRunTime.array_apply` (Figure 2.4), which switches over the concrete type of the array to implement the access. This indirection has a cost: generic array accesses can be "three to four times slower" than concrete array accesses [2].

Once an array is created, the generic accessor approach is possible because primitive arrays have different types that can be differentiated at run time. However, creating a generic array (e.g., `new Array[T](size)`) is a different story, because Scala has no runtime notion of `T`; the code cannot know whether to allocate an integer array, an object

```
1  def main(args: Array[String]): Unit = {
2    val intArr = makeArray[Int]
3  }
4  def makeArray[T: ClassTag]: Array[T] = {
5    new Array[T](10)
6  }
```

Figure 2.5: Code using a `ClassTag` context bound.

```
1  trait Ordering[T] {
2    def compare(a: T, b: T): Boolean  // true if a is "less than" b
3  }
4  given Ordering[Int] with {
5    def compare(a: Int, b: Int): Boolean = a < b
6  }
7  ... // more given instances
8
9  def main(args: Array[String]): Unit = {
10   max[Int](41, 42)
11 }
12 def max[T: Ordering](a: T, b: T): T = {
13   val ord: Ordering[T] = summon[Ordering[T]]
14   if (ord.compare(a, b)) b else a
15 }
```

Figure 2.6: Code using an `Ordering` type class.

array, or something else. To support generic array creation requires a `ClassTag`, a kind of *context parameter*.

Context bounds

Type parameters in Scala can be annotated with *context bounds*, written as `T: Bound`, which indicate that the definition has an implicit *context parameter* of type `Bound[T]` at run time. At the use site of a generic definition, where the concrete type of the parameter is known, the compiler automatically creates context arguments and passes them around.

Take `ClassTag` as an example. A `ClassTag[T]` is a reified (run-time) representation of a type `T`. It can be used to instantiate an array of type `T`. `ClassTag` is often used as a context bound when the code needs to allocate arrays or perform other reflective operations.

In Figure 2.5, the `makeArray` function creates a generic `Array[T]`, so it requires a context bound `T : ClassTag`. At the use site `makeArray[Int]`, the compiler knows `T` is `Int` for the call. It creates a `ClassTag[Int]` object and passes it as an implicit argument to the call. Then, inside `makeArray`, the compiler implicitly invokes an array creation method on the `ClassTag` to instantiate an `Array[T]` (i.e., an `Array[Int]`).

`ClassTag`s are used throughout Scala's standard library to overcome the limitations of erasure. A recent survey of over 7,000 Scala projects [30] found that as much as 90% of

large applications that use the standard library rely on `ClassTags`[1].

Type classes [17] are another common use case for context bounds. A type class effectively extends a type `T` with additional capabilities. An example is the `Ordering[T]` type class in Figure 2.6, which imposes an ordering on `T` using a `compare` method. The method `max` defines an `Ordering` context bound. At call sites (e.g., `max[Int](41, 42)`) the compiler searches for an appropriate `Ordering` definition (e.g., `Ordering[Int]`) and supplies it to the call. Inside the call, `max summons` the context parameter and invokes its `compare` method to compare the two generic parameters.

2.2 Scala's TASTy representation

Scala's traditional execution platform is the Java Virtual Machine (JVM). The Scala compiler takes Scala source code and compiles it to JVM bytecode, which the JVM can execute. The conversion to JVM bytecode *loses information* about the original Scala program because it erases type information (Section 2.1.3).

Scala 3 introduced a new representation for Scala programs, TASTy[2]. TASTy is a binary encoding with "complete information" about the program [1]. It consists mostly of definitions, terms (expressions that produce a value), and types. The compiler generates TASTy after parsing and type-checking, but before type erasure, so all definitions and terms are tagged with precise types. These type trees are expressive: beyond simple named types (e.g., a class name), they can model type constructors, type parameters, type bounds, and more.

Consider the `Box` class in Figure 2.7a as an example. It is parametrized on type `T`, wrapping a value of type `T` with getter and setter methods. In Figure 2.7b is an abridged textual version of the TASTy for `Box`. The specific details about TASTy are beyond the scope of this thesis, but the reader should note the similarities between TASTy and Scala source.

The tree contains a type definition of the `Box` class (offset `5`), which contains a type parameter `T` (offset `11`) and regular parameter `initial` (offset `26`). TASTy trees are typed, so the `initial` parameter is tagged with the type `T` (via a type reference to offset `11`). The tree also contains a field definition for `value` (offset `74`), which is also of type `T` (note the `SHAREDtype` reference at offset `77`); it gets initialized to the value of `initial` (via a term reference to offset `26`).

The tree contains method definitions as well. For brevity, only `get` is presented (offset `86`). This method has result type `T` (offset `89`). Its body is a term that selects `value` (offset `93`) from `this` (offset `95`); this "select" corresponds to a read of the `value` field from the receiver.

[1]This percentage includes `TypeTags` and `Manifests`, which are similar to `ClassTags`.

[2]The name TASTy is an acronym for *typed abstract syntax trees*.

```scala
1 class Box[T](initial: T) {
2   private var value: T =
       initial
3   def get: T = value
4   def set(x: T): Unit = {
5     value = x
6   }
7 }
```

(a) Scala source

```
0: PACKAGE
...
5:   TYPEDEF 2 [Box]
9:     TEMPLATE
11:      TYPEPARAM 3 [T]
...
26:      PARAM 7 [initial]
29:        TYPEREFsymbol 11
...
74:      VALDEF 16 [value]
77:        SHAREDtype 29
79:        TERMREFsymbol 26
...
86:      DEFDEF 17 [get]
89:        IDENTtpt 3 [T]
91:          SHAREDtype 29
93:        SELECT 16 [value]
95:          QUALTHIS
96:            IDENTtpt 2 [Box]
...
```

(b) TASTy (abridged)

Figure 2.7: Scala source and TASTy for a simple `Box` class.

2.3 The Graal compiler

Graal is a relatively new just-in-time (JIT) compiler for the Java Virtual Machine (JVM) [48]. Java source code is compiled ahead-of-time to JVM bytecode, and the JVM executes this bytecode. When the JVM determines a method is "hot" (i.e., frequently executed), it invokes a JIT compiler (like Graal) to generate optimized code for the method. The JIT compiles the method from JVM bytecode to machine language; then, instead of interpreting the method, the JVM executes its optimized machine code. The whole compilation and execution process is depicted in Figure 2.8.

At the core of Graal's design is speculation [21]. The JVM provides profiling data (about types seen, branches taken, etc.), which Graal uses to make educated guesses about the future behaviour of a method that it needs to support. For example, if an if-statement condition has only evaluated to `true` during profiling, Graal may speculate that it will always be `true` and eliminate the `else` branch. Of course, these guesses can be wrong, in which case the compiled code must transfer control back to the interpreted version of the method so that the unexpected case can be handled. This process, called *deoptimization*, is complicated and expensive. Nevertheless, when these speculations pay off, they allow Graal to eke out performance gains that could not be obtained by a more conservative JIT.

Of specific interest in this thesis is the intermediate representation (IR) [21] that Graal uses to represent programs. Examining a compiled program's IR is one of the best ways to understand its performance on Graal, but IR graphs can be difficult to decipher. This section introduces the IR by example. The goal is not to be exhaustive, but to equip the reader with the background knowledge to interpret the IR graphs that appear in this

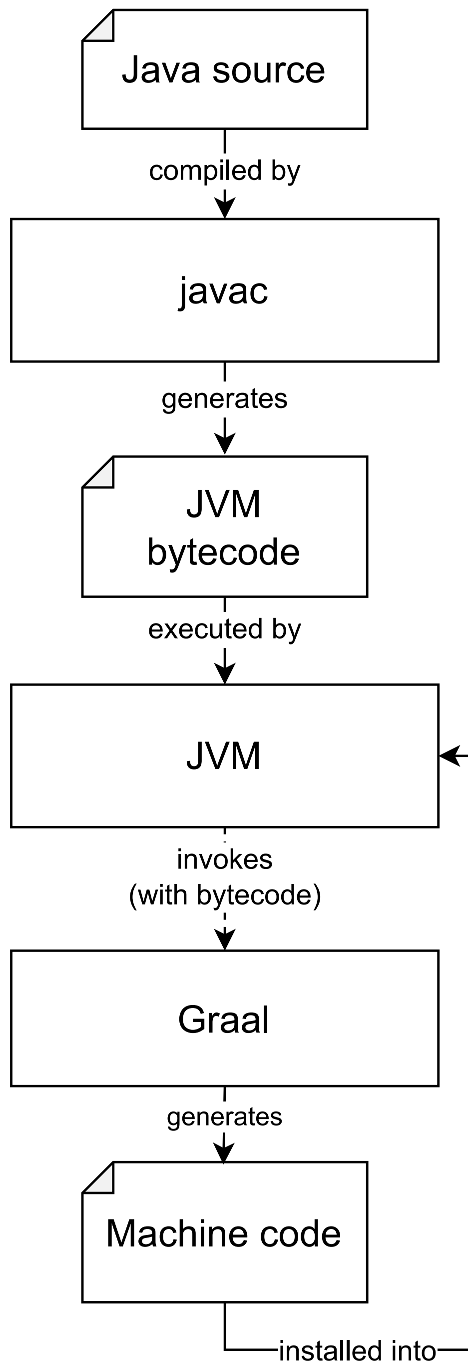


Figure 2.8: High-level view of Graal and the Java ecosystem.

thesis. This section also examines some of the important optimizations Graal performs to aggressively optimize the code it compiles.

2.3.1 Graal IR

When Graal compiles a method, it first parses its bytecode into Graal IR [21]. Then, each of Graal's optimization phases transforms this IR. After an extensive set of optimizations, the IR is lowered to platform-specific machine code that can be executed by the JVM.

It is valuable for readers to have a surface-level understanding of the IR: this thesis examines IR graphs to discuss the implications of various design decisions on the compiled code. Compiler graphs are rendered using the Seafoam visualizer[3]. Graphs are generated at specific stages of optimization in order to support the discussion—they are not necessarily taken from the same point in compilation. Also, for non-trivial programs, the IR graph can be large and unwieldy, so simplified graphs may be shown for the sake of presentation.

Graal IR follows the "sea-of-nodes" design [13, 14] popularized in the HotSpot Server compiler [36]. The IR is a directed graph where each node corresponds to a single operation. An example graph is depicted in Figure 2.9. Most edges in the graph model a control flow dependency (shown in red) or a data flow dependency (shown in teal). Nodes with control flow dependencies are *fixed* and define a "backbone" for the structure of the program. In contrast, nodes with data flow dependencies are *floating* and can be moved around as long as their dependencies are preserved. Thus, one can usually understand a graph by following the control flow edges and then looking at each node's data dependencies.

The IR is in *static single assignment* (SSA) form [16]. Each node can produce at most one value, and data flow dependencies ensure each use of a value is preceded by its definition. The IR contains phi (ϕ) nodes to represent values computed on different control flow paths.

Every value has a *stamp* that tracks its type as well as other properties like its nullity. For primitive values, these stamps retain more precise information. For example, the IR tracks the range and bit masks of an integer value.

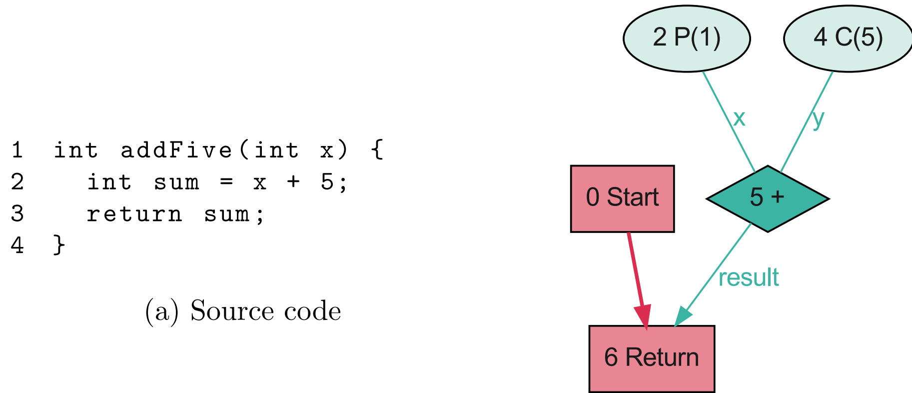
This section introduces the IR by example, starting with Figure 2.9. The `addFive` method adds `5` to its parameter and returns the result. The flow of control in this method is simple: control begins at `Start` and flows to `Return`. The data flow edge pointing to `Return` indicates that this method returns the result computed by the `+` node. This `+` node (which adds two integers) itself depends on two other values: `P(1)`, the parameter at index 1, i.e., `x`[4], and `C(5)`, the constant `5`.

Notice that the local variable `sum` is not present in the graph. Since Graal IR uses SSA, local variables disappear. Instead, a variable's value at any given time is modeled by a node in the graph; reads are then replaced by data flow edges from these nodes. In this example, Graal determines that the value of `sum` at the `return` statement is the value produced by `+`.

A second example is presented in Figure 2.10. The `Counter` class has a `count` method that returns how many times it has been called. It uses a `state` field to count previous

[3] https://github.com/Shopify/seafoam/

[4] `P(0)` is the method receiver.

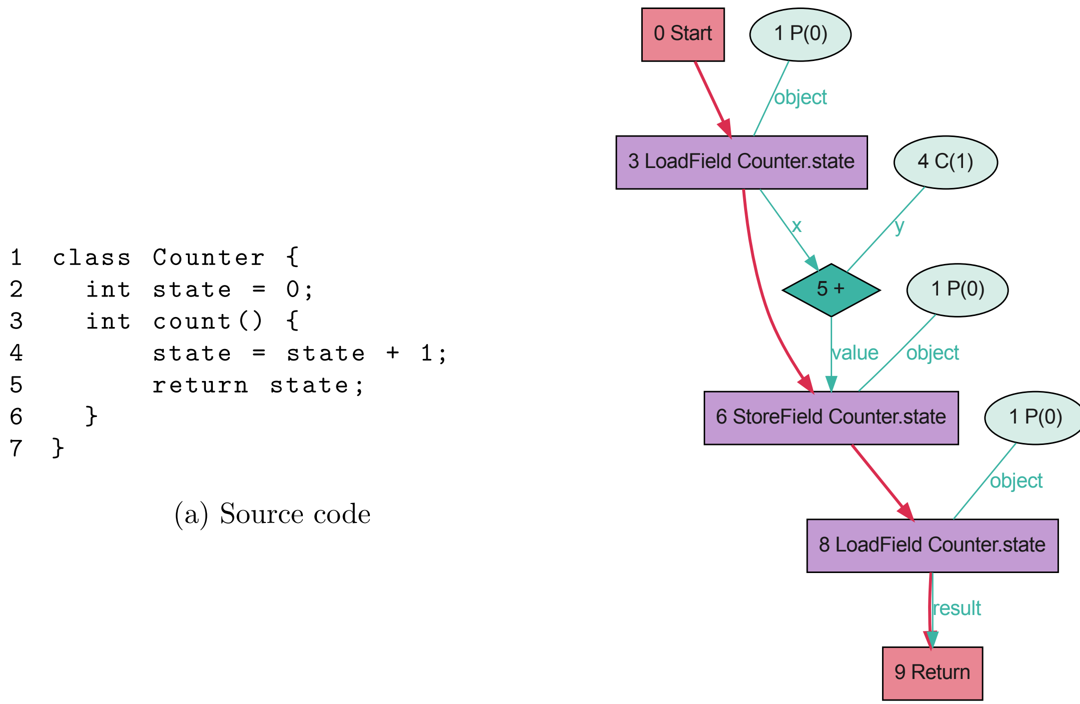



```
1  int addFive(int x) {
2    int sum = x + 5;
3    return sum;
4  }
```

(a) Source code

(b) Graal IR

Figure 2.9: Source code and IR for a simple `addFive` method.




```
1  class Counter {
2    int state = 0;
3    int count() {
4          state = state + 1;
5          return state;
6    }
7  }
```

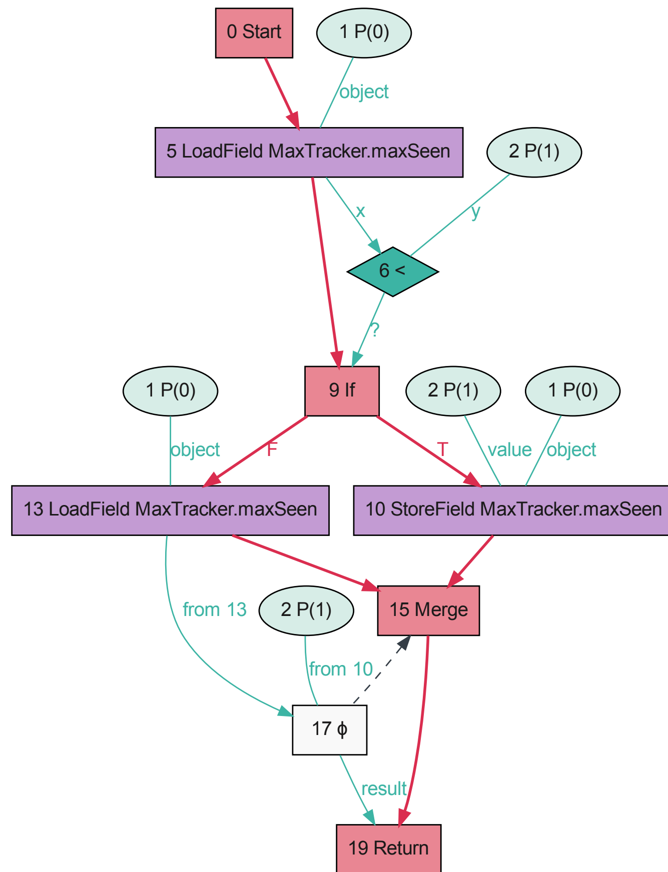
(a) Source code

(b) Graal IR

Figure 2.10: Source code and IR for a `Counter` and its `count` method.

```
 1  class MaxTracker {
 2    int maxSeen = 0;
 3    int max(int x) {
 4      int result = 0;
 5      if (x > maxSeen) {
 6        maxSeen = x;
 7        result = x;
 8      } else {
 9        result = maxSeen;
10      }
11      return result;
12    }
13  }
```

(a) Source code



(b) Graal IR

Figure 2.11: Source code and IR for a `max` method that tracks the largest value seen.

invocations. The graph starts with a `LoadField`, which reads the `state` field from the receiver (`P(0)`). Then, the `StoreField` stores `state + 1` back into the `state` field of the receiver. Finally, another `LoadField` reads the new value of `state` and returns it.

The examples so far have had simple control flow with no branches. The example in Figure 2.11 contains a control flow split and merge. The `MaxTracker` class tracks the largest value passed to its `max` method in a `maxSeen` field. The `max` method returns the larger value between its parameter `x` and `maxSeen`.

The IR in Figure 2.11 represents the `max` method. Since the method starts with an if-statement, it must first evaluate the condition. The program reads `maxSeen` from the receiver (`P(0)`) and compares it against `x` (`P(1)`). The result of this comparison is passed to the `If`, at which point there is a control flow split. If the condition is `true`, control flows through the `T` edge, where the `maxSeen` field is updated to the value of `x` and `result` is assigned the value of `x`. If the condition is `false`, control flows through the `F` edge, and `result` is assigned the value of `maxSeen`. Then, in either case, control flows into the `Merge` node and a value is returned.

Recall that local variables are not present in the Graal IR; instead, the value of a variable at a particular point in time is represented by a node in the graph. This presents a problem: after control flow merges, the variable `result` takes on different values depending on which branch was taken. Graal IR (and SSA in general) models such a variable with a phi node (ϕ). When unambiguous, this section uses ϕ_{x} to denote the phi node that represents a variable `x`.

A ϕ is always associated with a control flow merge point. It has data inputs for the value of a variable on each path that flows to the merge point. The labels on the edges (e.g., "from `10`") indicate the association between data edges and control flow paths. For example, when the condition in Figure 2.11 is `true`, control flows to the merge from the `StoreField` (node `10`); in this case, ϕ_{result} takes on the value of `x` (`P(1)`). When the condition is `false`, control flows to the merge from the `LoadField` (node `13`), and ϕ_{result} takes on the value of `maxSeen`.

Phi nodes are common when a code has loops. A loop can run for an indeterminate number of iterations, or even no iterations at all, so the value of a variable depends on the control flow path taken. Graal IR models loops with a few nodes: `LoopBegin` denotes the beginning of the loop, `LoopEnd` denotes a back-edge to the beginning of the loop, and `LoopExit` denotes a point where control exits the loop.

Consider the iterative `factorial` function in Figure 2.12. Given that two variables are modified in the loop, it contains two ϕs: ϕ_{x} (node `14`) and ϕ_{result} (node `15`). When control first reaches the loop (via the `If`, node `8`), ϕ_{x} takes on the initial value of `x` (`P(1)`), and ϕ_{result} takes on the value `1`.

At the start of the loop, the condition is checked. Note that `x > 1` has been canonicalized to `!(x < 2)`, so when `x < 2` is `false` the loop body executes, and when it is `true` the loop is exited. In this case, executing the loop involves simply jumping back to the beginning of the loop (via the back-edge from `LoopEnd` to `LoopBegin`). However, that does not mean nothing is executed: every time control flows into `LoopBegin`, its ϕs are updated. This time, control flows in from node `28`, so ϕ_{result} is updated to `result * x` and ϕ_{x} is updated to `x + (-1)`.

The loop continues iterating this way until `LoopExit` is hit. After the loop exit, the computed value of `result` is returned[5].

The Graal IR must also support method calls. It uses a `Call` node and an associated `MethodCallTarget` to represent each call. These nodes contain metadata about the call, such as the name, signature, and invoke kind. The arguments to the call are indicated by data flow edges into the `MethodCallTarget`. The call itself can produce a value.

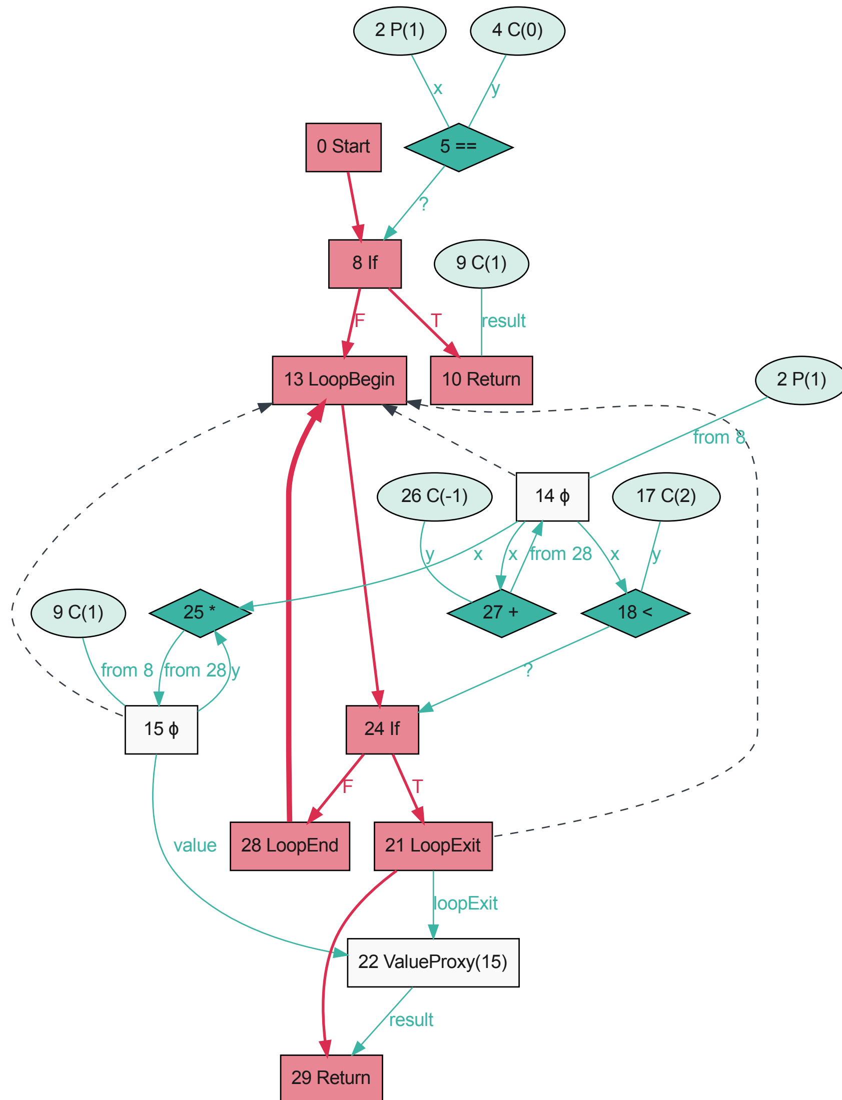
Another implementation of `factorial` that uses recursion is depicted in Figure 2.13. The call in this method takes two arguments: the receiver (`P(0)`) and the value of `x-1`. It produces a value (the result of `factorial(x-1)`), which is multiplied by `x` to obtain the return value.

One limitation of method calls is that they are opaque: the compiler cannot perform optimization across a method call. This restriction is especially limiting in a language like Java where method-based abstractions (interfaces, template methods, adapters, etc.) are

[5] The `ValueProxy` forwards a value computed inside a loop to a point outside of it. The technical reasons for this node are not relevant to us.

```
1  int factorial(int x) {
2    if (x == 0) return 1;
3    int result = 1;
4    while (x > 1) {
5      result *= x;
6      x -= 1;
7    }
8    return result;
9  }
```

(a) Source code

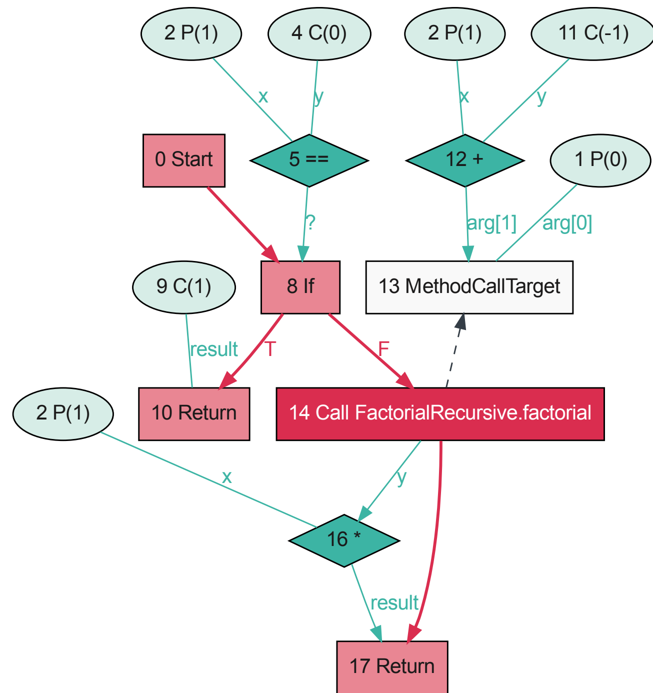


(b) Graal IR

Figure 2.12: Source code and IR for an iterative `factorial` method.

```
1  int factorial(int x) {
2    if (x == 0) return 1;
3    else return x *
         factorial(x-1);
4  }
```

(a) Source code



(b) Graal IR

Figure 2.13: Source code and IR for a recursive `factorial` method.

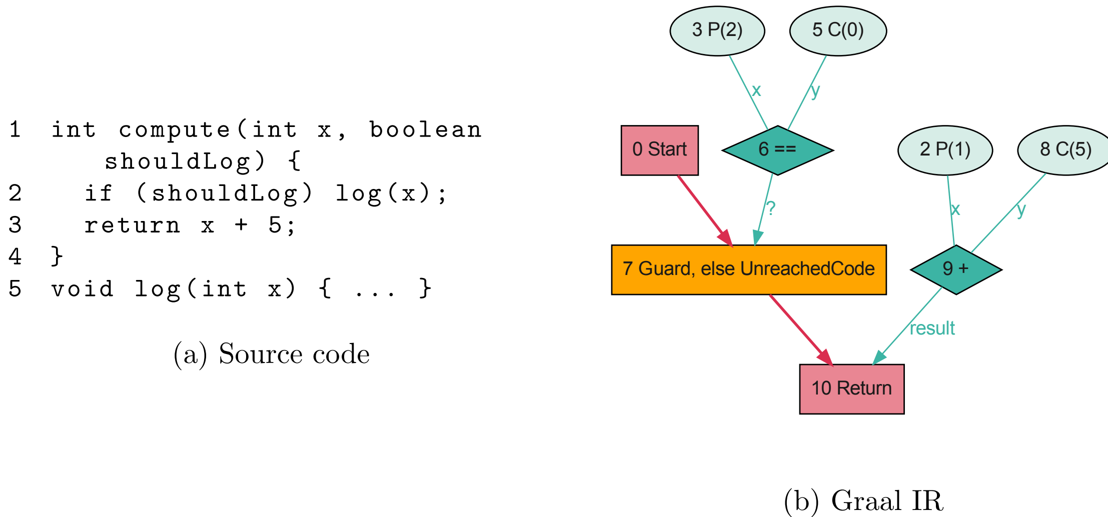
abundant. Graal attempts to inline method calls when possible (Section 2.3.2). Recursive methods are one case where inlining is not possible.

Graal makes extensive use of speculation. This speculation is modeled by `Guard` nodes in the IR. Graal creates guards to assert certain program properties; for example, that a branch is unreachable, that a value has a particular type, or that an exception is never thrown. By inserting these guards, Graal can simplify the graph and produce more efficient code.

For example, consider the `compute` method in Figure 2.14, which optionally logs information depending on its `shouldLog` parameter. If the run-time profiling indicates that `shouldLog` is rarely `true`, Graal may speculate that the logging branch is unlikely. It inserts a `Guard` asserting that `shouldLog` is `false`[6], and then completely removes the call to `log`. As long as the guard condition continues to hold, the compiled code remains valid.

Of course, guards do fail. When this happens, the code *deoptimizes*, transferring control from the compiled code back to the JVM's interpreter. Deoptimization is a complicated process: in order to resume execution in the interpreter, the compiled code must reconstruct the state of the interpreter at the deoptimization point. The IR contains `FrameState` nodes that track this data. `FrameState` nodes are omitted from the graphs in this thesis for simplicity, but they are nevertheless an essential aspect of Graal's speculative IR.

[6] Booleans can be `0` (`false`) or `1` (`true`).




```
1  int compute(int x, boolean
      shouldLog) {
2    if (shouldLog) log(x);
3    return x + 5;
4  }
5  void log(int x) { ... }
```

(a) Source code

(b) Graal IR

Figure 2.14: Source code and IR for a method with dynamically unreachable code.

```
1   class Point { double x, y; }
2   interface Transform {
3     Point apply(Point p);
4   }
5   class Translate implements Transform {
6     double dx, dy;
7     Point apply(Point p) {
8       return new Point(p.x + dx, p.y + dy);
9     }
10  }
11  // more Transforms
```

Figure 2.15: Definition of a `Point` class and a hierarchy of `Transforms`.

2.3.2 Optimizations

Graal contains an extensive suite of compiler optimizations [41]. Two of them are particularly important: type-checked inlining and partial escape analysis. The definitions in Figure 2.15 are used in the discussion that follows. There is a `Point` class comprising two `double` fields, and a `Transform` interface, which takes an input `Point` and produces a transformed `Point`.

Type-checked inlining: Virtual method calls are prevalent throughout Java code. The target of these virtual calls changes depending on the run-time type of the receiver, so the compiler cannot easily inline such a call. If the type profile for a call site has only a few concrete types, Graal may assume that compiled code only encounters these types and inline the method targets for those types. It inserts type guards to validate the assumption. Then, the compiled code avoids the dynamic method dispatch, and Graal can optimize across the call.

For example, in Figure 2.16, suppose `applyTransform` is compiled, and the profile

for `t` only ever observes the type `Translate`. Graal may use this profile to speculate that `t` is only ever a `Translate` instance. If so, it can insert a guard to check this assumption, and then inline the `apply` implementation for `Translate` directly into `applyTransform`. In the IR, the guard has been lowered to an `If` (node `21`) with an `InstanceOf` check (node `19`). If the check fails, the code deoptimizes (node `22`); otherwise, it executes the body of `Translate.apply`, which allocates a new `Point` with coordinates translated by `t`'s `dx` and `dy` fields. The compiled code is used for each call as long as `t` continues to be a `Translate` instance.

Partial escape analysis: Another significant source of overhead in Java is object allocations. *Escape analysis* is a technique to address this [12]: if an object does not "escape" the method that allocates it (e.g., by being returned, passed to another method, or stored in a field), certain optimizations can be performed. One important optimization is *scalar replacement of aggregates*, whereby the compiler elides an object allocation and stores the object's fields as locals[7].

Graal has its own concept of *partial escape analysis* [43] which is flow-sensitive. When an object escapes only on certain branches, Graal can still perform scalar replacement in the non-escaping branches.

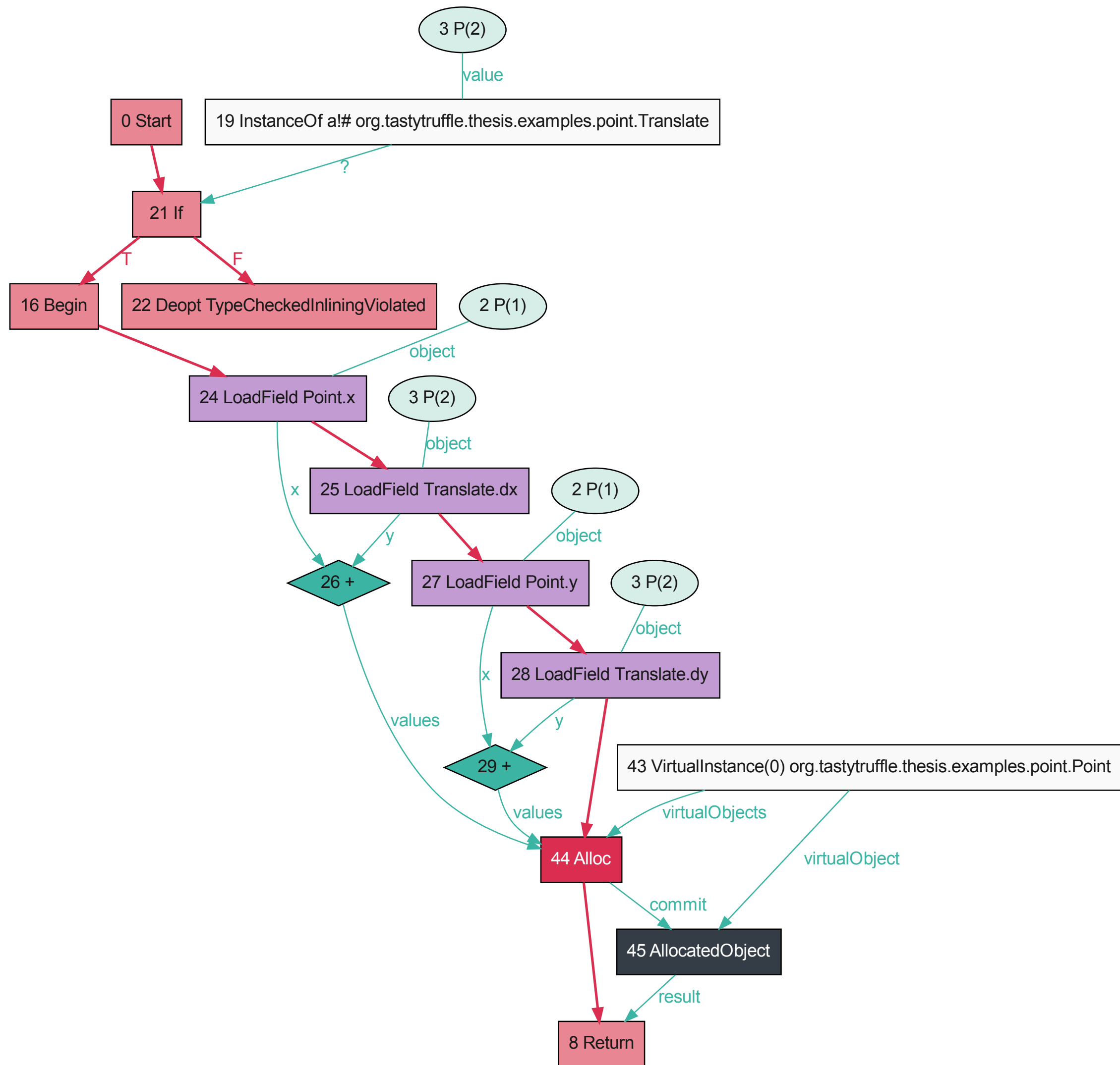
For example, in the method in Figure 2.17, local `p` contains a newly-allocated `Point` object. This object escapes in the `true` branch of the if-statement since it is stored in a static `escapedPoint` field; however, in the `false` branch, the point does not escape the method. Graal's escape analysis detects this and performs scalar replacement. The `Point` is not allocated immediately; instead, Graal tracks that `p.x` has the value of `x` (`P(1)`), and `p.y` has the value of `x * 2` (node `7`), and the field read in the `false` case can return the value of `x * 2` directly. An object allocation (represented by `Alloc`) is still required in the `true` branch, but it can be deferred until control actually enters the branch.

Compiler optimizations can have an enabling effect on other optimizations. For example, in Figure 2.18, suppose again that `t` is always an instance of `Translate`. If the call to `apply` does not get inlined, the object escapes its allocation site (since it is created and returned by `apply`) and must be allocated. However, after Graal inlines `apply` using type-checked inlining, the allocated `Point` *does not* escape this method, and the allocation can be elided. Note the lack of `Alloc` nodes in the IR. Since these optimizations can have such a compounding effect on performance, it is important for performance-critical code to be written in a way that is amenable to Graal's optimizations. This rquirement has important implications for Truffle interpreters, which are discussed in the next section.

[7] If any of the fields are unused, the compiler can also optimize the corresponding locals away.

```
1  Point applyTransform(Point p, Transform t) {
2    return t.apply(p);
3  }
```

(a) Source code

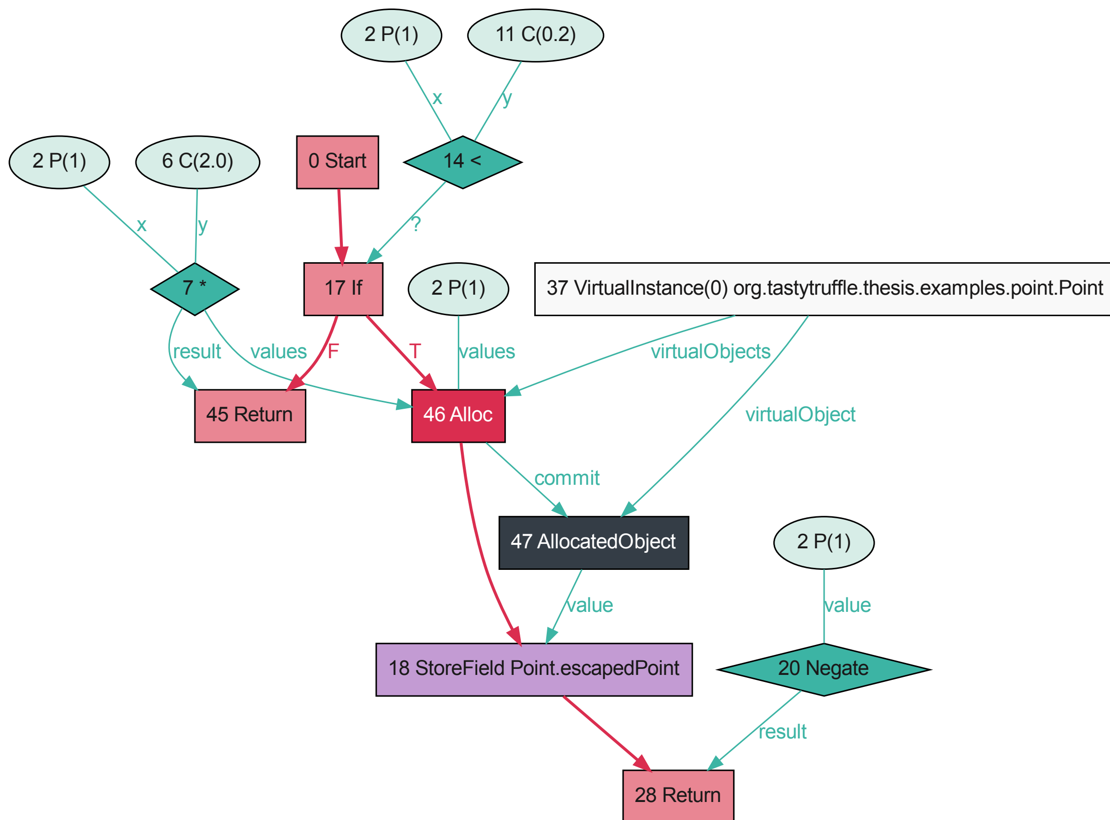


(b) Graal IR

Figure 2.16: Source code and IR for a method transformed by type-checked inlining.

```
1  double doSomething(double x) {
2    Point p = new Point(x, x * 2);
3    if (p.x < 0.2) {
4      Point.escapedPoint = p;
5      return -p.x;
6    } else return p.y;
7  }
```

(a) Source code

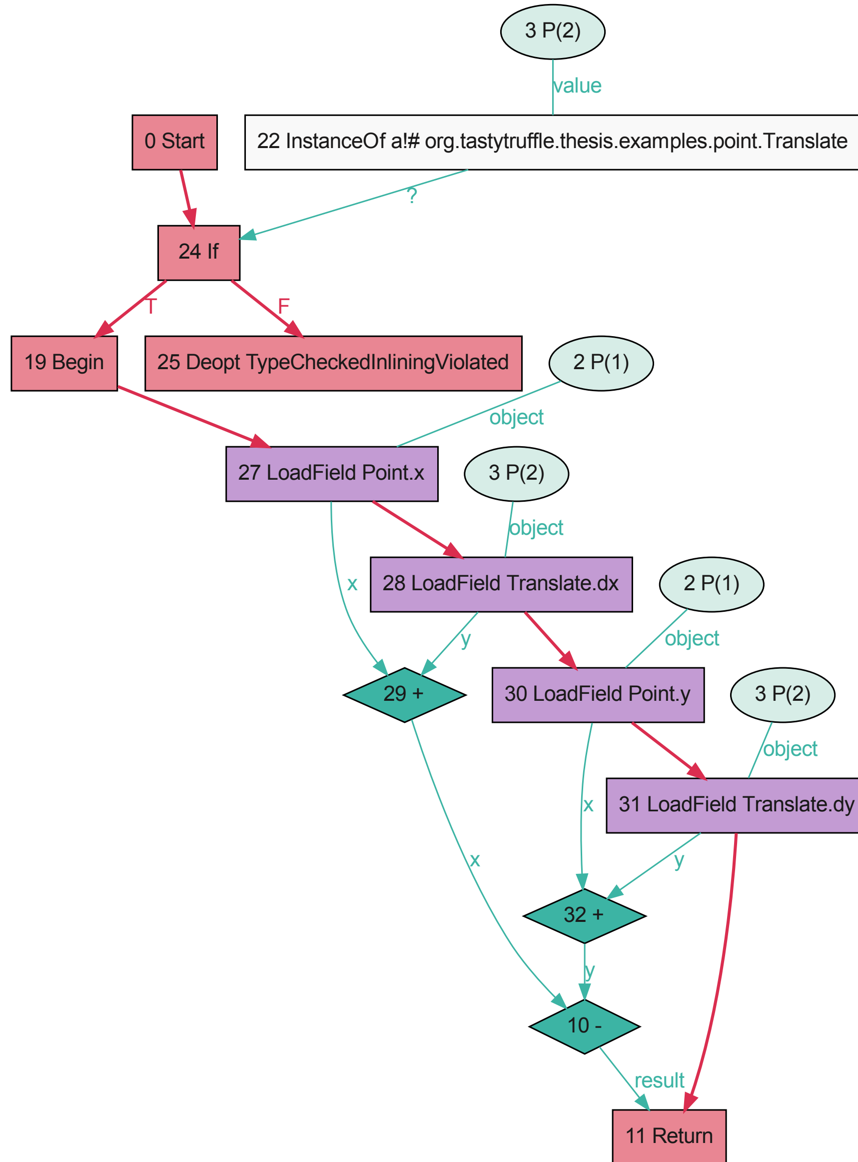


(b) Graal IR

Figure 2.17: Source code and IR for a method transformed by partial escape analysis.

```
1  double doSomething(Point p, Transform t) {
2    Point result = t.apply(p, t);
3    return result.x - result.y
4  }
```

(a) Source code



(b) Graal IR

Figure 2.18: Source code and IR for a method transformed by type-checked inlining and partial escape analysis.

2.4 The Truffle ecosystem

Implementing a programming language is a complicated endeavour requiring significant engineering effort. It is even more challenging to make the implementation performant. The Truffle framework [48] aims to make high-performance language implementations possible with "modest effort".

This section introduce the basics of the Truffle framework. The Truffle ecosystem comprises several components, including (but not limited to) a domain-specific language (DSL) for writing interpreters, a set of general-purpose libraries, and a custom front-end to the Graal compiler, which plays an important role in Truffle's performance. The goal in this section is to introduce Truffle and disentangle the complexity. Figure 2.8 from the previous section depicts the relation between Java programs and Graal; Figure 2.19 extends this diagram to incorporate Truffle.

The discussion begins with Truffle interpreters, which take the form of abstract syntax trees (ASTs) written in Java. AST interpreters are relatively intuitive to implement, since a node's semantics often follow from its syntax. Next, an overview of Truffle's partial evaluation (PE) [47] is given. Truffle compiles hot ASTs using Graal, but ASTs are difficult to optimize due to their indirection. Partial evaluation removes this indirection (and performs other optimizations), which allows Graal to generate much more efficient code. The *self-optimizing* design of Truffle interpreters [49] is also discussed. Truffle interpreters can change their behaviour depending on the run-time characteristics of the interpreted program, for example, by specializing a node when it only receives specific types of inputs. Truffle defines a domain-specific language (DSL) [26] and an accompanying processor that runs at compile-time to generate these self-optimizing interpreters. Finally, Truffle's object model is briefly discussed. This model allows interpreters to support guest language objects (class instances, structs, etc.) efficiently; furthermore, it plays well with Graal's escape analysis so that guest object allocations can be elided.

2.4.1 Truffle interpreters

For clarity, it is worth introducing some terminology. The language implemented by an interpreter is the *guest language*. In contrast, the language the interpreter is written in is the *host language*. For Truffle interpreters, the host language is Java. Since Truffle interpreters are Java programs, they can leverage many features of the Java runtime rather than implement them from scratch: automatic memory management, exceptions, and just-in-time (JIT) compilation, to name a few.

Truffle is designed around abstract syntax tree (AST) interpreters. Each AST's root node usually represents a function in the guest language. An example AST is shown in Figure 2.20a. This tree represents the function `f(x) = x * 2`.

Figure 2.20b depicts the source code for these nodes. Each node defines an `execute` method describing how it should be executed[8]. For example, the `Constant` node simply returns the value of its `constant` field.

[8] To fit Truffle's API, `RootNodes` must return an `Object`, but other nodes are free to define their own return type.

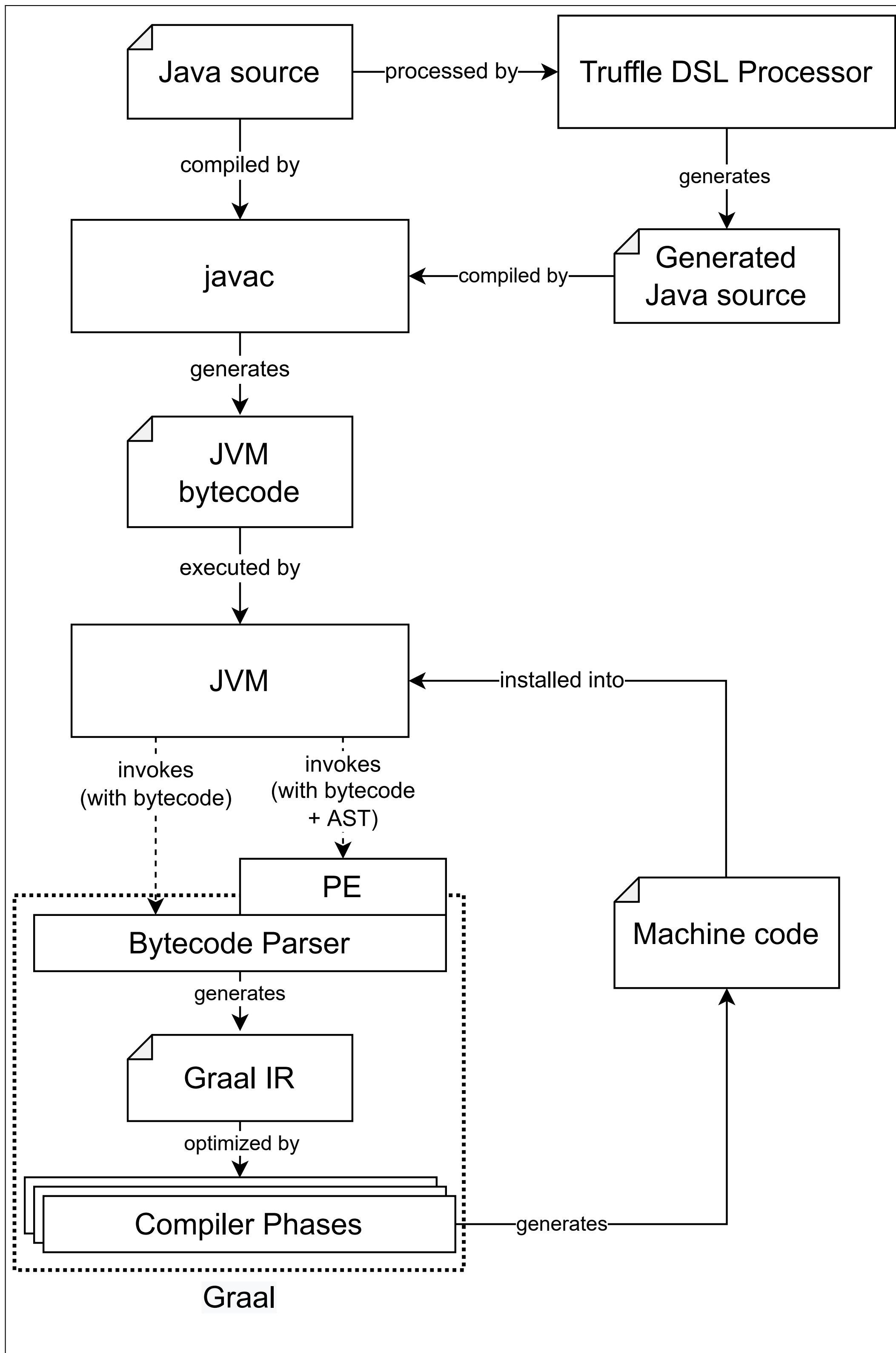
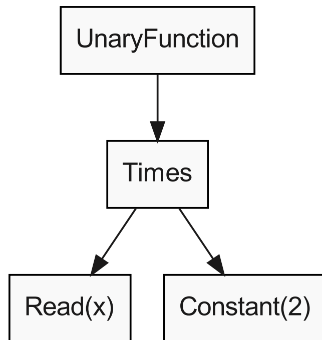


Figure 2.19: High-level view of the Truffle ecosystem.



(a) AST for a function `f(x) = x * 2`

```
1  class Constant extends Node {
2    int constant;
3    int execute(VirtualFrame frame) {
4      return constant;
5    }
6  }
7  class Read extends Node {
8    String name;
9    int index;
10   int execute(VirtualFrame frame) {
11     return frame.getInt(index);
12   }
13 }
14 class Times extends Node {
15   @Child Node left;
16   @Child Node right;
17   int execute(VirtualFrame frame) {
18     return left.execute(frame) *
19         right.execute(frame);
20   }
21 }
22 class UnaryFunction extends RootNode {
23   @Child Node body;
24   Object execute(VirtualFrame frame) {
25     frame.setInt(0, (int) args[0]);
26     return body.execute(frame);
27   }
28 }
```

(b) Source code

Figure 2.20: An example AST and the source code for its nodes.

A node may access local state through the `frame` parameter, which stores all of the local variables of a guest method. During AST construction, each local is assigned a unique index in the frame. When a `RootNode` like `UnaryFunction` is called, Truffle allocates a frame with an array to store its locals. Then, nodes can access locals by reading from and writing to specific indices in the `frame`. The function in Figure 2.20 has a single local `x`, so its frame would have an array of length 1. The `UnaryFunction` node stores the parameter `x` into index `0` of the frame; later, the `Read` node reads it.

A node's behaviour may depend on the behaviour of other nodes. It stores these nodes as fields with the `@Child` annotation. When the `UnaryFunction` node is executed, it invokes its `body`'s `execute` method. This node is a `Times` node that also invokes the `execute` methods of its `left` and `right` children. The `frame` gets passed along through each `execute` call so children can access local state.

2.4.2 Partial evaluation

The primary limitation of AST interpreters is poor performance. ASTs contain significant indirection: chiefly, executing an AST requires many calls to different `execute` methods. Each of these calls requires a virtual method dispatch, and these calls are expensive in aggregate.

An obvious solution might be to compile ASTs with a just-in-time compiler (JIT), but it would be ineffective. Since every node overrides the same `execute` method, each `execute` call site is highly polymorphic, and a JIT would have a difficult time performing any optimizations. For example, every `UnaryFunction` can have a different `body`, and so the call to `body.execute` can resolve to arbitrarily many different `execute` implementations. So, it would not be effective to compile `UnaryFunction`'s `execute` method.

The important insight is that, for a *particular* AST, these call sites are usually stable. The `UnaryFunction` for `f(x) = x * 2` always has the `Times` node as its `body`, and the `Times` node always has the `Read` and `Constant` nodes as its children. Assuming an AST stays constant, many of the virtual calls (to `execute` or otherwise) actually resolve to statically known implementations, and each call can be *devirtualized* by inlining the implementation at the call site.

This approach is the essence of Truffle's *partial evaluation* [22, 47]. Partial evaluation (PE) is a general technique to specialize a program with respect to its statically determined inputs. For Truffle, PE assumes a particular AST is stable (with help from directives like `@Child`) and devirtualizes as many calls as it can. For this reason, the original Truffle paper describes partial evaluation as "compilation with aggressive method inlining" [48].

Of course, it is possible for PE's assumptions to be wrong, in which case any partially-evaluated code becomes invalid. A Truffle AST undergoes partial evaluation when it is parsed into Graal IR, immediately before being compiled, so PE can leverage Graal's existing support for deoptimization (Section 2.3). When control reaches an invalid code path, the code deoptimizes and resumes execution in the Truffle interpreter.

```
1 Object execute(VirtualFrame frame) {
2   frame.setInt(0, (int) args[0]);
3   return body.execute(frame);
4 }
```

(a) Before partial evaluation.

```
1 Object execute(VirtualFrame frame) {
2   frame.setInt(0, (int) args[0]);
3   return body.left.execute(frame) *
4       body.right.execute(frame);
5 }
```

(b) After `body.execute` is inlined.

```
1 Object execute(VirtualFrame frame) {
2   frame.setInt(0, (int) args[0]);
3   return frame.getInt(body.left.index) *
4       body.right.constant;
5 }
```

(c) After `body`'s child methods are inlined.

```
1 Object execute(VirtualFrame frame) {
2   frame.setInt(0, (int) args[0]);
3   return frame.getInt(0) * 2;
4 }
```

(d) After constant folding.

Figure 2.21: Pseudocode for `f(x) = x * 2` (from Figure 2.20) after partial evaluation.

Example: Partial evaluation of `f(x) = x * 2`

Consider the example in Figure 2.20. The `UnaryFunction` calls `execute` on its `body`, a `Times` node, and the `Times` node calls `execute` on its `left` and `right` children. The children are effectively constant, but partial evaluation cannot conservatively assume this. Truffle interpreters communicate these sorts of invariants to PE using directives. In this case, the `@Child` annotations tell PE that the nodes do not change. This information allows PE to inline the implementations.

Figure 2.21 demonstrates how `UnaryFunction`'s `execute` method gets partially evaluated. Partial evaluation is a monolithic process during the conversion from bytecode to Graal IR (Section 2.3.1), so the figure instead uses pseudocode to incrementally illustrate how PE works.

Before partial evaluation, the code is as shown in Figure 2.21a. The inlining process first sees `body.execute`, and since `body` is a `@Child`, the method is inlined (Figure 2.21b). For the same reasons, PE can then inline the calls on `body.left` and `body.right` (Figure 2.21c). Inlining is applied to all calls that can be devirtualized—not just `execute` methods—so helper methods can be inlined as well if they are used. The resultant code has no `execute` indirection: the interpreter has been specialized for the AST.

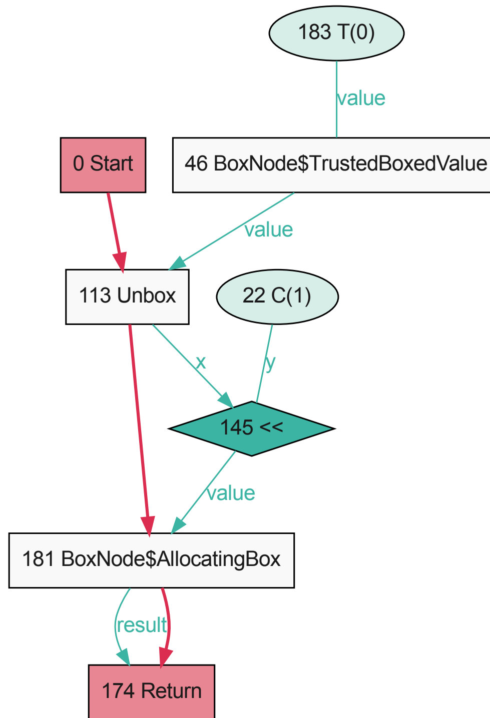


Figure 2.22: IR for `f(x) = x * 2` after partial evaluation and escape analysis.

Partial evaluation also performs constant folding. If a field is constant, PE can replace field reads with the value stored in the field. Suppose `Constant`'s `constant` field and `Read`'s `index` fields were marked `final`. Partial evaluation could further simplify the code, replacing the field reads with constants (Figure 2.21d). Sometimes, fields are mutable but effectively immutable (e.g., due to lazy initialization); Truffle provides a `@CompilationFinal` annotation for such cases.

Partial evaluation has an important effect on frames: since each `execute` call that passes the frame to a child node gets inlined, the frame usually does not escape the compiled code. Graal can perform scalar replacement (Section 2.3.2) on these frames, and then frame accesses have the same overhead as regular variable accesses. This is another important source of indirection that PE allows the compiler to remove.[9]

Figure 2.22 depicts the IR for `f(x) = x * 2` after Graal's escape analysis phase. The single argument to the Truffle AST, `T(0)`, represents `x`. `T(0)` is a boxed `Integer` (since an `int` is automatically boxed where an `Object` is expected), so it is first unboxed. Then, it is multiplied by `2`, which is canonicalized by Graal into a single left bit-shift. Finally, it is re-boxed (since an `Object` is expected), and returned.

Partial evaluation makes ASTs much more amenable to optimization by Graal. Ex-

[9]Scalar replacement of frames is so important to performance that the partial evaluator actually aborts compilations if the frame escapes. Developers must invoke a `materialize` method to explicitly declare their intention to let the frame escape.

```
 1 class LazyLoadedConstant extends Node {
 2   @CompilationFinal int constant;
 3   @CompilationFinal boolean loaded = false;
 4
 5   int execute(VirtualFrame frame) {
 6     if (!loaded) {
 7       CompilerDirectives.transferToInterpreterAndInvalidate();
 8       constant = compute();
 9       loaded = true;
10     }
11     return constant;
12   }
13
14   int compute() { ... }
15 }
```

Figure 2.23: Source code for a node with a deoptimization directive.

cluding the boxing, this compiled code is practically free of indirection. Even the boxing can be removed when `RootNode`s are inlined into each other. For example, if `f(x) = x * 2` gets inlined into another AST, Graal can eliminate the redundant box-unbox sequences that occur at the beginning and end of the function.

Compilation boundaries

Sometimes, a `@Child` or `@CompilationFinal` field needs to be modified[10]. However, if an AST has been compiled with the assumption that the field does not change, changing the field could affect the correctness of the program. In this situation, the AST can use a compiler directive to deoptimize to the interpreter and invalidate any compiled code.

For example, the node in Figure 2.23 returns a lazily-computed constant. The constant is marked `@CompilationFinal` so that it can be constant-folded by PE. If it is `loaded`, PE folds the if-statement away as dead code and the code simplifies to a return of the `constant`. However, if the node does not get executed before compilation, it must compute the constant the first time it runs. This operation invalidates the `@CompilationFinal` guarantee, so a `transferToInterpreterAndInvalidate` directive is inserted at the top of the block. When executing in the interpreter, this directive is a no-op; in compiled code, it is replaced by a deoptimization point. Code after the deoptimization point is not included in compilation. If the deoptimization point is reached, any compiled code that executes the node is invalidated, and the current invocation resumes execution in the interpreter where it can compute the constant. The node can later be recompiled with the loaded constant.

Truffle interpreters must also be careful about what code gets inlined by partial evaluation. Code that calls into complex library code should not be inlined, lest a significant amount of code (too much to properly optimize) get included in the compilation. A good example is `HashMap` accesses, which invoke the key's `hashCode` method and search a table us-

[10] Compilation-constant fields are often modified during self-optimization (Section 2.4.3).

```
 1 class Cache extends Node {
 2   @Child Node arg;
 3   @Child Node body;
 4   @CompilationFinal int size;
 5   @CompilationFinal(dimensions = 1) int[] cacheKeys;
 6   @CompilationFinal(dimensions = 1) int[] cacheValues;
 7
 8   @ExplodeLoop
 9   int execute(VirtualFrame frame) {
10     int argument = arg.execute(frame);
11     for (int i = 0; i < size; i++) {
12       if (cacheKeys[i] == argument) return cacheValues[i];
13     }
14     int result = body.execute(frame);
15     if (CompilerDirectives.inInterpreter() &&
16         size < cacheKeys.length) {
17       cacheKeys[size] = argument;
18       cacheValues[size] = result;
19       size++;
20     }
21     return result;
22   }
23 }
```

Figure 2.24: Source code for a `Cache` node.

ing chaining. A method that performs such calls can be annotated with `@TruffleBoundary` to get excluded from partial evaluation.

Loop unrolling and dead code elimination

Constant folding can sometimes allow PE to eliminate dead branches of code altogether. For example, if PE determines an `if` condition is always `true`, it can omit the `else` branch from the IR entirely. The Truffle `inInterpreter` directive is useful for guarding branches that should not be compiled, because it is always `false` to PE.

Truffle also supports loop unrolling. When a node has loops that run for a compilation-final number of iterations, it can use an `@ExplodeLoop` directive to instruct the bytecode parser to unroll the loop.

Take, for instance, the `Cache` node in Figure 2.24. This node wraps a `body` node, caching its results to avoid repeated computations. It uses arrays to cache arguments and results, and loops through the arrays to search for previously-computed results. The `execute` method is annotated with `@ExplodeLoop`.

Suppose `Cache` has `x * 2` as its body, and it gets compiled with cache entries for `2` and `4`. The resultant IR is shown in Figure 2.25. The loop gets unrolled because of the `@ExplodeLoop` annotation. Now, the IR contains two `If` nodes which compare the input argument against the keys in the cache. If there is a match, a cached value is returned, otherwise a result is computed. The `CompilationFinal(dimensions=1)` annotations cause PE to treat the array *contents* as compilation constants, so the array reads are replaced

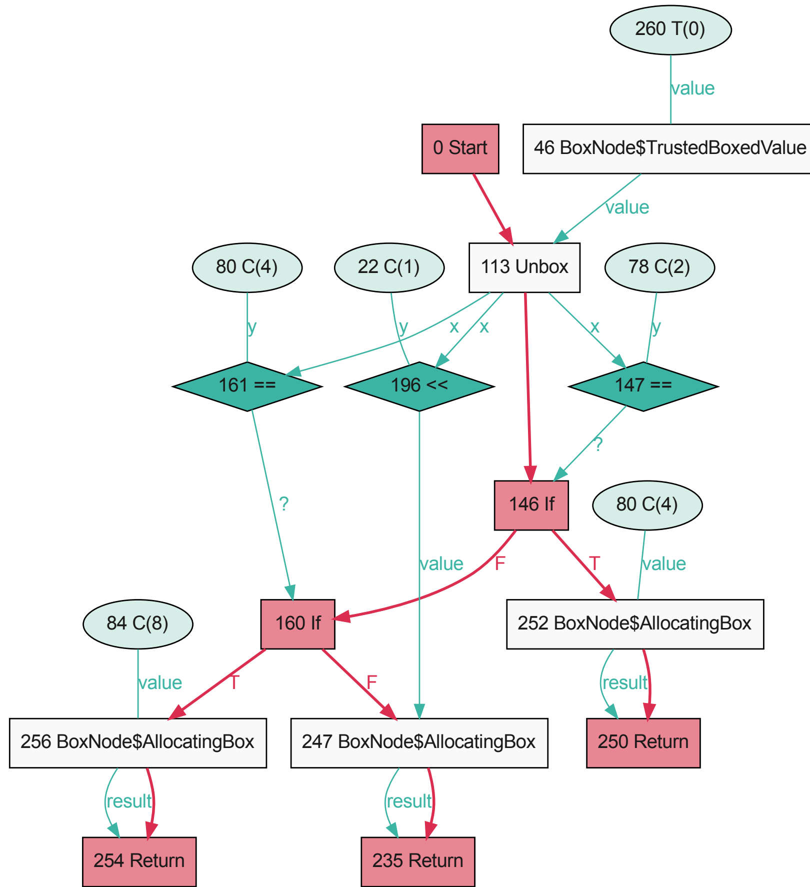


Figure 2.25: Graal IR for a `Cache` of `x * 2`.

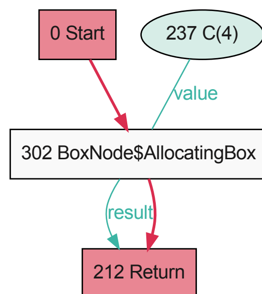


Figure 2.26: Graal IR from Figure 2.25 after dead code elimination.

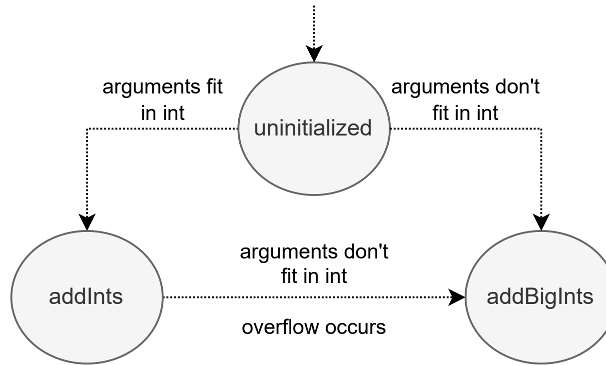


Figure 2.27: State machine for a self-optimizing `Plus` node.

by the cache keys and values directly. The cache update is guarded by the `inInterpreter` directive, so it does not get compiled.

Loop unrolling is especially effective in combination with dead code elimination. Suppose PE determines from some outside context that the argument to the `Cache` is always `2`. It can determine that the first comparison in Figure 2.25 (node `147`) always yields `true` and eliminate the checks entirely. The resulting IR, in Figure 2.26, immediately returns the value `4` from the cache[11]. By leveraging loop unrolling and dead code elimination, PE effectively converts a linear search to a constant-time access in the compiled code.

2.4.3 Self-optimizing ASTs

A challenge with implementing languages efficiently is that they can be highly dynamic. An interpreter may theoretically need to support a wide range of (potentially expensive) operations, but if only certain cases are observed during execution, it is more efficient to only handle the cases that actually occur. Truffle nodes are *self-optimizing* [49]: a node can have multiple *specializations*[12], each defining a different way to execute the node. By profiling the behaviour they observe during execution, nodes can automatically select which specialization should run. Self-optimization is tedious to implement manually, so Truffle provides a DSL to generate these nodes automatically. Truffle's DSL processor generates these nodes during regular Java source compilation.

Example: A self-optimizing `Plus` node

A simple example is a `Plus` node for a language with arbitrary precision integers (i.e., Java `BigInteger`s). `BigInteger` arithmetic is more expensive than arithmetic over 32-bit `int`s, so if a `Plus` node's operands and results fit into an `int`, it is preferable to perform the arithmetic using `int`s.

Abstractly, the `Plus` node should implement the state machine in Figure 2.27. On first execution, it is uninitialized. If its arguments fit into `int`, it specializes itself to add `int`s;

[11] Since the expression to compute is just `x * 2`, PE could have trivially produced `4` using constant folding, but this caching technique generalizes for any deterministic computation.

[12] A Truffle specialization is conceptually different from a generic specialization (Section 2.1.1).

```
1  @NodeChild("left")
2  @NodeChild("right")
3  abstract class Plus extends Node {
4
5    @Specialization(rewriteOn = ArithmeticException.class)
6    int addInts(int leftValue, int rightValue) {
7      return Math.addExact(leftValue, rightValue);
8    }
9
10   @Specialization(replaces = "addInts")
11   BigInteger addBigInts(BigInteger leftValue, BigInteger rightValue){
12     return leftValue.add(rightValue);
13   }
14 }
```

Figure 2.28: Source code for a self-optimizing `Plus` node.

otherwise, it specializes itself to add `BigInteger`s. If the node is specialized to `int`s and the addition overflows or its arguments suddenly do not fit into `int`, it should re-specialize itself to add `BigInteger`s. If the arguments continue to fit in `int` and the addition does not overflow, it can stay in the `addInts` specialization.

This behaviour can be implemented in Truffle as shown in Figure 2.28. The node defines two implementations `addInts` and `addBigInts` that operate on `int`s and `BigInteger`s respectively. Each version is annotated as a `@Specialization`. `Plus` also declares its two children as `@NodeChild`, which indicates that their results should be computed and passed as arguments to the specializations (unlike `@Child`).

Specializations define *guards* that determine when they are applicable or need to be replaced. The specializations' parameter types serve as *type guards* to decide which specialization to use. For example, `addInts` can only be used when the child nodes return `int` values, and `addBigInts` can only be used when the child nodes return `BigInteger` values[13]. Another example is the *event guard* in the annotation for `addInts`. Integer arithmetic has the potential to overflow, in which case it throws an `ArithmeticException`[14]. The `rewriteOn` field indicates that the node should re-specialize itself if such an exception is thrown.

Truffle uses these specializations and their guards to generate an `execute` method with some internal state fields to track which specialization is enabled. The generated code is complicated, so it is not presented here, but it effectively implements the state machine in Figure 2.27. The `execute` method is made up of different branches that inspect the node's internal state to determine which specialization to invoke. The internal state fields are marked `@CompilationFinal`, which allows PE to simplify `execute` to just the active specialization and its guards.

[13] Truffle supports implicit conversions, so if one child returns a `BigInteger` and the other an `int`, the interpreter can coerce the `int` to a `BigInteger` so that `addBigInts` applies.

[14] `Math.addExact` detects overflow and throws `ArithmeticException`, but the standard `+` operator does not.

```
1  @NodeChild("argument")
2  class DeterministicComputation extends Node {
3
4    @Specialization(guards = "argument == cacheKey")
5    int computeCached(
6        int argument,
7        @Cached("argument") int cacheKey
8        @Cached("compute(cacheKey)") int cacheValue
9    ) {
10     return cacheValue;
11   }
12
13   @Specialization(replaces = "computeCached")
14   int computeUncached(int argument) {
15     return compute(name);
16   }
17
18   int compute(int argument) { ... }
19 }
```

Figure 2.29: Source code for a node that caches deterministic computations.

Caching

Another useful feature of Truffle specializations is caching. A specialization can declare `@Cached` parameters whose values are computed once and reused for subsequent invocations of the specialization. In combination with *expression guards*, caches can avoid repeating computations when the results are stable.

An example is presented in Figure 2.29. The `DeterministicComputation` node performs a potentially-expensive computation. Since it is deterministic, its results can be cached to avoid recomputation[15].

When the node is first executed, it selects the `computeCached` specialization. The `computeCached` specialization declares two `@Cached` parameters which get initialized using the expressions in the annotations. It initializes `cacheKey` to the value of `argument`, the parameter obtained by executing the child node. Then, it initializes `cacheValue` to the result of `compute(cacheKey)`; that is, the node performs the computation and caches the result. The body of the specialization simply returns `cacheValue`. On subsequent executions, this specialization can be used as long as the *expression guard* does not fail. In this case, as long as the input `argument` is equal to `cacheKey`, `DeterministicComputation` can return the same constant without recomputing it.

There can be multiple cache entries. If `DeterministicComputation` encounters a different value for `argument`, it can compute and cache this second result as well. The code generated for the `computeCached` specialization loops through the cached values for `cacheKey` and `cacheValue` until it finds a match. If there is no match, it can continue to add to the cache, until a preset cache limit is reached, at which point it re-specializes to

[15] `DeterministicComputation` fulfills a similar role to the `Cache` from Figure 2.24, but the code is much simpler.

the `computeUncached` specialization and performs the computation every time.

The generated code for `DeterministicComputation` plays well with partial evaluation. As with `Cache` (Figure 2.24), `execute` is annotated with `@ExplodeLoop` and the cache entries are `@CompilationFinal`. Thus, PE replaces cache lookups with comparisons against each `cacheKey`; if the comparison succeeds, the corresponding `cacheValue` is returned directly.

This `@Cache` idiom is especially useful for implementing polymorphic inline caches [25]. Virtual call nodes can cache the result of a virtual dispatch (i.e., a call target), using the receiver type as the cache key. If such a call site is monomorphic, PE can replace the call with the call target's body guarded by a simple type check (usually a pointer comparison).

2.4.4 Truffle's object model

Interpreters often need to support guest-language objects (structures, class instances, etc.). An object is logically a collection of named properties. Accessing these properties should be efficient, and it should be easy for the compiler to optimize the objects—for example, using scalar replacement (Section 2.3.2).

Truffle offers two object models: a *dynamic* model for objects whose properties (and their types) can change during execution [46], and a newer *static* model for objects with a fixed set of properties [7]. This discussion focuses on the static object model since it is the one used by TastyTruffle. The static object model is suitable for languages where objects have a statically known set of properties, such as C, Java, or Scala.

Example: Receipt items

Consider modeling items on a receipt. Each `Item` has an `id` and a `price`. Figure 2.30 demonstrates how to model these items using the static object model. The interpreter first declares a `StaticProperty` for each property; this happens during run time, usually when parsing a guest language definition (for example, a `struct`). It passes these properties to a `StaticShape` builder. The properties are assigned a type and can be marked `final`.

When the `StaticShape` is built, Truffle synthesizes and loads a new JVM class with a field to store each property. Each field is stored at a specific offset in the class's object layout; Truffle updates each `StaticProperty` with its corresponding offset. Truffle also synthesizes a factory to construct these objects. Interpreters use the factory to `create` instances.

Since the storage class is synthesized at run time, an interpreter cannot access its fields using regular field syntax (e.g., `myItem.price`). Instead, it can use the `StaticProperty` instances to perform accesses. The relationship between a generated class and its static properties is depicted in Figure 2.31. A `StaticProperty` points to a specific offset in the generated class instance. Its accessor methods read from/write to the data at that offset using Java's `Unsafe` API. These accesses are fast and incur minimal overhead.

```
1  StaticProperty id = new DefaultStaticProperty("id");
2  StaticProperty price = new DefaultStaticProperty("price");
3
4  StaticShape.Builder builder = StaticShape.newBuilder(...);
5  builder.property(id, int.class, /*final = */ true);
6  builder.property(price, double.class, /*final = */ false);
7  StaticShape itemShape = builder.build(...);
8
9  Object myItem = itemShape.getFactory().create();
10 id.setInt(myItem, 42);
11 price.getDouble(myItem);
```

Figure 2.30: Modeling an `Item` with the static object model.

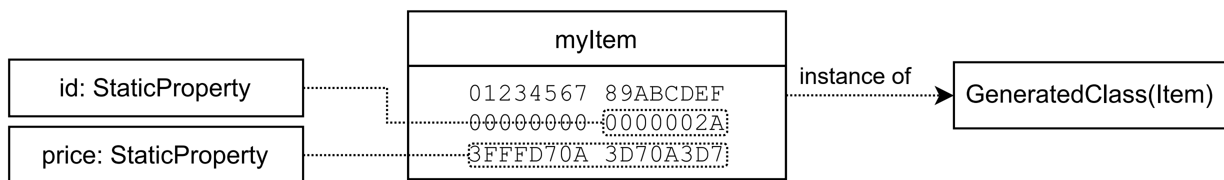


Figure 2.31: The relation between the generated `Item` class and its static properties.

Shape inheritance

`StaticShape`s can be built from a parent shape. This ability is convenient for implementing inheritance—the resultant shape contains all of its parent's properties. Figure 2.32 demonstrates how `Item` can be extended to model a `SaleItem` with a `discount` field. When building `saleItemShape`, the parent `itemShape` is supplied to the builder, and the class it synthesizes is a subclass of the generated `Item` class.

Parent properties have the same offsets in child objects, so a `StaticProperty` can be reused for shapes that inherit it. In this example, the `id` and `price` properties still point to the correct data in `mySaleItem` (Figure 2.33).

Interactions with the compiler

Guest language objects work well with the compiler because of the design of the static object model. Some of the benefits are a direct consequence of modeling guest objects with synthesized JVM classes: the compiler can optimize around guest objects in much the same way it optimizes around regular Java objects.

Constant offsets: Properties always have the same offset even when they are inherited. This guarantee is important for compiled performance: if a field access is compiled, it can use a single offset regardless of the receiver's type (e.g., whether the receiver is a parent or child object).

Scalar replacement: When a static object does not escape its allocating context, it can be decomposed into the fields that it comprises (Section 2.3.2). The compiler understands the `StaticProperty` accessors and can replace them with the appropriate local accesses.

```
1  StaticProperty discount = new DefaultStaticProperty("discount");
2
3  StaticShape.Builder builder = StaticShape.newBuilder(...);
4  builder.property(discount, double.class, false);
5  StaticShape saleItemShape = builder.build(itemShape);
6
7  Object mySaleItem = saleItemShape.getFactory().create();
8  discount.setDouble(mySaleItem, 0.5d);
9  id.setInt(mySaleItem, 43);
```

Figure 2.32: Modeling a `SaleItem` with the static object model.

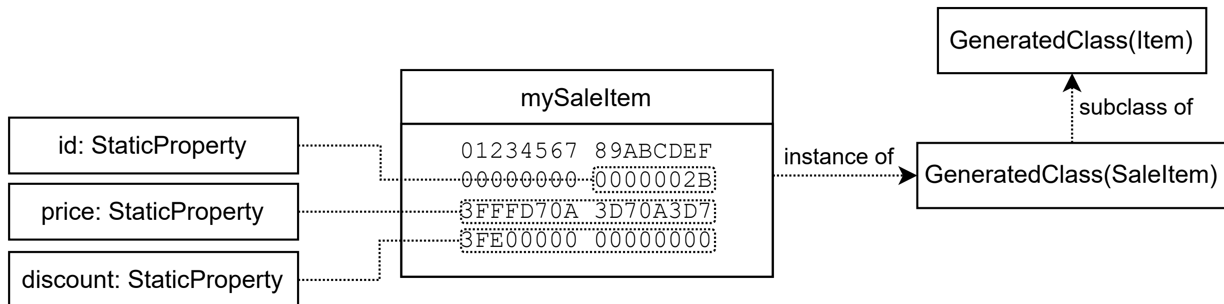


Figure 2.33: The relation between the generated `SaleItem` class and its static properties.

Final properties: When a property is marked final, partial evaluation (Section 2.4.2) can treat it as constant.

Chapter 3

TastyTruffle: A Truffle interpreter for Scala

TastyTruffle is an interpreter built using the Truffle framework [48]. It is a research project (rather than a full-fledged implementation) that implements a subset of the Scala language. In particular, TastyTruffle supports: primitives, singleton objects, and classes with single inheritance; if-statements, while loops, and early returns; and both direct and indirect method dispatch.

This chapter provides a brief overview of how TastyTruffle interprets Scala code. Whereas [50] describes the interpreter internals in great depth, this chapter presents only the high-level details necessary to understand the subsequent chapters. It discusses the design of TastyTruffle's abstract syntax tree (AST), the way it models definitions, the run-time data representation, and method dispatch. The discussion of reified types—a key aspect of TastyTruffle—is deferred until Chapter 4.

While interpreter performance is important for an industry-grade implementation, the primary goal of TastyTruffle is to generate efficient compiled code. Thus, it uses indirections and abstractions that are sometimes inefficient to interpret but become much simpler after compilation—in particular, after partial evaluation (Section 2.4.2). The discussion throughout this chapter emphasizes how such abstractions get simplified in compiled code.

3.1 The TastyTruffle AST

TastyTruffle's AST hierarchy defines a variety of nodes to implement expressions (e.g., `ReadLocal`, `Constant`, `Call`) and control flow (e.g., `IfStmt`, `WhileLoop`). This hierarchy includes intrinsic nodes to support arithmetic, comparisons, and arrays, among other operations. Most nodes are self-explanatory, so the AST is introduced with a few examples.

The basic unit of execution in the interpreter (i.e., the Truffle `RootNode`) is a `Method`. When a `Method` is called, it copies arguments into its frame and executes its body. The body is an AST that evaluates to a value, using the frame to resolve local variables.

A simple example is the Scala `addOne` method depicted in Figure 3.1. This method reads its integer parameter `x` and adds the constant value `1` to it. In the AST, the body

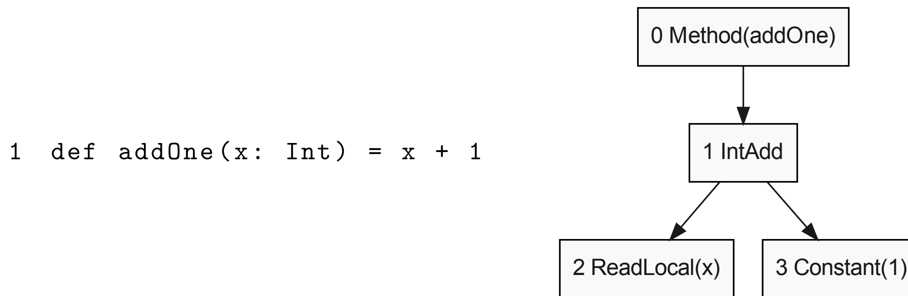


Figure 3.1: Source code and TastyTruffle AST for an addOne method.

of addOne is an IntAdd node that has children for its two operands. The first operand, ReadLocal(x), reads the value of x from the frame. The second operand, Constant(1), simply returns the value 1. The IntAdd node evaluates both children and adds their results together.

Another example is presented in Figure 3.2. This example contains an if-statement and a method call. In the AST, the body of factorial is an If node with subtrees for the condition, then case, and else case. When the condition is true, i.e., x == 0, it executes the then case, which returns the simple constant 1. In the else case, it performs a recursive call using the Call(factorial) node. This node calls the factorial method on the object computed by its receiver subtree, passing its arguments along. In this case, the receiver is the singleton Factorial object that has been stored in a local named Factorial$. The only argument to this call is x - 1. The result of the call is multiplied by x and returned.

Figure 3.3 presents a third example. The Counter class uses a count field to count the number of times increment is called. In Scala, field accesses are proxied through accessor methods that read from or write to the field[1]. TastyTruffle synthesizes these accessors during parsing, and then uses CallFieldRead and CallFieldWrite nodes to invoke the accessors. Like regular Calls, these nodes have a receiver subtree to compute the object containing the field.

The body of increment is a Block that executes a list of statements sequentially. The first statement computes count + 1, reading count using the CallFieldRead node, and then assigns it to the newCount local. The second statement writes newCount back into the count field using a CallFieldWrite. Finally, the third statement reads and returns the value of newCount.

3.2 Definitions

Scala code contains different kinds of definitions. TastyTruffle creates Java objects during parsing to model definitions in the input Scala program. There are five kinds of definitions in TastyTruffle:

[1] Private fields do not use accessor methods and are instead accessed directly (Section 3.3.3).

```scala
1  object Factorial {
2    def factorial(x: Int): Int = {
3      if (x == 0) 1 else x * factorial(x - 1)
4    }
5  }
```

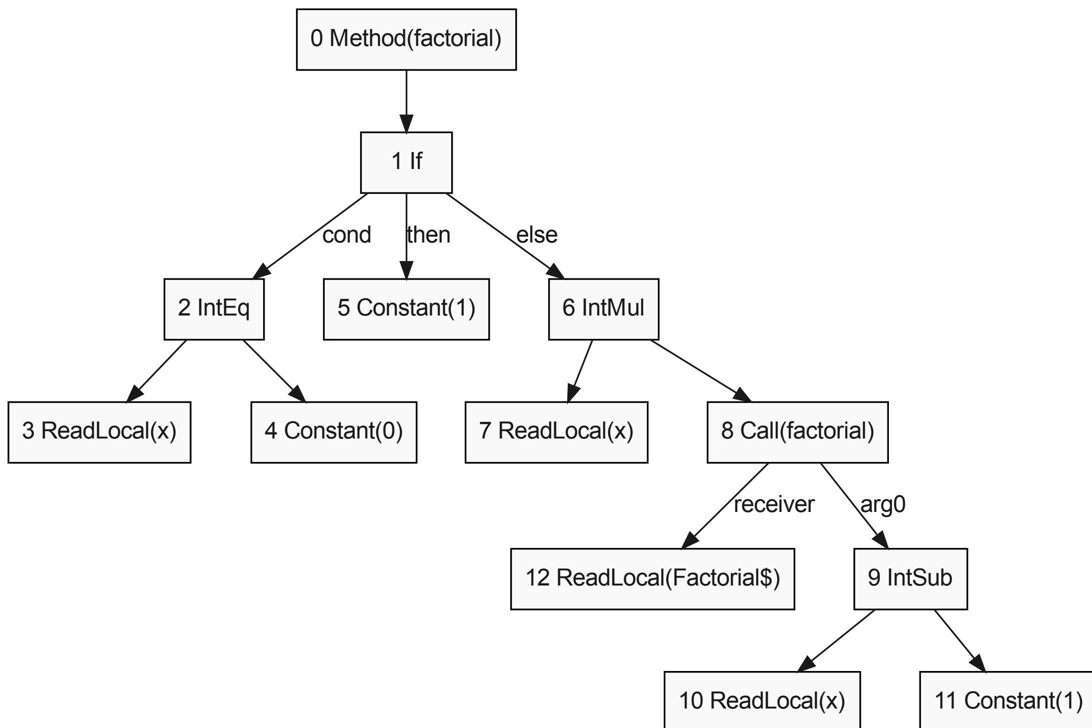


Figure 3.2: Source code and TastyTruffle AST for a `factorial` method.

```
1  class Counter {
2    var count = 0
3    def increment(): Int = {
4      val newCount = count + 1
5      count = newCount
6      newCount
7    }
8  }
```

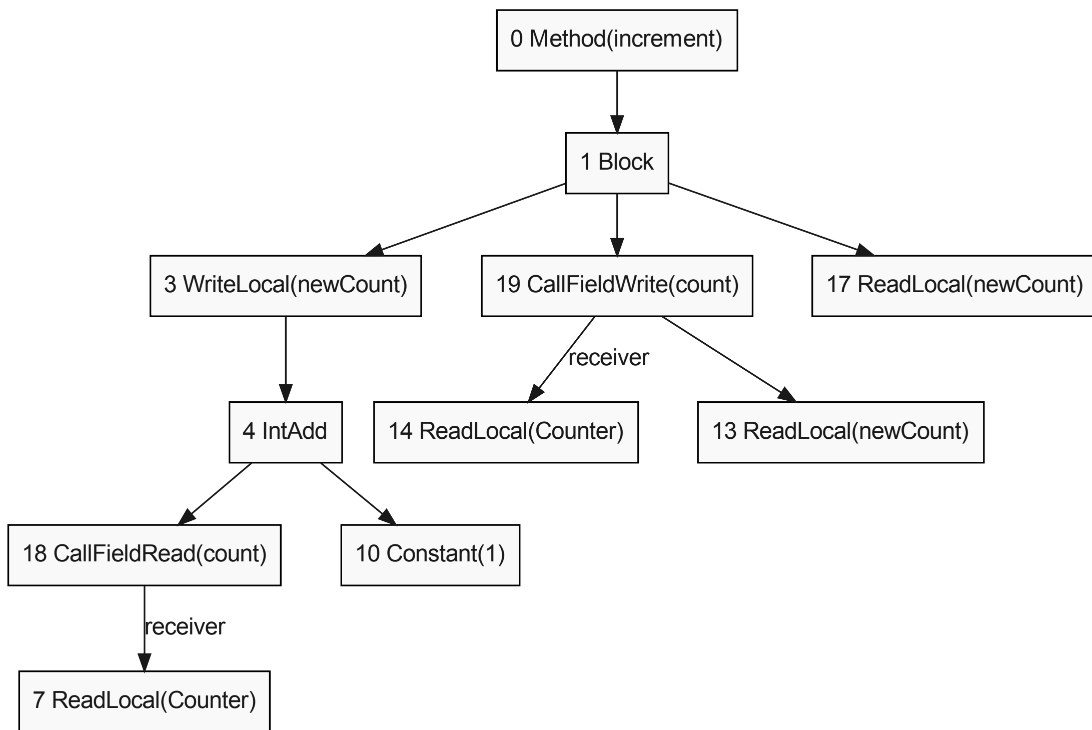


Figure 3.3: Source code for `Counter` and TastyTruffle AST for its `increment` method.

1. `Local` models a local variable.
2. `Field` models a field.
3. `Method` models a method.
4. `Shape` models a class.
5. `Singleton` models a singleton object.

The AST uses these definitions to execute the program. For example, a `ReadLocal` node contains a `Local` definition that it uses to perform read operations.

3.2.1 Referencing definitions in TastyTruffle

When Scala code references definitions, the interpreter must connect each reference in a program to its *referent* (the actual definition) in order to execute the program.

Local references can be resolved at parse-time. When a method body is parsed, all of its locals are known, and actual `Local` instances can be stored in the AST.

In other cases, TastyTruffle may not have loaded/parsed the referent yet, or the referent may not be known until run time, so the parser stores a *symbolic reference* in the AST. This reference can be used at run time to resolve the actual definition. TastyTruffle uses two kinds of symbolic references:

Symbols: For `Field`s, `Shape`s, and `Singleton`s, the referent is statically determined from the textual name and the context of the reference. For example, `new C` refers to the `Shape` for a class `C` imported in the current scope, and a field access `f` refers to the `Field` named `f` defined in the receiver's `Shape`. In such cases, the parser models the reference with a `Symbol`.[2]

Each `Symbol` contains a textual name (e.g., `"C"`). When a definition is nested within another definition, its `Symbol` is qualified by a parent `Symbol`. For example, if `C` is declared inside a package `p`, `C`'s `Symbol` would have `p`'s `Symbol` as its parent, and its fully-qualified name would be `"p.C"`. Similarly, the symbol of a field `f` defined inside `C` would have `C`'s `Symbol` as a parent; its fully-qualified name would be `"p.C.f"`. `Symbol`s are interned to enable efficient hashing and comparison.

`Shape`s and `Singleton`s are stored in global tables, so a node can resolve these definitions at run time by indexing into the appropriate table with a `Symbol`. `Field`s are not stored in a global table, but on each `Shape`. A node resolves a `Field` by first obtaining the `Shape` using the symbol's parent and then using the symbol itself to look up the `Field` from the `Shape` (Section 3.3.2).

Since Scala does not support redefinition of classes, objects, or fields, resolving these definitions is an idempotent operation. Nodes resolve definitions during their first invocation and then cache the results for future invocations. The definitions are marked `@CompilationFinal` so that partial evaluation can treat them as constants.

[2] TastyTruffle's `Symbols` closely mirror the symbols used in TASTy.

Signatures: Scala methods calls are usually virtual. Virtual method calls do not reference specific definitions because the call target depends on the run-time type of the receiver. For example, a call `foo.bar(42)` references *some* method `bar` with a single `Int` parameter, but the concrete implementation depends on the run-time type of `foo`. `Symbols` are inapt for method references because they reference specific definitions.

Instead, TastyTruffle uses a `Signature`, which describes a call target using both its textual name and the types of its parameters.[3] The call to `foo.bar(42)`, for instance, has the signature with name `"bar"` and parameter type `Int`—written as `bar(Int)` for brevity. Parameters are included in signatures in order to support overloading: `bar(Int)` and `bar(Int, Int)` describe different methods altogether.

TastyTruffle's method call nodes use a `Signature` to dispatch to the appropriate `Method` (Section 3.4).

3.3 Data representation in TastyTruffle

An important design consideration is how to represent program data in TastyTruffle. Scala's type system is similar to Java's, so many Scala values can be represented in Truffle with minimal friction. Primitive values (`Int`, `Double`, etc.) are represented by their corresponding Java primitives. Representing primitives this way avoids the overhead of boxing. Arrays of primitives (`Array[Int]`, `Array[Double]`, etc.) are also represented using their corresponding Java arrays. Arrays of reference types are represented using `Object[]`. What remains are class instances (i.e., objects), which contain fields and can have methods invoked on them.

The set of data representations is defined by a `Representation` enum, presented in Figure 3.4. TastyTruffle nodes store a `Representation` when their implementation depends on the representation.

3.3.1 Objects

TastyTruffle uses Truffle's static object model (see Section 2.4.4) to implement Scala objects. The parser creates a `StaticShape` for each Scala class, mapping each field to a `Field` (a subclass of `StaticProperty`) that gets registered on the `StaticShape`. Each `Field` stores the `Representation` that is used for the field (Figure 3.5). Truffle uses these `Fields` to synthesize a JVM class with storage for each field. Each Scala object is an instance of this synthetic class, which is a subclass of a base class, `ClassInstance`.

Consider the `Rectangle` class in Figure 3.6. The `Rectangle` class defines two `Int` fields, `length` and `width`. In the interpreter (during parsing), two `Fields` are created and used to build a `StaticShape`. Truffle generates a class `GeneratedClass(Rectangle)` to represent `Rectangle` instances[4].

[3] The parameter types are also symbolic references.

[4] The actual class name is mangled, but it is presented this way for clarity.

```
1  enum Representation {
2    OBJECT,
3    BOOL,
4    INT,
5    LONG,
6    DOUBLE,
7    ...
8    OBJECT_ARRAY,
9    BOOL_ARRAY,
10   INT_ARRAY,
11   LONG_ARRAY,
12   DOUBLE_ARRAY,
13   ...
14 }
```

Figure 3.4: Source code for the `Representation` enum.

```
1  class Field extends StaticProperty {
2    Symbol name;
3    Representation repr;
4  }
```

Figure 3.5: Data definition for the `Field` class.

The interpreter uses the generated class to represent `Rectangle` objects. It can use the `Field`s to access the fields of the objects. For example, the `Field` object named `length` can be used to read from/write to the `length` field of a `Rectangle` object. The field layout in the generated class is fixed, so each `Field` is assigned a fixed offset. This offset is used to access the field of a given object instance.

The static object model supports inheritance, which TastyTruffle uses to implement single inheritance.[5] For example, in Figure 3.7, `TranslucentRectangle` extends `Rectangle` with an `opacity` field. The class generated for `TranslucentRectangle` extends the generated class for `Rectangle`. It inherits all of `Rectangle`'s fields at the same offsets, and the `opacity` field appears after the inherited fields in the layout.

[5] TastyTruffle does not yet support traits.

```
1  class Rectangle(val length: Int, val width: Int) { ... }
2  myRect = new Rectangle(6, 7)
```

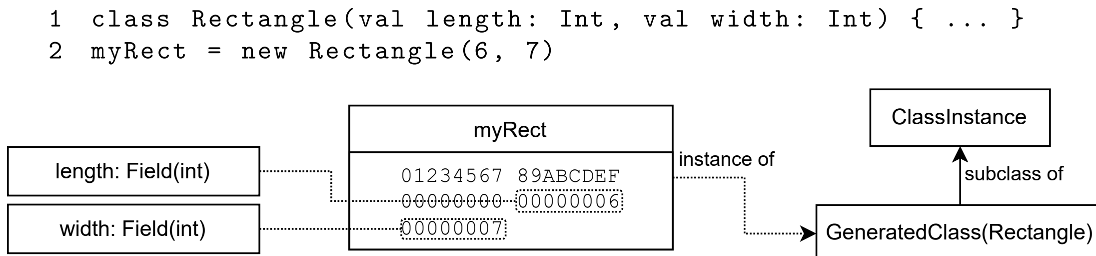


Figure 3.6: A `Rectangle` class and its run-time representation in TastyTruffle.

```
1 class TranslucentRectangle(length: Int, width: Int, val opacity:
      Float) extends Rectangle(length, width) { ... }
2 myTRect = new TranslucentRectangle(2, 21, 0.5)
```

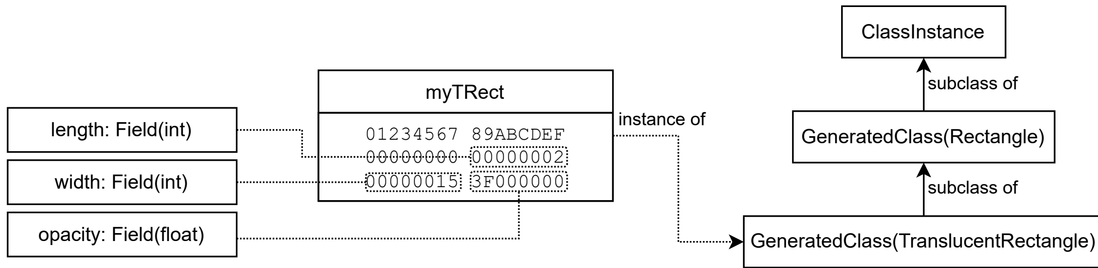


Figure 3.7: A `TranslucentRectangle` class and its run-time representation in TastyTruffle.

```
1 class Shape {
2   Map<Symbol, Field> fields;
3   Map<MethodSignature, Method> methods;
4   Map<MethodSignature, Symbol> vtable;
5   Symbol parent;
6   StaticShape staticShape;
7 }
```

Figure 3.8: Data definition for the `Shape` class.

3.3.2 Shapes

TastyTruffle uses its own `Shape` abstraction to represent all of the information about a Scala class (Figure 3.8). A `Shape` has tables for its declared `fields` and `methods`, a virtual method table (`vtable`), a symbolic reference to its `parent`, and a `staticShape` to create object instances (Section 3.3.1).

`Shape` properties can be computed during parsing. The fields and methods declared inside the class are used to contruct the `fields` and `methods` tables. To construct the `vtable`, the parser uses Scala compiler APIs to determine a class's declared and inherited methods, and then creates a table mapping each method `Signature` to the `Symbol` of its implementing class. A symbolic reference to the `parent` class can also be obtained from the AST. To simplify the class loading process, the parent `Shape` is lazily resolved during interpretation. The `staticShape`, which itself depends on the parent's `staticShape`, is also computed lazily.

Every object has a `shape` field referencing its `Shape`. The interpreter uses an object's `shape` to execute type-specific operations, such as looking up fields, performing method dispatch, and checking the type of the object.

The `Shape` for `Rectangle` is presented in Figure 3.9. Its `fields` list stores the `length` and `width` fields. Its `methods` table contains entries for the `area` and `draw` methods it declares, as well as the initializer method (`<init>`) that gets invoked when a `Rectangle` is created. The `vtable` maps a method signature to the shape that implements each virtual

```
1  class Rectangle(val length: Int, val width: Int) {
2    // ...
3    def area: Int = length * width
4    def draw: Unit = ...
5  }
6
7  class TranslucentRectangle(length: Int, width: Int, val opacity:
       Float) extends Rectangle(length, width) {
8    // ...
9    override def draw: Unit = ...
10 }
```

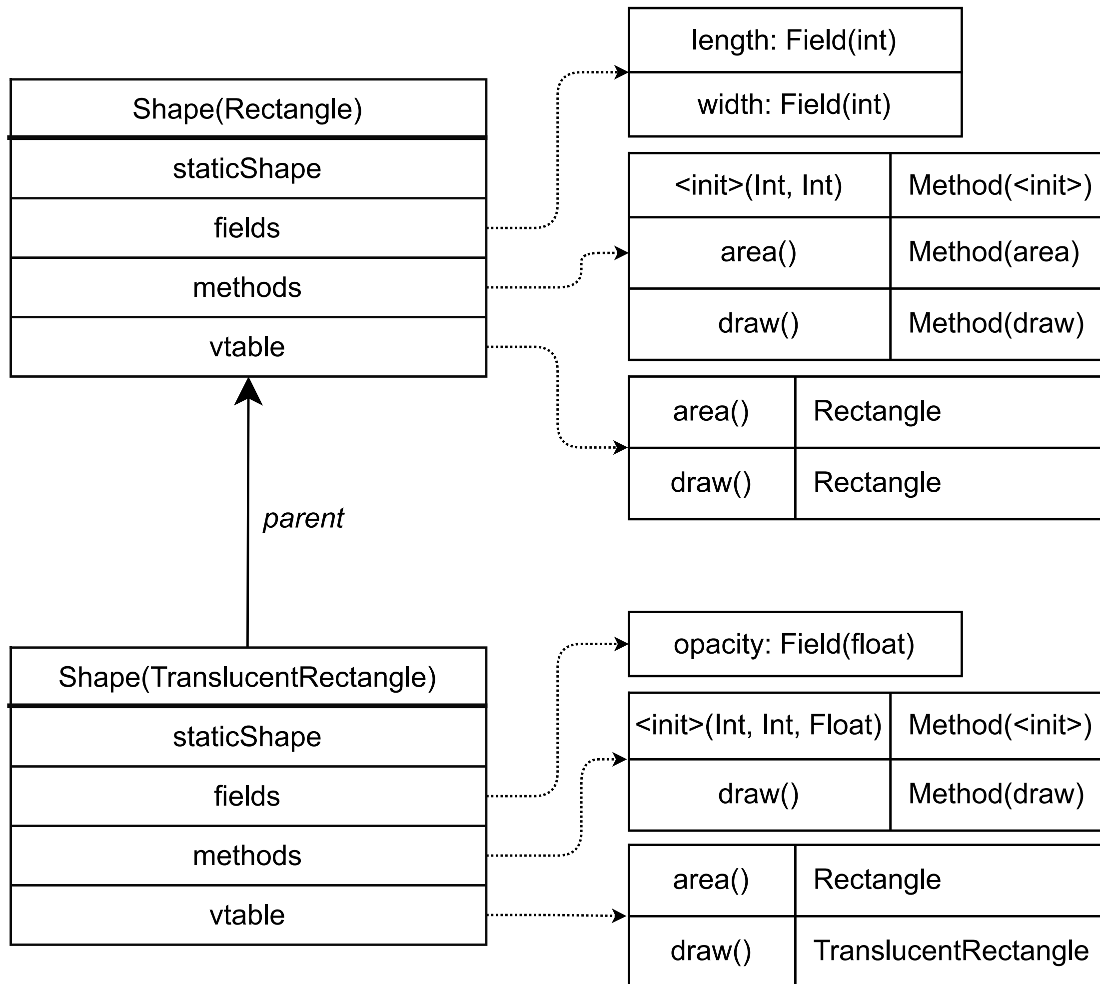


Figure 3.9: Method declarations and `Shapes` for the rectangle classes.

method; in this case, both `area` and `draw` are implemented by `Rectangle`.

A `Shape` can have a `parent`. In Figure 3.9, the `TranslucentRectangle` class extends `Rectangle`. Its `fields` and `methods` store the fields and methods newly declared in `TranslucentRectangle`. The `vtable` reflects the fact that `draw` has been overridden by `TranslucentRectangle` and that the `area` implementation has not changed. The `parent` pointer allows TastyTruffle to search the parent `Shape` for fields and methods not declared by the current `Shape`.

3.3.3 Data representation and the AST

Since values in TastyTruffle are represented in different ways, the AST must know a value's representation in order to interpret it correctly. For example, to read a local from the frame, TastyTruffle needs to know whether the value is an `int`, a `double`, an object, or something else. In such situations, nodes store a `Representation` (Figure 3.4) and use it in their implementation.

Locals

TastyTruffle's `ReadLocal` and `WriteLocal` nodes implement local accesses. Each of these nodes contains a `Local` object that stores the frame index and `Representation` of a local. Local accessor nodes use `Locals` to access local variables at the correct frame index and with the correct representation.

A simplified version of `ReadLocal` is depicted in Figure 3.10a. To read a local from the frame, `ReadLocal` first switches over the local's representation to determine how to access it. Once the representation is known, `ReadLocal` reads the local from the frame using the appropriate type and the index specified by the `local`.

This design interacts well with partial evaluation. Since `local` is annotated with `@CompilationFinal`, the compiler treats it as a constant and can replace the calls to `local.getRepresentation()` and `local.getIndex()` with the constant values of those fields. Then, seeing a constant argument to the `switch`, the compiler can remove the branching, replacing it with only the matching branch. For example, suppose `ReadLocal` is partially evaluated and its `Local` has index `2` and type `INT`. The code after partial evaluation is significantly simpler (Figure 3.10b).[6] The branching and indirect calls in the original code are eliminated, and what remains is a single frame read `frame.getInt(2)`. When Graal performs scalar replacement (Section 2.3.2), the frame read further simplifies to a regular local variable read.

The source code for `WriteLocal` is depicted in Figure 3.11. The implementation is similar to `ReadLocal`, with a couple of differences worth noting:

- First, `WriteLocal` evaluates a `rhsNode` to compute a value to assign to the local. Since `rhsNode` returns an `Object`, the value may need to be unboxed depending on the local representation. For example, when the representation is `INT`, `rhsNode` returns an `int`, which it boxes up as an `Integer`; then, `(int) rhs` casts the `rhs` back

[6]Partial evaluation happens during bytecode parsing (Section 2.4.2); source code is presented for the sake of illustration.

```
 1  class ReadLocal extends Node {
 2    @CompilationFinal Local local;
 3
 4    Object execute(VirtualFrame frame) {
 5      return switch(local.getRepresentation()) {
 6        case BOOL -> frame.getBoolean(local.getIndex());
 7        case INT -> frame.getInt(local.getIndex());
 8        case LONG -> frame.getLong(local.getIndex());
 9        ... // other primitives
10        default -> frame.getObject(local.getIndex());
11      }
12    }
13  }
```

(a) Source code for a `ReadLocal` node before partial evaluation.

```
 1  class ReadLocal extends Node {
 2    @CompilationFinal Local local;
 3
 4    Object execute(VirtualFrame frame) {
 5      return frame.getInt(2);
 6    }
 7  }
```

(b) Source code for a `ReadLocal` node after partial evaluation (when `local` has type `INT` and index `2`).

Figure 3.10: Source code for a `ReadLocal` node before and after partial evaluation.

```
 1  class WriteLocal extends Node {
 2    @CompilationFinal Local local;
 3    @Child Node rhsNode;
 4
 5    Object execute(VirtualFrame frame) {
 6      Object rhs = rhsNode.execute(frame);
 7      switch(local.getRepresentation()) {
 8        case BOOL -> frame.setBoolean(local.getIndex(), (boolean) rhs);
 9        case INT -> frame.setInt(local.getIndex(), (int) rhs);
10        case LONG -> frame.setLong(local.getIndex(), (long) rhs);
11        ... // other primitives
12        case /* array representation */ ->
13          frame.setObject(
14            local.getIndex(),
15            CompilerDirectives.castExact(rhs, /* array class */)
16          )
17        default -> frame.getObject(local.getIndex());
18      }
19      return UNIT;
20    }
21  }
```

Figure 3.11: Source code for a `WriteLocal` node.

to Integer and extracts the primitive int value. This boxing and unboxing may seem inefficient, but when partial evaluation inlines rhsNode into WriteLocal, Graal can easily detect and remove the box-unbox chain.

- Secondly, WriteLocal has a special case to handle array representations. Unlike primitives, Truffle frames store all reference-type values, including arrays, in slots that are statically of type Object. As a consequence, when an array is stored in a local variable, sometimes Graal cannot infer the actual type of it (e.g., int[]). WriteLocal uses the castExact directive to indicate the precise type of an array local to Graal. Since all locals are written to before they are read, Graal propagates the type information through the program; when a subsequent ReadLocal loads the array from an Object index, Graal already knows the value is an array—ReadLocal does not need to insert its own directive.

Fields

TastyTruffle has ReadField and WriteField nodes to implement direct field accesses.[7] These nodes work analogously to local accessors, except they use Field objects. Each Field has a fixed Representation and points to a fixed offset in the receiver object, so each access compiles to a simple raw memory read/write. If Graal performs scalar replacement on the receiver, field accesses also simplify to local variable accesses.

Arrays

TastyTruffle has several kinds of array nodes that create and operate over arrays. The array nodes need to support each array representation used in TastyTruffle: namely, they should work with each primitive array type (int[], boolean[], and so on) and Object[]. One implementation strategy would be to create new nodes for each representation, but this leads to a lot of duplicated code. Instead, TastyTruffle's array nodes store the Representation of the array's component type and use it to implement each operation.

The ArrayInit node instantiates a new array with a given length (Figure 3.12). It determines what representation to create by switching over the component's Representation. The component type is @CompilationFinal, so partial evaluation can fold away the type switch and Graal can infer a precise array type for the result.

The ArrayApply, ArrayUpdate, and ArrayLength nodes operate on existing arrays. In Java bytecode, primitive arrays are accessed using a unique set of operations, so these nodes must cast arrays to their corresponding types before they can operate on them.[8] The array nodes switch over the component Representation to determine how to cast the array object.

[7] Recall that non-private field accesses are *indirect* accesses proxied through accessor methods (Section 3.1). Private field accesses and the accessor methods themselves are implemented using ReadField and WriteField.

[8] For example, array application (as in array[i]) is a different Java operation for int[], double[], boolean[] and so on. Reference-type arrays use a different set of operations that is shared across reference types.

```
 1 class ArrayInit extends Node {
 2   @Child Node length;
 3   @CompilationFinal Representation componentRepr;
 4
 5   Object execute(VirtualFrame frame) {
 6     int size = (int) length.execute(frame);
 7     return switch(componentRepr) {
 8       case BOOL -> new boolean[size];
 9       case INT -> new int[size];
10       case LONG -> new long[size];
11       case DOUBLE -> new double[size];
12       ...
13       default -> new Object[size];
14     };
15   }
16 }
```

Figure 3.12: Source code for an `ArrayInit` node.

To illustrate, consider the implementation of `ArrayApply` in Figure 3.13. When the `componentRepr` is `INT`, `ArrayApply` knows the array is an `int[]`, so it casts it to `int[]` before reading index `i`. Again, since the component type is `@CompilationFinal`, partial evaluation can fold away the switch. Graal can usually infer the array's type from context, so the cast can often be removed during compilation.[9]

3.4 Method dispatch

To execute a `Call`, TastyTruffle must perform method dispatch to identify the call target (i.e., the concrete method that should be invoked). TastyTruffle supports two kinds of calls, which use different approaches for method dispatch.

3.4.1 Direct calls

Sometimes, a call target is statically known. These calls are *direct calls*. For example, in Figure 3.14, a call to `foo` can only ever resolve to the implementation in `C` because `foo` is marked `final`. TastyTruffle uses direct calls when the call target cannot be overridden, which is the case for `final` methods, `private` methods, and constructors.

The code to implement a `DirectCall` is depicted in Figure 3.15. The node has a symbolic reference to the class that implements the method (`owner`) and a method signature (`sig`). To dispatch the method, `getCallTarget` obtains the `Shape` of the `owner` class from a global registry. Then, it uses the signature `sig` to look up the appropriate `Method` from the shape's table of declared `methods`.

[9] An array can either flow into the current method or be created inside the method (possibly by a callee). In the former case, `WriteLocal` casts parameters to precise array types; in the latter case, the array allocation is usually inlined into the graph.

```
1  class ArrayApply extends Node {
2    @Child Node self;
3    @Child Node index;
4    @CompilationFinal Representation componentRepr;
5
6    @Override
7    Object execute(VirtualFrame frame) {
8      Object array = self.execute(frame);
9      int i = (int) index.execute(frame);
10
11     return switch(componentRepr) {
12       case BOOL -> ((boolean[]) array)[i];
13       case INT -> ((int[]) array)[i];
14       case LONG -> ((long[]) array)[i];
15       case DOUBLE -> ((double[]) array)[i];
16       ...
17       default -> ((Object[]) array)[i];
18     };
19   }
20 }
```

Figure 3.13: Source code for an `ArrayApply` node.

```
1  class C {
2    final def foo(x: Int) = ???
3  }
4
5  val x: C = ???
6  x.foo(42)
```

Figure 3.14: Example code with a direct call.

```
1  @NodeChild("receiver")
2  class DirectCall extends Node {
3    @Children Node[] args;
4    @CompilationFinal Symbol owner;
5    @CompilationFinal Signature sig;
6
7    @Specialization
8    Object execute(VirtualFrame frame, ClassInstance receiver,
9        @Cached("getCallTarget()") Method callTarget) {
10     Object[] args = ... // evaluate arguments
11     return callTarget.call(args);
12   }
13
14   Method getCallTarget() {
15     Shape shape = Globals.lookup(owner);
16     return shape.methods.get(sig);
17   }
18 }
```

Figure 3.15: Source code for a `DirectCall` node.

The direct call dispatch requires two hash-table lookups, which can be expensive, but since the call target never changes, it only needs to be performed once. During its first invocation, a `DirectCall` computes and caches the call target, and then subsequent invocations reuse it. Since the cached target is a partial evaluation constant, the method can be inlined directly into the call site during compilation.

Because most methods can be overridden, direct calls occur infrequently in TastyTruffle outside of the three cases mentioned. There are techniques that could potentially increase the number of direct call sites, like class hierarchy analysis [18] or Truffle's speculative `Assumption`s. However, in TastyTruffle, monomorphic indirect calls enjoy similar compiled performance to direct calls, so such a transformation would likely have a negligible effect on performance.

3.4.2 Indirect calls

In Scala, most methods defined by a class can be redefined by subclasses. For example, in Figure 3.16, class `D` overrides method `bar`. As a result of overriding, dispatching the call to `bar` is not as simple as a direct call dispatch. The call could dispatch to `C`'s implementation, `D`'s implementation, or some other arbitrary subclass's implementation, depending on the concrete type of the receiver. These kinds of calls are *indirect calls*.

The code to implement an `IndirectCall` is presented in Figure 3.17. For a `DirectCall`, the implementing class is statically known, but for an `indirectCall`, `getCallTarget` must determine the implementing class using the receiver's virtual method table (`vtable`) stored on its `Shape`. Once the implementing class is known, `DirectCall` looks up its `Shape` and then looks up the implementation from the `Shape`'s `methods` table.

Indirect calls are difficult to optimize. In the worst case, an indirect call must perform a virtual dispatch for each invocation, since the receiver type can change each time. However,

```
1  class C {
2    def bar() = ???
3    def baz() = ???
4  }
5  class D extends C {
6    override def bar() = ???
7  }
8
9  val x: C = ???
10 x.bar()
```

Figure 3.16: Example code with an indirect call.

```
1  @NodeChild("receiver")
2  class IndirectCall extends Node {
3    @CompilationFinal Signature sig;
4
5    @Specialization(guards = "receiver.shape==cachedShape", limit="5")
6    Object executeCached(VirtualFrame frame, ClassInstance receiver,
7        @Cached("receiver.shape") Shape cachedShape,
8        @Cached("getCallTarget(receiver)") Method callTarget) {
9      Object[] args = ... // evaluate arguments
10     return callTarget.call(args);
11   }
12
13   @Specialization(replaces = "executeCached")
14   Object executeUncached(VirtualFrame frame, ClassInstance receiver){
15     Object[] args = ... // evaluate arguments
16     Method callTarget = getCallTarget(receiver);
17     return callTarget.call(args);
18   }
19
20   Method getCallTarget(ClassInstance receiver) {
21     Symbol implementer = receiver.shape.vtable.lookup(sig);
22     Shape implShape = Globals.lookup(implementer);
23     return implShape.methods.get(sig);
24   }
25 }
```

Figure 3.17: Source code for an `IndirectCall` node.

```
1 shape = receiver.shape
2 if (shape == C) {
3   // body of C.bar
4 } else if (shape == D) {
5   // body of D.bar
6 } else {
7   // deoptimize
8 }
```

(a) Polymorphic case: the cache contains `C` and `D`.

```
1 shape = receiver.shape
2 if (shape == C) {
3   // body of C.bar
4 } else {
5   // deoptimize
6 }
```

(b) Monomorphic case: the cache contains only `C`.

Figure 3.18: Pseudocode generated by PE for the indirect call to `bar` in Figure 3.16.

in realistic workloads, the vast majority of call sites observe just a few receiver types [6], and polymorphic inline caches [25] can be used to improve performance.

`IndirectCall` uses Truffle's caching to implement a polymorphic inline cache. On first invocation, it starts in the `executeCached` specialization. The node performs the virtual dispatch, caching the receiver's shape (`cachedShape`) and the result (`callTarget`), and then calls the method. On subsequent invocations, the specialization guard is checked: the same `callTarget` is reused as long as `receiver.shape` is equal to `cachedShape`. If the `receiver` has a different `Shape`, `IndirectCall` performs the same process to add new cache entries. The `execute` method generated by Truffle's DSL processor loops through the cache entries to search for a match. Eventually, if the cache limit is exceeded, the `executeUncached` replaces `executeCached`.[10] This specialization performs the virtual dispatch each time.

Indirect calls pose a problem for JIT compilation: when Graal cannot determine the call target, it cannot inline the method and optimize across the call boundary. The call is effectively a black box: it returns an arbitrary value, and can modify arbitrary program state (e.g., globals, fields, etc.). The opaqueness of an indirect call can prevent the compiler from reasoning about the state of the program after the call.

However, if an indirect call is not megamorphic (i.e., it encounters few actual receiver types), using inline caches mitigates the compilation problem. Graal unrolls the cache loop into a series of if-statements that check whether the receiver `Shape` matches each `cachedShape`. Inside these if-statement bodies, the call targets *are* statically known, and Graal can inline them into the body of `IndirectCall`.

Consider the call to `bar` in Figure 3.16. If the call does not hit the cache limit, PE unrolls the cache lookup loop and inlines the cached implementations of `bar`. The pseudocode for the call after PE is depicted in Figure 3.18.[11] The figure demonstrates two cases: a polymorphic call site and a monomorphic call site.

Suppose the `IndirectCall` observes two receiver types during execution, `C` and `D`. The call site is polymorphic and its inline cache has two entries. The code generated by PE (Figure 3.18a) performs a pointer comparison between the receiver's `shape` and each

[10] TastyTruffle uses a cache limit of 5, which is used by [6] to describe a *megamorphic* call site.

[11] Again, recall that partial evaluation happens during bytecode parsing (Section 2.4.2); source code is presented for illustrative purposes.

cached shape until it finds a match. If it finds a match, the code executes the body of the inlined call target. Otherwise, the cache lookup fails, and the code deoptimizes back to the interpreter to handle the call (before possibly recompiling later).

Now, suppose the `IndirectCall` only observes a single receiver type `C`. The call site is monomorphic and its inline cache has a single entry. The code generated by PE (Figure 3.18b) is similar to the polymorphic case. However, with just a single cache entry, the indirect call is effectively a direct call guarded by a single pointer comparison. Graal can optimize the call just as effectively as it can optimize direct calls.

Chapter 4

Using reified types in TastyTruffle

TASTy was chosen as the interpretation target for TastyTruffle because, unlike JVM bytecode, TASTy has complete type information, which can be used to *reify* types. This chapter discusses TastyTruffle's implementation of reified types, and in turn describes how reified types are used to implement parametric polymorphism. These details are also described in [50] as one monolithic extension. This chapter contributes an intermediate implementation that supports parametric polymorphism *without* specialization.

4.1 Reified types

The high-level idea behind reifying types is to pass around type information into generic contexts. When generic code understands the types of values that flow into generic contexts, it can support heterogeneous representations for those values.

Consider the example code in Figure 4.1. When compiled to JVM bytecode, the type parameter `T` of `swap` is erased. The bytecode does not "remember" that `swap` is generic, and it has no knowledge about the actual types of its arguments. Both `array` and `value` are given type `Object`, and the implementation of `swap` operates on the values in a uniform way, independently of the actual values provided to it. Forcing values into a uniform representation when they flow into generic contexts—for example, by boxing primitives—introduces run-time overhead and makes it harder for a compiler to produce efficient code.

If types are reified, generic code can avoid these performance penalties, instead supporting heterogeneous representations of generic values. In a reified scheme, `swap` knows about its type parameter `T`, and `swap`'s caller knows that it invokes the generic `swap` method with `T` being `Int`. The caller can pass the run-time representation of the `Int` type as an argument to the call, and then `swap` can inspect the value of `T` to determine that `array` is an `Array[Int]`, `value` is actually an `Int`, and that the array operations in the method body operate over integer arrays. If it also performed generic calls, `swap` could propagate the value of `T` to its callees. Having precise knowledge about the types used in generic contexts enables heterogeneous representations for generic values in TastyTruffle.

```
1 def swap[T](array: Array[T], i: Int, value: T): T = {
2   val oldValue: T = array(i)
3   array(i) = value
4   oldValue
5 }
6
7 val array = new Array[Int](1,2,3,4)
8 swap(array, 0, 5)
```

Figure 4.1: A generic `swap` method that updates an array and returns the previous value stored.

4.1.1 TypeNodes

To reify types in TastyTruffle, the AST must be augmented with additional metadata. This metadata takes the form of a `TypeNode`. A `TypeNode` is a Truffle tree that can be executed to obtain a `Shape`, TastyTruffle's run-time representation of a type (Section 3.3.2). This `Shape` is the "type" passed around the interpreter; `TypeNode`s are *not* passed around, but merely augment the TastyTruffle AST so that it can *compute* the types being passed around.

`TypeNode`s model:

- Type parameters (e.g., `T`) with `MethodTypeParam` and `ClassTypeParam`.
- Primitive types (e.g., `Int`) with `IntType`, `DoubleType`, etc.
- Class types (e.g., `Foo`) with `NamedType`.
- Generic type applications (e.g., `List[Int]`) with `AppliedType`.

The first three kinds of nodes are simple symbolic references to a type. The `AppliedType` node represents a generic type with concrete type arguments. Each of its type arguments is itself a `TypeNode`.

Type reification is possible in TastyTruffle because every expression and definition in TASTy is annotated with type information. This information is modeled by TASTy's `TypeRepr` hierarchy. The `TypeRepr` hierarchy includes a variety of classes to model Scala's rich type system, but in practice only a couple of these classes are relevant for the simple kinds of programs supported by TastyTruffle: `TypeRef`, which is a symbolic reference to a type (e.g., the name of a class), and `AppliedType`, which models a generic type with a set of supplied type arguments.

During parsing, TastyTruffle can obtain `TypeRepr`s for definitions and expressions from the TASTy representation. For any TastyTruffle node that needs a `TypeNode`, the parser takes the corresponding `TypeRepr` and converts it to a `TypeNode` using the algorithm in Figure 4.2.

As mentioned earlier, `TypeNode`s are Truffle trees that can be executed to produce a `Shape`, TastyTruffle's run-time representation of a type. The `TypeNode`s for primitives and classes are simple: they look up a `Shape` by name from the `Globals` registry.

```
def parseType(tpe: TypeRepr): TypeNode = tpe match {
  case TypeRef if tpe.isTypeParameter => {
    if (tpe.isMethodTypeParameter)
      new MethodTypeParam(...)
    else
      new ClassTypeParam(...)
  }
  case TypeRef if tpe.isPrimitiveType => {
    if (tpe.isInt)
      new IntType
    else if (tpe.isDouble)
      new DoubleType
    ...
  }
  case TypeRef => new NamedType(symbolOf(tpe))
  case AppliedType(template, typeArgs) =>
    new tastytruffle.AppliedType(
      symbolOf(template),
      typeArgs.map(parseType)
    )
}
```

Figure 4.2: Scala pseudocode to construct a TastyTruffle `TypeNode` from a TASTy `TypeRepr`.

The implementations for type parameter and type application nodes are discussed in the subsequent sections.

Reifying types in TastyTruffle means extra information must be computed and passed around the interpreter, but partial evaluation can reduce the performance impact. If a `TypeNode` evaluates to a fixed `Shape`, PE can detect that its result is a compilation constant and elide code that executes it. Further, when code performs run-time type tests over a constant `TypeNode`, PE can remove the type tests and fold away any unreachable branches.

4.2 Generic methods

A generic method behaves independently of the specific types of (some of) the values it operates on. It uses type parameters to model the types of its generic values. In a reified system, callers can compute and pass concrete values for these type parameters into generic methods.

In TastyTruffle's calling convention, type arguments are passed just like regular arguments, appearing after the receiver but before the regular arguments in the argument list. When the parser encounters a generic call, it creates a `Call` node with additional `TypeNode`s to compute the type arguments. In the callee, the type parameters get copied into its frame; then, the callee can read its type parameters using a `MethodTypeParam` node (Figure 4.3). This node loads a type parameter directly from the frame just like a `ReadLocal` node.

```
1 class MethodTypeParam extends TypeNode {
2   final Local typeParam;
3
4   Object execute(VirtualFrame frame) {
5     return frame.getObjectStatic(typeParam.getIndex());
6   }
7 }
```

Figure 4.3: Source code for a `MethodTypeParam` node.

Consider the `foo` method in Figure 4.4a. This method reads from its generic array parameter `x`, stores the generic result in a local `y`, and then calls another generic method `bar` with `y`. In TastyTruffle, a `Call` to `foo` takes three arguments: the `GenericMethods` receiver, a type argument for `T`, and a value for `x`. The AST for `foo` is depicted in Figure 4.4b.

Whereas generic methods in an erasure scheme use a single data representation regardless of the type arguments they are invoked with, reified types enable the implementation to vary its representation. For example, when `T` is `Int`, TastyTruffle can represent a value of type `T` with a primitive `int` rather than `Object`; similarly, it can represent an `Array[T]` with `int[]` rather than `Object[]`. This representation is more precise and avoids boxing. Generic methods in TastyTruffle use type parameters to dynamically support these different data representations.

4.2.1 Generic locals

When a local variable is generic, its representation changes dynamically; consequently, the way that it should be accessed from the frame changes dynamically. TastyTruffle defines `GenericReadLocal` and `GenericWriteLocal` nodes that use reified types to implement generic accesses.

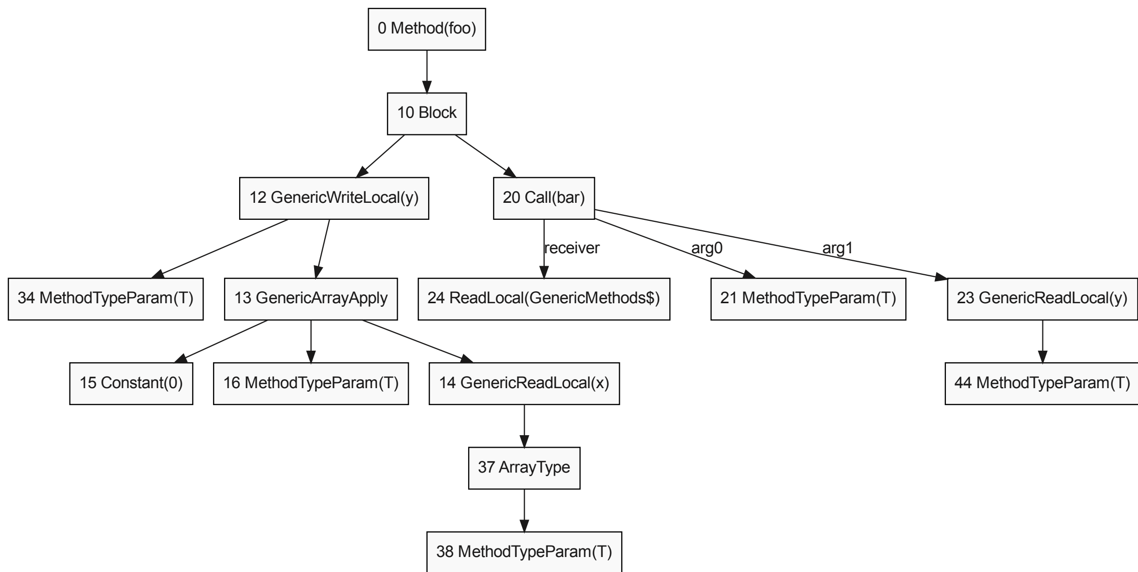
Consider the locals in the `foo` method (Figure 4.4a). Both `x` and `y` are generic over `T`. When `T` is `Int`, `y` contains an `int` and is stored in the primitives section of the frame; conversely, when `T` is a reference type, `y` is stored in the `Object` section. In either case, `x` is an array stored in the `Object` section, but the reified types are still useful for hinting at the underlying type with compiler directives (as in Section 3.3.3).

The implementation of `GenericReadLocal` is depicted in Figure 4.5. It defines a separate specialization for each possible representation. Whereas a `ReadLocal` is annotated with a static `Representation` (Figure 3.10a), a `GenericReadLocal` is annotated with a `TypeNode` to dynamically compute the representation. The generated `execute` method evaluates the `TypeNode` to compute the local's `Shape`. Then, it compares the shape against each specialization's type guard to select an appropriate specialization.

Consider the `GenericReadLocal(y)` node in Figure 4.4b. The local `y` is of type `T`. When `T` is `Int`, the `TypeNode` for `y` evaluates to an `IntShape`, so the generated code selects and executes the `readInt` specialization. If `T` is a reference type, none of the primitive specializations apply, and `ReadLocalNode` uses the `@Fallback` specialization, which accesses the local from the `Object` section of the frame.

```
1  object GenericMethods {
2    def foo[T](x: Array[T]): T = {
3      val y: T = x(0)
4      bar[T](y)
5    }
6    def bar[U](x: U): U = ???
7  }
```

(a) Generic methods `foo` and its callee `bar`.



(b) Generic AST for `foo`.

Figure 4.4: Source code and AST for a generic method `foo`.

```
1  @NodeChild("typeNode")
2  class GenericReadLocal extends Node {
3    final int index;
4
5    @Specialization
6    Object readInt(VirtualFrame frame, IntShape shape) {
7      return frame.getIntStatic(index + INT.ordinal());
8    }
9
10   @Specialization
11   Object readDouble(VirtualFrame frame, DoubleShape shape) {
12     return frame.getDoubleStatic(index + DOUBLE.ordinal());
13   }
14
15   ...
16
17   @Fallback
18   Object readObject(VirtualFrame frame, Shape shape) {
19     return frame.getObjectStatic(index + OBJECT.ordinal());
20   }
21 }
```

Figure 4.5: Source code for a `GenericReadLocal` node.

Dynamically changing the representation of locals has subtle consequences for compiled code. In generic nodes like `GenericReadLocal`, multiple specializations can be active at once depending on which types flow into the node at run time. In the compiled code, if the `TypeNode` is not a compilation constant, each active specialization appears in a different branch of control flow. For example, after partial evaluation, a `GenericWriteLocal` that has seen `int` and `Object` representations during interpretation compiles to something like Figure 4.6. Control flow branches based on the type of `shape` and then merges afterwards.

These kinds of control flow splits and merges are challenging for Graal to optimize. Consider the type of the value in the frame after the store. Since the types differ on the two branches, the value can be either an `int` or a reference type. The compiler models the value using a ϕ with an imprecise type, and subsequent accesses in the compiled code may require boxing or dynamic type checks.

TastyTruffle works around this problem by allocating a unique index in the frame for each possible representation of a local. For example, a given index `n` may be used for reference types, `n+1` for `int`s, `n+2` for `double`s, and so on. In Figure 4.5, each specialization accesses a different index by adding the `Representation`'s enum ordinal to the base index; `GenericWriteLocal` works the same way.

Using multiple indices for generic locals can cause the interpreter to allocate more frame space than necessary. For example, if a generic local is only ever represented with `int`, the extra frame space is wasted memory. However, compiled code does not have this problem. When partial evaluation processes `GenericWriteLocal` and `GenericReadLocal` nodes, it only includes the specializations (i.e., representations) that the nodes used during interpretation. In the resulting code, if only a subset of the indices are actually used, Graal

```
 1  Object execute(VirtualFrame frame) {
 2    Object value = /* evaluate rhs */;
 3    Object shape = /* evaluate typeNode */;
 4    if (shape instanceof IntShape) {
 5      // store value as an int
 6    } else if (shape instanceof Shape) {
 7      // store value as an Object
 8    } else {
 9      // deopt
10    }
11    return UNIT;
12  }
```

Figure 4.6: Pseudocode for a `GenericWriteLocal` node's `execute` method after partial evaluation.

can elide storage for the unused indices.

4.2.2 Generic array accesses

Since TastyTruffle has multiple underlying representations for arrays, code that operates on generic arrays must dynamically support different representations at run time. For example, the `foo` method in Figure 4.4a takes a parameter of type `Array[T]`. When `T` is `Int`, the array is represented with an `int[]`, but when `T` is something else, the array uses a different representation.

TastyTruffle defines generic variants of the array accessor nodes described in Section 3.3.3. The implementation of `GenericArrayApply` is given in Figure 4.7. Like with generic local accessors, generic array nodes execute a `TypeNode` and use its result to determine the representation of the given array.

4.2.3 Propagating type information

Type parameters can also transitively flow into other call sites. When `foo` calls `bar` in Figure 4.4a, it evaluates its type parameter `T` and passes it as an argument. In this way, `bar` can also use the value of `T` to dynamically change its own behaviour.

4.3 Generic classes

Scala classes can also be generic. Like with generic methods, a generic class has type parameters that can indicate the types of values used in its methods. A generic class can also define generic fields whose types are described by its class type parameters. Since fields are stored in class instances, the representation of a generic class instance itself depends on the values of its type parameters. The code that operates on a generic class instance needs to account for the heterogeneous ways it may be represented.

```
1  @NodeChild("self")
2  @NodeChild("index")
3  @NodeChild("typeNode")
4  class GenericArrayApplyNode extends Node {
5
6    @Specialization
7    Object applyInt(Object array, int i, IntShape shape) {
8      return ((int[]) array)[i];
9    }
10
11   @Specialization
12   Object applyDouble(Object array, int i, DoubleShape shape) {
13     return ((double[]) array)[i];
14   }
15
16   ...
17
18   @Fallback
19   Object applyObject(Object array, int i, Shape shape) {
20     return ((Object[]) array)[i];
21   }
22 }
```

Figure 4.7: Source code for a `GenericArrayApply` node.

```
1  class Box[T](initial: T) {
2    private var value: T = initial
3    def get: T = value
4    def set(x: T): Unit = value = x
5  }
```

Figure 4.8: Generic `Box` class.

Consider the `Box` class in Figure 4.8. `Box[T]` wraps a generic value of type `T`, storing it in a field and manipulating it with `get` and `set` methods. In a reified system like TastyTruffle, the concrete value of `T` determines the field representation: when `T` is `Int`, the value should be stored in an `int` field, when it is `Double`, the value should be stored in a `double` field, and so on. The concrete value of `T` thus affects the object layout of a `Box[T]`.

This section discusses TastyTruffle's implementation of generic classes. Generic classes build on the implementation from Section 4.2. They use the same AST nodes to support generic locals and arrays—however, now the `TypeNode`s stored on those nodes may contain `ClassTypeParam`s as well.

4.3.1 Modeling applied generic classes

To support generic classes, the interpreter needs a way to model a generic class applied to concrete type arguments, or *applied generic class*, for short. This section discusses how

TastyTruffle models applied generic classes. The mechanism by which a generic class is actually applied to concrete types is discussed in the following section.

Recall that TastyTruffle uses `Shape`s to model non-generic classes in the guest program; the definition of a `Shape` from Chapter 3 is reproduced in Figure 4.9. The `StaticShape` synthesized by Truffle defines the layout of each class instance, and the `fields` mapping contains the `Field`s used to access fields from a class instance.

```
1 class Shape {
2   Map<Symbol, Field> fields;
3   Map<MethodSignature, Method> methods;
4   Map<MethodSignature, Symbol> vtable;
5   Symbol parent;
6   StaticShape staticShape;
7 }
```

Figure 4.9: Data definition for the `Shape` class (reproduced).

TastyTruffle models an applied generic class with a `GenericShape` (Figure 4.10). The `GenericShape` class extends the base `Shape`, introducing an additional `typeArgMap` field to map type parameters to the concrete type arguments applied to the class.

```
1 class GenericShape extends Shape {
2   Map<Symbol, Shape> typeArgMap;
3 }
```

Figure 4.10: Data definition for the `GenericShape` class.

For a given generic class, each unique set of type arguments corresponds to a unique `GenericShape`. Using different shapes for different applications of a generic class is necessary because the type arguments can impose different object layouts. Each `GenericShape` has its own layout with `fields` that use an appropriate representation; for example, the `GenericShape` created to model the `Box[Int]` class has a `value` field that uses an `int` representation.

Like with method type parameters, class type parameters can be directly referenced in method bodies. The TastyTruffle parser creates `ClassTypeParam`s when parsing `TypeNode`s. When a generic node (e.g., a generic local accessor) executes, it can evaluate `ClassTypeParam`s to determine the concrete type arguments used when applying the generic class. For example, in Figure 4.8, `set` computes the value of `T` to determine how the argument for `x` should be stored in the frame.

The implementation of `ClassTypeParam` is given in Figure 4.11. The node obtains the receiver's `GenericShape` and looks up the type argument from its `typeArgMap`. Since `Map` operations cannot be partially evaluated, `ClassTypeParam` caches the computed result as a compiler constant. Further, since a generic class method observes a different `GenericShape` for each unique application of the class, the cache is polymorphic over the receiver's shape. The node falls back on uncached access when it observes too many different `GenericShape`s.

```
1  @NodeChild("receiver")
2  class ClassTypeParam extends TypeNode {
3    final Symbol symbol;
4
5    @Specialization(guards = "receiver.shape==cachedShape", limit="5")
6    Shape resolveCached(
7        ClassInstance receiver,
8        @Cached("receiver.shape") Shape cachedShape,
9        @Cached("getTypeArgument(cachedShape)") Shape typeArgument) {
10     return typeArgument;
11   }
12
13   @Specialization(replaces = "resolveCached")
14   Shape resolveUncached(ClassInstance receiver) {
15     return getTypeArgument(receiver.shape);
16   }
17
18   Shape getTypeArgument(Shape shape) {
19     GenericShape gShape = (GenericShape) shape;
20     return gShape.typeArgMap.get(symbol);
21   }
22 }
```

Figure 4.11: Source code for a `ClassTypeParam` node.

```
1  class GenericShapeTemplate {
2    FieldTemplate[] fields;
3    Map<MethodSignature, Method> methods;
4    Map<MethodSignature, Symbol> vtable;
5    Symbol parent;
6    Symbol[] typeParams;
7  }
```

Figure 4.12: Data definition for the `GenericShapeTemplate` class.

4.3.2 Applying generic classes to type arguments

The interpreter needs a mechanism to apply a generic class to concrete type arguments. TastyTruffle uses the `GenericShapeTemplate` class for this purpose (Figure 4.12). The TastyTruffle parser creates a `GenericShapeTemplate` when it encounters a generic class. The parser stores shape templates in a global table indexed by `Symbol`s.

A shape template is conceptually a `Shape` waiting to have concrete type arguments supplied to it. It retains all of the metadata necessary to construct a `Shape` at a later time. Namely, since a `Field`'s representation is not known until a generic class is applied to concrete types, `GenericShapeTemplate` stores `FieldTemplate`s (Figure 4.13). A field template models a field's type with a `TypeNode`. If the field is generic, the type node contains `ClassTypeParam`s.

```
1 class FieldTemplate {
2   Symbol name;
3   TypeNode type;
4 }
```

Figure 4.13: Data definition for the `FieldTemplate` class.

Specializing `TypeNodes`

A critical aspect of the generic application algorithm is TastyTruffle's technique for transforming `TypeNodes` in the AST. Truffle ASTs support the visitor pattern, whereby a function is applied to each node in the AST in a top-down traversal. Visitors can be used to transform ASTs.

TastyTruffle defines a `TypeNodeSpecializer` that "fills in" type parameters in a `TypeNode` tree with concrete types (Figure 4.14a).[1] The specializer searches the AST for `ClassTypeParams`. For each `ClassTypeParam` it finds, it replaces the node with a `ConstType` node containing the concrete type of the parameter.

The `TypeNodeSpecializer` transforms generic `TypeNodes` into nodes with concrete types that have fixed representations. Figure 4.14b provides some examples. Importantly, since `TypeNodeSpecializer` visits subtrees, it can fill in type parameters nested in `AppliedType` nodes.

The application algorithm

To apply a generic class to type arguments, `GenericShapeTemplate` follows the algorithm in Figure 4.15:

1. First, create a `typeArgMap` mapping each type parameter to its concrete type argument.

2. Then, construct the `Fields`. Use the `TypeNodeSpecializer` to fill in type parameters in each field template's `TypeNode.` The resultant `TypeNode` is not generic and can be executed to obtain a `Shape`. Use the `Shape`'s `repr` to create a `Field` with a specialized representation.

3. Use the `Fields` to synthesize a `StaticShape`.

4. Create a new `GenericShape` with the computed fields, shape, and type argument mapping. Use the same methods and parent information.[2]

The resulting `GenericShape`, like any other `Shape`, can be used to instantiate instances of the generic class. Any generic fields on the class are specialized and use a precise representation.

[1] Some of TastyTruffle's implementation is written in Scala, including `TypeNodeSpecializer`. Scala's pattern matching is especially useful for tree visitors.

[2] Methods are shared among different generic applications. Inheritance with generic classes is discussed in a later section.

```scala
1  class TypeNodeSpecializer(typeArgMap: Map[Symbol, Shape])
2      extends NodeTransformer {
3    def visit(node: Node): Node = node match {
4      case param: ClassTypeParam =>
5        val typeArg = typeArgMap(param.symbol)
6        new ConstType(typeArg)
7      case _ => node
8    }
9  }
```

(a) Source code (Scala).

```java
1  Symbol T = symbolOf("T");
2  TypeNodeSpecializer specializer = new TypeNodeSpecializer(
3    Map.of(T, IntShape)
4  );
5
6  TypeNode simple = new ClassTypeParam(T);
7  specializer.visit(simple); //-> ConstType(IntShape)
8
9  TypeNode box = new AppliedType(symbolOf("Box"),
10     new ClassTypeParam(T));
11 specializer.visit(box); //-> AppliedType("Box", ConstType(
12                         //      IntShape))
13
14 Symbol U = symbolOf("U");
15 specializer = new TypeNodeSpecializer(
16   Map.of(
17     T, IntShape,
18     U, DoubleShape
19   )
20 );
21
22 TypeNode pair = new AppliedType(
23   symbolOf("Pair"), new ClassTypeParam(T), new ClassTypeParam(U)
24 );
25 specializer.visit(pair); //-> AppliedType("Pair",ConstType(IntShape),
26                          //       ConstType(DoubleShape))
```

(b) Example usage (Java).

Figure 4.14: Source code and example usage for the TypeNodeSpecializer.

```
1  class GenericShapeTemplate {
2    ...
3    GenericShape apply(Shape[] typeArgs) {
4      // Create a mapping from type parameter to argument
5      Map<Symbol, Shape> typeArgMap = new HashMap<>();
6      for (int i = 0; i < typeArgs.length; i++) {
7        typeArgMap.put(this.typeParams[i], typeArgs[i]);
8      }
9
10     // Build the fields
11     TypeNodeSpecializer specializer = new TypeNodeSpecializer(
12         typeArgMap);
13     Map<Symbol, Field> specializedFields = new HashMap<>();
14     for (FieldTemplate template : this.fields) {
15       // Specialize the field's type
16       TypeNode concreteType = specializer.visit(template.type);
17
18       // Compute the field's Shape
19       Shape concreteShape = concreteType.execute(null);
20
21       // Create a new Field
22       specializedFields.put(
23         field.name,
24         new Field(field.name, concreteShape.repr)
25       );
26     }
27
28     // Create a new StaticShape
29     StaticShape.Builder builder = StaticShape.newBuilder(...);
30     for (Field field : specializedFields.values()) {
31       builder.property(field, classOf(field.getRepresentation()));
32     }
33     StaticShape staticShape = builder.build(...);
34
35     return new GenericShape(
36       specializedFields,
37       this.methods,
38       this.vtable,
39       this.parent,
40       staticShape,
41       typeArgMap
42     )
43   }
44 }
```

Figure 4.15: Source code for the `GenericShapeTemplate`'s `apply` method.

Caching GenericShapes

When a GenericShapeTemplate is applied to type arguments, if the type arguments have been seen before, it should reuse the previously-computed GenericShape. Not only is it unnecessary to recompute the shape each time, but two different generic class instances with the same type arguments should have the same shape. For example, two Box[Int] instances created at different times during execution should both use the same GenericShape. Shape uniqueness is especially important for performance, since TastyTruffle's inline caches (e.g., on an IndirectCall node) use Shapes as cache keys.

Thus, GenericShapeTemplate caches the GenericShapes it creates. The caching code, which is omitted from Figure 4.15 for simplicity, is presented in Figure 4.16. For each application, it caches the list of concrete type arguments in keys and the resulting GenericShape in values. The lookup method compares the input type arguments against the cached arguments in keys; if there is a match, it returns the corresponding cached GenericShape. Otherwise, GenericShapeTemplate creates a new GenericShape and adds the result to the cache.

Partial evaluation

The caching code is written carefully to be amenable to partial evaluation. The keys and values arrays are both @CompilationFinal, including the array elements inside the arrays, so when PE encounters the @ExplodeLoop annotations in lookup and shapesMatch, it can unroll all of the loops into comparisons against constant Shapes.

As an example, consider a generic HashMap[K,V] class. Suppose that during interpretation, HashMap[K,V] is applied to [Int,Int], [Int,Double], [Double,Int], and [Double,Double]. Its GenericShapeTemplate has an entry for each set of arguments in its cache. Figure 4.17 illustrates how a call to lookup is incrementally transformed by partial evaluation, starting with the original code in Figure 4.17a.[3] First, since keys is a PE-constant, PE can unroll all four iterations of the loop (Figure 4.17b). Then, the shapesMatch method can be inlined (Figure 4.17c). Inlining adds new loops to the code, but each key[i] array is PE-constant, so PE can unroll each loop into two shape comparisons. The code in Figure 4.17d depicts the final result after constants have been inlined (and the control flow rewritten for presentation).

Generic application is complex: among other things, it synthesizes new Java classes, mutates @CompilationFinal cache fields, and modifies Truffle trees. This code cannot be included during partial evaluation. Thus, in the apply method, code that creates new GenericShapes is dominated by a deoptimization directive, so PE does not include it during compilation. If this code path is reached, compiled code deoptimizes to the interpreter.

4.3.3 Modeling generic classes in the AST

When Scala code instantiates a generic class, the TastyTruffle parser models the type in the AST using an AppliedType node. The AppliedType node evaluates to a GenericShape

[3] Partial evaluation happens during bytecode parsing, but source code is useful for illustration. The exact order in which PE transforms the code is not necessarily accurate, but the end result is the same.

```
1  class GenericShapeTemplate {
2    ...
3    @CompilationFinal(dimensions=2) Shape[][] keys;
4    @CompilationFinal(dimensions=1) GenericShape[] values;
5
6    GenericShape apply(Shape[] typeArgs) {
7      GenericShape cached = lookup(typeArgs);
8      if (cached != null) return cached;
9
10     // deopt; generic application cannot happen in compiled code
11     CompilerDirectives.transferToInterpreterAndInvalidate();
12
13     // regular application code
14
15     put(typeArgs, result);
16     return result;
17   }
18
19   @ExplodeLoop
20   GenericShape lookup(Shape[] typeArgs) {
21     for (int i = 0; i < keys.length; i++) {
22       if (shapesMatch(keys[i], typeArgs)) return values[i];
23     }
24     return null;
25   }
26
27   @ExplodeLoop
28   boolean shapesMatch(Shape[] expected, Shape[] actual) {
29     if (expected.length != actual.length) return false;
30     for (int i = 0; i < expected.length; i++) {
31       if (expected[i] != actual[i]) return false;
32     }
33     return true;
34   }
35
36   void put(Shape[] typeArgs, GenericShape result) {
37     int index = /* next slot, resizing if necessary */;
38     keys[index] = typeArgs;
39     values[index] = result;
40   }
41 }
```

Figure 4.16: Source code for the `GenericShapeTemplate`'s caching mechanism.

```
1 for (int i = 0; i < keys.length; i++) {
2   if (shapesMatch(keys[i], typeArgs)) return values[i];
3 }
4 return null;
```

(a) Before partial evaluation.

```
1 if (shapesMatch(keys[0], typeArgs)) return values[0];
2 if (shapesMatch(keys[1], typeArgs)) return values[1];
3 if (shapesMatch(keys[2], typeArgs)) return values[2];
4 if (shapesMatch(keys[3], typeArgs)) return values[3];
5 return null;
```

(b) After the outer loop is unrolled.

```
1  INT_INT_CHECK: for (int i = 0; i < keys[0].length; i++) {
2    if (keys[0][i] != typeArgs[i]) goto INT_DOUBLE_CHECK;
3  }
4  return values[0];
5
6  INT_DOUBLE_CHECK: for (int i = 0; i < keys[1].length; i++) {
7    if (keys[1][i] != typeArgs[i]) goto DOUBLE_INT_CHECK;
8  }
9  return values[1];
10
11 DOUBLE_INT_CHECK: for (int i = 0; i < keys[2].length; i++) {
12   if (keys[2][i] != typeArgs[i]) goto DOUBLE_DOUBLE_CHECK;
13 }
14 return values[2];
15
16 DOUBLE_DOUBLE_CHECK: for (int i = 0; i < keys[3].length; i++) {
17   if (keys[3][i] != typeArgs[i]) goto FAILURE;
18 }
19 return values[3];
20
21 FAILURE: return null;
```

(c) After `shapesMatch` is inlined.

```
1 if (IntShape==typeArgs[0] && IntShape==typeArgs[1])
2     return values[0];
3 if (IntShape==typeArgs[0] && DoubleShape==typeArgs[1])
4     return values[1];
5 if (DoubleShape==typeArgs[0] && IntShape==typeArgs[1])
6     return values[2];
7 if (DoubleShape==typeArgs[0] && DoubleShape==typeArgs[1])
8     return values[3];
9 return null;
```

(d) After unrolling the inner loops, inlining constants, and normalizing (assuming `IntShape` and `DoubleShape` are singleton shapes).

Figure 4.17: Pseudocode for `lookup` (from Figure 4.16) after partial evaluation.

```
1  class AppliedType extends TypeNode {
2    final Symbol templateSymbol;
3    @Children TypeNode[] typeArgs;
4
5    @ExplodeLoop
6    Object execute(VirtualFrame frame) {
7      GenericShapeTemplate template = Globals.lookup(templateSymbol);
8
9      Shape[] concreteArgs = new Shape[typeArgs.length];
10     for (int i = 0; i < typeArgs.length; i++) {
11       concreteArgs[i] = typeArgs[i].execute(frame);
12     }
13
14     return template.apply(concreteArgs);
15   }
16 }
```

Figure 4.18: Source code for an `AppliedType` node.

that can be used to create new instances of the applied generic class. The implementation of `AppliedType` is given in Figure 4.18. The node looks up a `GenericShapeTemplate` from the global template table, evaluates its type arguments, and calls `apply` to apply the template to the type arguments. This method returns a `GenericShape` with the given type arguments.

Again, consider a generic `HashMap[K,V]` class. To create a new instance of this class with specific type arguments, a Scala programmer writes `new HashMap[Int,Double](...)`. In TastyTruffle, the generic type application is modeled by an `AppliedType` with a symbolic reference to `HashMap` and a `typeArgs` array containing `IntType` and `DoubleType`. When the `AppliedType` executes, it obtains a `GenericShape` with a specialized representation; this shape can then be used to create new instances.

Partial evaluation

Like the caching mechanism in `GenericShapeTemplate`, `AppliedType` is written carefully for partial evaluation. Its type arguments are `@Children`, which means they have compilation-final semantics. The `@ExplodeLoop` annotation instructs PE to unroll the code evaluating the type arguments. After unrolling, `concreteArgs` is a fixed-size array that can be statically compared against the entries in the `GenericShapeTemplate` cache.

To illustrate, consider again the `HashMap[K,V]` example used in Figure 4.17. Partial evaluation can already simplify the `lookup` call to a series of shape comparisons, as in Figure 4.17d. If `TypeApply` gets partially evaluated, PE has even more context about the type arguments and can further simplify Figure 4.17d.

When the i-th type argument is a PE constant, PE can eliminate any cache entries that have a different i-th key. For example, with the `TypeApply` for `HashMap[Int,T]`, PE removes the two entries that do not have `IntShape` as their first shape (lines 3 and 4). The first two if-statements can also be simplified, since the condition `IntShape == typeArgs[0]` is trivially true.

When all of the type arguments are PE constants, such as `HashMap[Int,Double]`, PE can resolve the exact `GenericShape` from the cache directly, completely removing the overhead of the cache lookup.[4]

4.3.4 Generic fields

Each `GenericShape` generated for a generic class can have a different layout for its `Fields`. These fields may themselves have different representations. The TastyTruffle nodes that access fields must support these different layouts and representations dynamically. TastyTruffle defines `GenericReadField` and `GenericWriteField` nodes to support direct field accesses on generic instances.

When a class is generic, the TastyTruffle parser generates generic accessor nodes for all of its direct field accesses.[5] Even though a generic class's non-generic fields have a fixed representation, they must also use generic accessor nodes because their position in the object layout can change.

The source code for `GenericReadField` is depicted in Figure 4.19. It looks up the `Field` from the `shape` of the receiver object, and then switches over the field's representation to determine how to read the field contents. Since `lookup` uses a `Map`, it is not partially-evaluatable, and so `GenericReadField` caches these lookups. In compiled code, PE can treat a cached `Field` as a compilation constant and simplify the code that switches over its representation. Since `Fields` are different for each `GenericShape`, the cache is polymorphic over the receiver `Shape` (not unlike `ClassTypeParam` or `IndirectCall`).

4.3.5 Method dispatch

Generic classes affect method dispatch in a couple of ways:

Direct calls: Direct calls should be used for call sites with a single statically-known implementation. If a method's defining class is generic, then the implementation may change depending on the concrete type arguments, so direct calls cannot be used.[6] Therefore, TastyTruffle does not create `DirectCall` nodes for methods defined on generic classes, instead using `IndirectCall` nodes.

Indirect calls: Each `Shape` has a `vtable` mapping method signatures to `Symbols` of the `Shapes` that implement each method. An indirect call obtains a `Symbol` from the receiver's `vtable` and then looks up the corresponding `Shape` from a global table. Unfortunately, for methods defined by generic classes, the `Symbol` of the generic class does not correspond to an actual `Shape`, but instead a `GenericShapeTemplate`.

[4] Since TastyTruffle runs in the interpreter before compiling hot methods, it is likely that the `GenericShape` is in the cache. If no cache entry is found, the code deoptimizes to the interpreter where the generic class can be applied to new types.

[5] Recall that in Scala, indirect accesses are proxied through accessor methods. The accessor methods use these nodes in their implementations.

[6] Technically, since different applications of a generic class currently share the same method implementations, direct calls *can* be used, but this changes when specialization is introduced in Chapter 5.

```
1  @NodeChild("receiver")
2  class GenericReadField extends Node {
3    final Symbol selector;
4
5    @Specialization(guards = "instance.shape==cachedShape", limit="5")
6    Object readCached(
7        ClassInstance instance,
8        @Cached("instance.shape") Shape cachedShape,
9        @Cached("cachedShape.lookup(selector)") Field field) {
10     return getValue(instance, field)
11   }
12
13   @Specialization(replaces = "readCached")
14   Object readUncached(ClassInstance instance) {
15     Field field = instance.shape.lookup(selector);
16     return getValue(instance, field);
17   }
18
19   Object getValue(ClassInstance instance, Field field) {
20     return switch(field.getRepresentation()) {
21       case BOOL -> field.getBoolean(instance);
22       case INT -> field.getInt(instance);
23       case LONG -> field.getLong(instance);
24       ... // other primitives
25       default -> field.getObject(instance);
26     }
27   }
28 }
```

Figure 4.19: Source code for a `GenericReadField` node.

```
1  class Vector[T] {
2    def get(i: Int): T
3    def set(i: Int, value: T): Unit
4    def append(value: T): Unit
5  }
6
7  class IntVector extends Vector[Int] {
8    def sort(): Unit
9  }
10
11 class Stack[T] extends Vector[T] {
12   def push(value: T): Unit
13   def pop(): T
14 }
```

Figure 4.20: Example code with generic parent classes.

TastyTruffle solves this problem by not looking up the `Shape` directly. Instead, `IndirectCall` searches the receiver's `Shape` hierarchy for a `Shape` with the same `Symbol` as the one in the `vtable`. This search happens in the slow path when a method is first dispatched; the result is cached for subsequent lookups.

For example, consider the `IntVector` class in Figure 4.20. If `get` is invoked on an `IntVector`, its `vtable` indicates that `get` is defined on `Vector`. `IndirectCall` iterates over the receiver's `Shape` hierarchy to search for a `Shape` with `Vector` as its "generic" symbol. It finds a `GenericShape` for `Vector[Int]` and dispatches to the shape's `get` method.

4.3.6 Generic parent classes

Scala classes can have generic parent classes, like the `IntVector` and `Stack` classes in Figure 4.20. To support generic parent classes in TastyTruffle, the implementation must be modified in a few ways:

Parent references: Prior to generic classes, the `parent` of a `Shape` was modeled with a `Symbol`. However, a symbol cannot precisely model any type arguments supplied to a generic parent class. For example, `IntVector`'s parent symbol is simply `Vector`. Without the concrete type arguments, TastyTruffle cannot determine the parent `Shape`, and thus cannot build the `Shape` of `IntVector` itself.

Instead, parent references should be modeled with `TypeNode`s. The parent `TypeNode` can be evaluated to obtain a precise `Shape` for the parent class.

Generic class application: Sometimes, a generic child supplies type parameters to a generic parent. For example, in the definition of `Stack[T]`, the parent `Vector[T]` obtains its type parameter `T` from `Stack`.

When a generic class is applied to a set of types, as described in Section 4.3.2, the application algorithm should specialize the parent `TypeNode` over the given set

of types. The `TypeNodeSpecializer` visitor can be used to specialize the parent `TypeNode`.

The parent `TypeNode` must be specialized because of a chicken-and-egg problem. Consider what happens when TastyTruffle tries to create an instance of `Stack[Int]`:

1. To create a `Stack[Int]` instance requires a `Shape` for `Stack[Int]`.
2. To build the `Shape` for `Stack[Int]` requires the parent `Shape`.
3. To resolve the parent `Shape` requires executing the `TypeNode` for `Vector[T]`.
4. The `TypeNode` for `Vector[T]` contains a `ClassTypeParam`, which requires a `Stack[T]` instance in order to read the concrete type argument for `T`.

TastyTruffle avoids this circular dependency by specializing the parent `TypeNode`. When `Stack[T]` is applied to `Int`, TastyTruffle specializes the parent `TypeNode` from `Vector[T]` to `Vector[Int]`. The specialized `TypeNode` can then be evaluated directly without a class instance.

Resolving type parameters: `ClassTypeParam`'s implementation reads a type argument from the receiver's `Shape` (Figure 4.11). However, when a generic class can be subclassed, the type argument is not necessarily stored in the receiver's `Shape`. Instead, `ClassTypeParam` should search for the type parameters in the receiver's entire `Shape` hierarchy.

For example, a `Vector[T]` method may take an `IntVector` instance as its receiver. The `Shape` of this receiver is `IntVector`, which does not store the type argument `T`. Any `ClassTypeParams` in the method's AST need to search the `Shape` hierarchy to find the concrete type of `T`.

Chapter 5

Specializing generic code

Chapter 4 describes a scheme to support generic methods and classes in TastyTruffle using reified types. The implementation, as presented, can exhibit inconsistent performance. This chapter explains the performance problem and extends the scheme with specialization to achieve more reliable performance. Unlike [50], which presents a *static* specialization scheme, this chapter shows how specialization can be done *dynamically* without any major transformations to the AST. This chapter also presents the aforementioned static specialization scheme.

5.1 Motivation

The implementation of generic methods and classes presented in Chapter 4 is highly dynamic. During interpretation, a variety of different representations can flow through a generic AST. The AST dynamically changes its behaviour for each representation, but the type profiles and specialization states of the nodes are shared, which can lead to megamorphic code that is poorly optimized by Graal.

Consider the `swap` method in Figure 5.1. Suppose that during interpretation, the program invokes `swap` with `T` being `Double`, `Int`, and a reference type. Each generic node in the AST specializes itself to handle each representation. When Graal compiles `swap`, it generates a control flow split to handle each representation for each generic node. Control flow merges after each generic node. The resulting control flow graph looks something like Figure 5.2. These frequent control flow splits inhibit optimization, since Graal cannot infer a precise state after each merge.

```
1 def swap[T](array: Array[T], i: Int, value: T): T = {
2   val oldValue: T = array(i)
3   array(i) = value
4   oldValue
5 }
```

Figure 5.1: A generic `swap` method that updates an array and returns the previous value stored (reproduced from Figure 4.1).

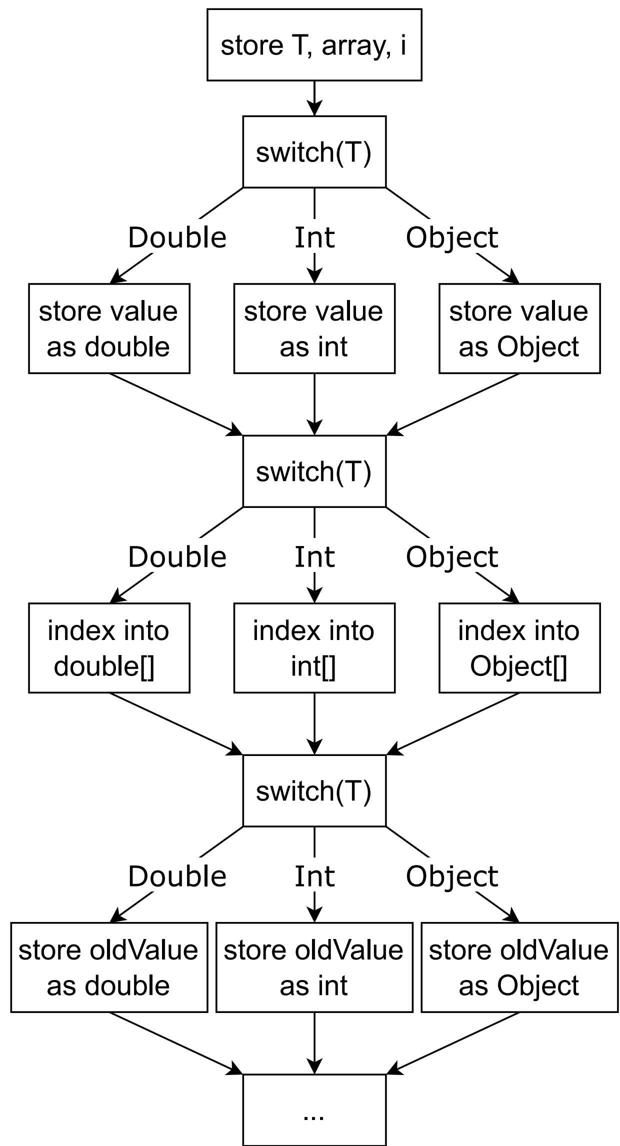


Figure 5.2: Part of the control flow graph for `swap`.

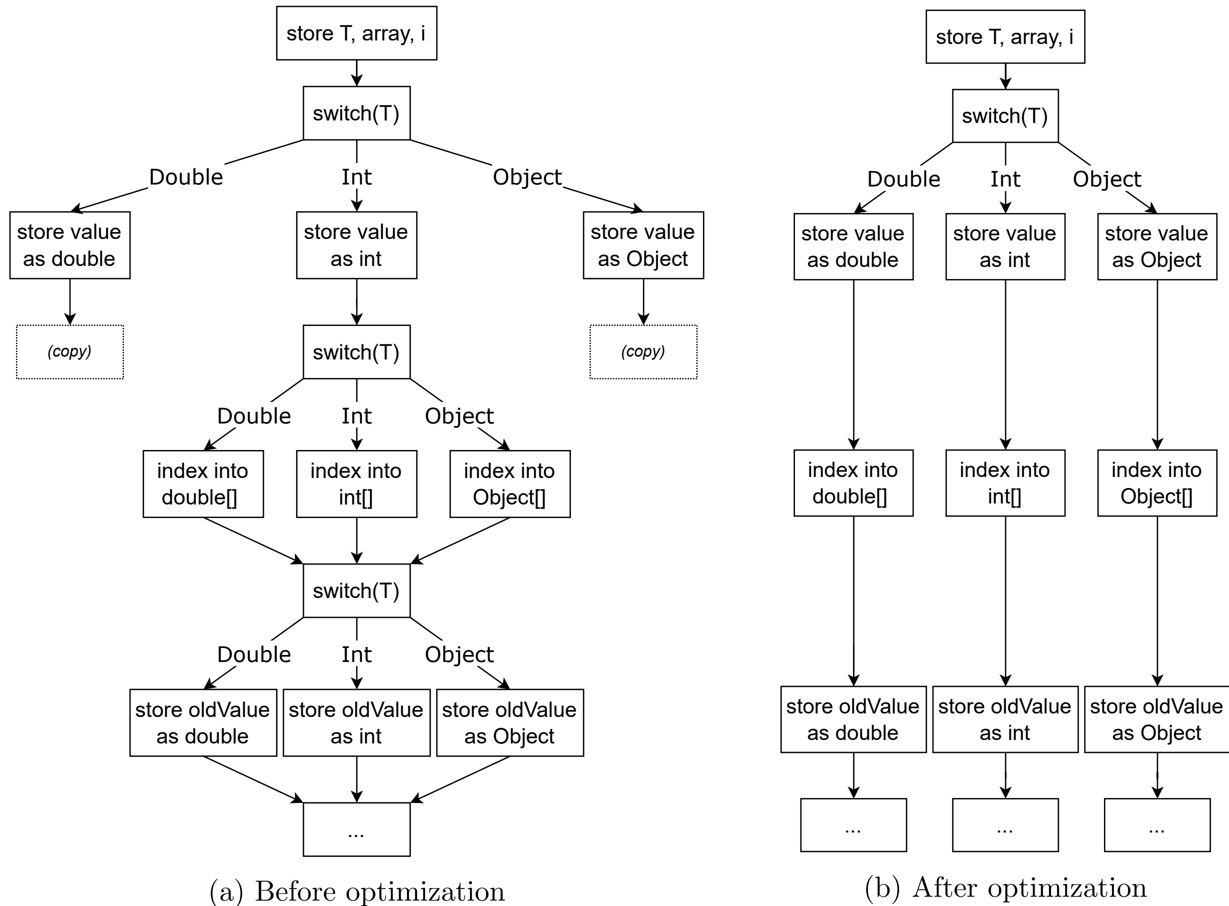


Figure 5.3: Part of the control flow graph for `swap` after tail duplication.

Graal can usually eliminate the control flow splits when generic code is inlined into a non-generic call site. For instance, if `swap` is inlined into a context where the type argument is statically `Int`, Graal eliminates the branches where `T` is not `Int`. However, inlining decisions are complicated, depending on inlining budget, compilation tier, and other compiler parameters. If (for whatever reason) generic code is not inlined into a non-generic call site, the compiled code often performs poorly. Ideally, the performance of generic code should not depend so significantly on whether it gets inlined by the compiler.

The control flow splits in generic code are especially unfortunate because they switch over the same type parameters. When `swap` is invoked with `T` being `Int`, the second branch in Figure 5.2 is taken every time. If the result of the first type switch could somehow be propagated through the tree, the subsequent type switches could be elided.

This approach is the idea behind the *replication* or *tail duplication* [35] transformation performed by optimizing compilers. Instead of merging control flow after a conditional branch, the compiler duplicates the code after the merge into each branch. Then, since the outcome of the branch condition (e.g., a type check) is known in each branch, the compiler can aggressively optimize the code in each branch.

In the case of `swap`, tail duplication might produce a control flow graph like Figure 5.3a. After the first type switch, the rest of the graph gets copied into each branch. Then,

since the value of `T` is statically known inside each branch, the compiler can propagate it through the AST and fold away the redundant type switches. The resulting control flow graph would look something like Figure 5.3b. After the first type switch, all of the remaining type switches can be optimized away. Within each branch, the compiler has precise knowledge about the types of values, which can create further opportunities for optimization.

The motivation for TastyTruffle's specialization is to perform a tail-duplication-like transformation at the AST level. Graal supports tail duplication, but (like inlining) deciding when to perform it is an imperfect process driven by heuristics; improving Graal's tail duplication decisions is an active area of research [32]. If generic ASTs could be transformed during interpretation to achieve the same effect as tail duplication, generic code could enjoy more reliable performance that would not depend on Graal performing the transformation.

5.2 Specializing generic methods

To specialize a generic method, TastyTruffle executes a different copy of the generic AST (a *specialization*) for each set of type arguments passed to the method. Since each set of type arguments uses a different AST, the generic nodes within each AST dynamically specialize themselves to support only a single representation. When each generic node is compiled, it only handles one representation, so within each copy of the AST there are no control flow splits caused by type switches.

Consider the generic `swap` method from Figure 5.1. When `T` is `Int`, `swap` executes a copy of the generic AST only used for `Int`. This AST contains various generic nodes that self-optimize to a monomorphic state. For example, the AST has a `GenericReadLocal` to read the generic `value` variable. Within the `Int` specialization, `T` takes on the value `Int`, so the `GenericReadLocal` specializes itself to read a primitive `int` from the frame. Since `T` is always `Int` for this AST, it never specializes to any other representations, and when Graal compiles it, the resulting code does not contain type switches.

To implement method specialization, the TastyTruffle parser replaces a generic method's body with a `TypeSwitch` node (Figure 5.5). This node uses the same dispatching and caching technique as `GenericShapeTemplate` (Figure 4.16). At run time, the `TypeSwitch` node evaluates the type parameters into `typeArgs`, and then uses `lookup` to search for an existing specialization in its cache. If the method has never been invoked with those arguments, it creates a new copy of the AST; otherwise, it reuses the previously-created specialization. Finally, it calls the specialization. Each copy of the AST is wrapped in a `Method`—the compilation unit in TastyTruffle—so that specializations can be compiled independently from each other.

Figure 5.4 depicts the generic `swap` method after specialization. The body of `swap` consists solely of a `TypeSwitch` node that dispatches to an appropriate copy of its generic AST. In this AST, `swap` has been called with `T` being `Int` and `Double`, so there are two specializations cached on the `TypeSwitch` node.

As with `GenericShapeTemplate`, `TypeSwitch` is carefully written to work well with partial evaluation. When some or all of the type parameters are PE-constant, the type

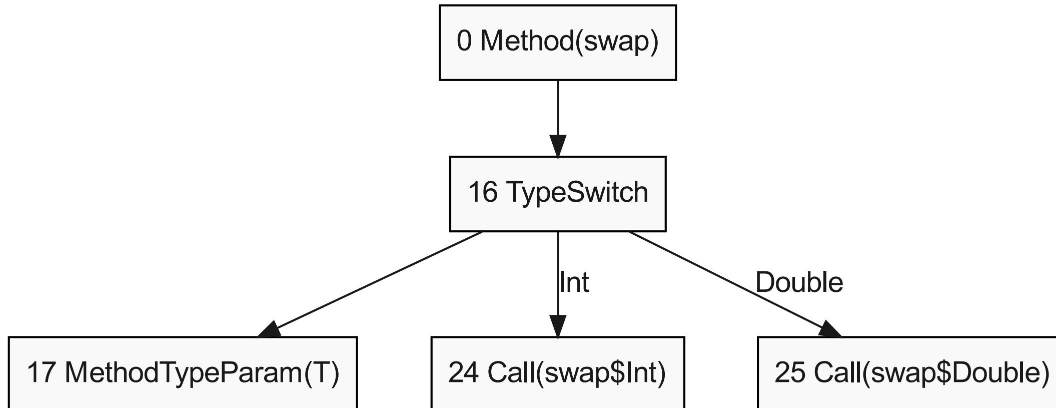


Figure 5.4: Generic AST for `swap` after specialization.

switch can be simplified or even removed, and the specialized ASTs can be inlined directly into the compiled code. Section 4.3.2 discusses how the type dispatch gets compiled in greater detail.

5.3 Specializing generic class methods

To specialize the methods in a generic class, different copies of each method's AST should be used for each unique set of type arguments supplied to the class.

As it turns out, specializing class methods is rather simple, thanks to two important facts:

- TastyTruffle uses the receiver's `Shape` to dispatch method calls (Section 3.4).
- TastyTruffle creates a unique `GenericShape` for each unique set of type arguments supplied to a generic class (Section 4.3).

To support class method specialization, TastyTruffle simply needs to make a copy of each method's AST when a generic class is applied (as opposed to sharing the ASTs). When each `GenericShape` has different copies of its methods, each copy is only invoked with a fixed set of type arguments. The generic nodes in these methods only specialize themselves over these type arguments, and the resultant code is monomorphic.

Consider the `Box` class in Figure 5.6. When `Box` is applied to `Int`, TastyTruffle creates a new `GenericShape` and duplicates its methods. The generic nodes in these methods dynamically specialize themselves to only execute with `T` being `Int`, since they are only executed for `Box[Int]` receivers.

Unlike method specialization, methods specialized over class type parameters do not use `TypeSwitch` nodes. In effect, TastyTruffle performs the type switching at class

```java
1  class TypeSwitch extends Node {
2    @Children TypeNode[] typeParams;
3    @CompilationFinal(dimensions=2) Shape[][] keys;
4    @CompilationFinal(dimensions=1) Method[] values;
5
6    @ExplodeLoop
7    Object execute(VirtualFrame frame) {
8      Shape[] typeArgs = new Shape[typeParams.length];
9      for (int i = 0; i < typeParams.length; i++)
10       typeArgs[i] = typeParams[i].resolve(frame);
11
12     Method specialization = lookup(typeArgs)
13     if (specialization == null) {
14       CompilerDirectives.transferToInterpreterAndInvalidate();
15       specialization = // create new AST and add it to the cache
16     }
17
18     return specialization.call(...);
19   }
20
21   @ExplodeLoop
22   Method lookup(Shape[] typeArgs) {
23     for (int i = 0; i < keys.length; i++) {
24       if (shapesMatch(keys[i], typeArgs)) return values[i];
25     }
26     return null;
27   }
28
29   @ExplodeLoop
30   boolean shapesMatch(Shape[] expected, Shape[] actual) {
31     if (expected.length != actual.length) return false;
32     for (int i = 0; i < expected.length; i++) {
33       if (expected[i] != actual[i]) return false;
34     }
35     return true;
36   }
37 }
```

Figure 5.5: Source code for the `TypeSwitch` node.

```
1 class Box[T](initial: T) {
2   private var value: T = initial
3   def get: T = value
4   def set(x: T): Unit = value = x
5 }
```

Figure 5.6: Generic `Box` class (reproduced from Figure 4.8).

```
1 class List[T] {
2   def map[U](fn: T => U): List[U] = ...
3 }
```

Figure 5.7: A `List` `map` method with both class and method type parameters.

application time: the generic ASTs that should be executed are predetermined as soon as a `GenericShape` is resolved.

Note that a method can be specialized over both class and method type parameters. This generalization poses no problem for the specialization scheme. The `map` method in Figure 5.7 is generic over the class type parameter `T` and the method type parameter `U`. The `map` AST is first duplicated when `List[T]` is applied to a concrete type. Later, when `map` is invoked with a concrete type for `U`, the AST is duplicated again. The net effect is that TastyTruffle uses a unique AST for each unique combination of class and method type parameters.

5.4 Static specialization

Sections 5.2 and 5.3 describe a scheme for *dynamic* specialization, whereby the generic ASTs are monomorphized during interpretation using Truffle's self-optimization. Dynamic specialization requires minimal implementation effort: simply duplicate ASTs, and then let each AST self-optimize to a monomorphic state. This approach is convenient, but has a couple of drawbacks:

- Even though the generic nodes are monomorphic, they can still perform unnecessary computation. For example, the `ClassTypeParam` (Figure 4.11) uses an inline cache indexed on the receiver's shape. Though the receiver's shape may never change, `ClassTypeParam` must still check the shape before yielding the cached type.[1]
- From an engineering perspective, dynamic specialization is not very robust. Suppose there is a bug in the interpreter that causes a copy of the AST to specialize itself to multiple representations. The dynamic specialization approach fails silently, quietly running with degraded performance without alerting the developer of the bug.

[1] Usually, Graal can elide any redundant checks, but (as with tail duplication) TastyTruffle tries to make Graal's job easier by statically performing optimizations on the AST when it can.

For both of these reasons, it is desirable to *statically* specialize generic ASTs. Since each copy of a generic AST is only used when type parameters take on specific values, TastyTruffle can replace generic nodes with non-generic nodes that do not rely on run-time type checks. As with the `TypeNodeSpecializer` (Figure 4.14a), TastyTruffle uses tree visitors to specialize method bodies.

5.4.1 Statically specializing generic methods

TastyTruffle defines a `GenericMethodSpecializer` visitor to statically transform the ASTs of generic methods (Figure 5.8). When a generic method creates a new copy of its AST, it uses `GenericMethodSpecializer` to transform the generic AST to a non-generic AST. The visitor handles all of the generic nodes introduced in Chapter 4.2:

Type parameters: `MethodTypeParams` are replaced with `ConstTypes`.

Local accessors: `GenericReadLocal` and `GenericWriteLocal` nodes are replaced with non-generic local accessor nodes. After specializing the `TypeNode` child of a local accessor, the resulting `specType` is non-generic, and the local's representation can be statically determined.

Array accessors: `GenericArrayApply` and the other array accessor nodes are replaced with non-generic array accessors. Again, their `TypeNode` children are specialized and the array representation is statically determined.

Recall the `swap` method from Figure 5.1. Figure 5.9 depicts the AST after it is statically specialized over type `Int`. The AST is much simpler than the original generic AST (provided for comparison in Appendix A) and all of the generic nodes are replaced with non-generic ones.

5.4.2 Statically specializing generic class methods

Generic class methods can be statically specialized in much the same way as generic methods. However, since methods can be generic over both class *and* method type parameters, the AST produced by specializing over class type parameters can still be generic.

TastyTruffle defines a `GenericClassSpecializer` visitor that statically transforms generic class methods (Figure 5.10). It handles the following kinds of nodes:

Type parameters: `ClassTypeParams` are replaced with `ConstTypes`.

Field accessors: `GenericReadField` and `GenericWriteField` nodes are replaced with non-generic field accessors. A field cannot be generic over method parameters, so the field is always non-generic after specialization. The specializer does not need to specialize any `TypeNodes` since the field accessors use the receiver's `Shape` to resolve `Field` objects.

Local and array accessors: If they are only generic over class type parameters, generic local and array accessors are replaced with their non-generic counterparts. Otherwise,

```
1  class GenericMethodSpecializer(typeArgMap: Map[Symbol, Shape])
2      extends NodeTransformer {
3
4    def visit(node: Node): Node = node match {
5      // Type parameters
6      case param: MethodTypeParam =>
7        new ConstType(typeArgMap(param.symbol))
8
9      // Local accessors
10     case read: GenericReadLocal =>
11       val specType = visit(read.typeNode)
12       new ReadLocal(..., getRepresentation(specType))
13     case write: GenericWriteLocal =>
14       ...
15
16     // Array accessors
17     case apply: GenericArrayApply =>
18       val specType = visit(apply.typeNode)
19       new ArrayApply(..., getRepresentation(specType))
20     ... // other generic array accessors
21
22     case _ => node
23   }
24
25   def getRepresentation(typeNode: Node): Representation = { ... }
26 }
```

Figure 5.8: Source code for the `GenericMethodSpecializer` class (Scala).

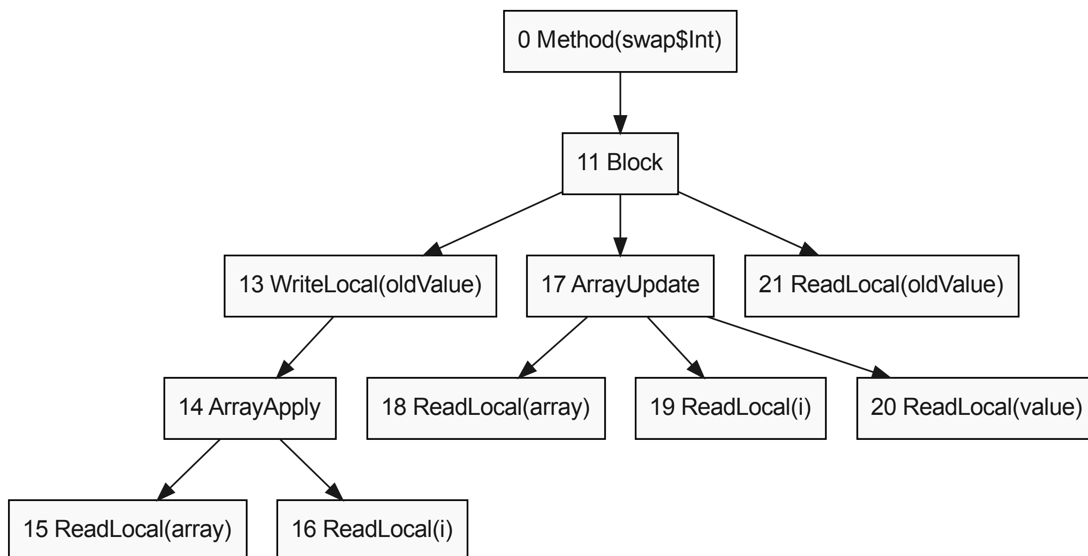


Figure 5.9: AST for `swap` specialized over `Int`.

```scala
1  class GenericClassSpecializer(typeArgMap: Map[Symbol, Shape])
2      extends NodeTransformer {
3
4    def visit(node: Node): Node = node match {
5      // Type parameters
6      case param: ClassTypeParam =>
7        new ConstType(typeArgMap(param.symbol))
8
9      // Field accessors
10     case read: GenericReadField => new ReadField(...)
11     case write: GenericWriteField => new WriteField(...)
12
13     // Local and array accessors
14     case read: GenericReadLocal =>
15       val specType = visit(read.typeNode)
16       if (isConstantType(specType))
17         new ReadLocal(..., getRepresentation(specType))
18       else new GenericReadLocal(..., specType)
19     ... // other generic local and array accessors
20
21     case _ => node
22   }
23
24   def isConstantType(typeNode: Node): Boolean = { ... }
25   def getRepresentation(typeNode: Node): Representation = { ... }
26 }
```

Figure 5.10: Source code for the `GenericClassSpecializer` class (Scala).

the accessors remain generic, but their `TypeNode`s may be partially specialized (e.g., if they are generic over both class and method type parameters).

If a method is generic over both class and method type parameters, it is specialized twice: first by class type arguments, then by method type arguments. The resultant AST is non-generic and can be easily compiled by Graal to monomorphic code.

Chapter 6

Evaluating TastyTruffle

This chapter evaluates the implementation of TastyTruffle on a series of small benchmark programs. The intention of the evaluation is to validate the design of TastyTruffle and explore the consequences of specific design choices.

6.1 Benchmarks

The benchmarks are small Scala programs that use generics. Many of the benchmarks were devised based on existing Scala standard library code. Since the standard library makes extensive use of generics, and most Scala code relies on the standard library, supporting the generic idioms used in these benchmarks is a first step toward supporting more complex generic programs. The standard library uses many features that are not yet supported by TastyTruffle (traits, anonymous functions, etc.), so most of the benchmarks are hand-written in the supported subset of Scala.

The full list of benchmarks is in Table 6.1. There are seven benchmarks loosely ordered by increasing complexity. Later benchmarks often use a superset of the generic idioms used by earlier ones (e.g., in addition to type classes, StdDev also uses higher-order functions).

Every benchmark relies on generic arrays in some form. Benchmarks that do not use arrays, such as graph traversals, were considered for the evaluation, but they were deemed less interesting since their execution time would likely be dominated by pointer chasing.

6.2 Setup

The benchmarks are executed in a few different configurations:

Graal (G): The Scala benchmarks are compiled to JVM bytecode and executed by a Graal-equipped JVM (a typical execution environment for Scala programs).

Unspecialized TastyTruffle (T_U): The Scala benchmarks are compiled to TASTy and executed by TastyTruffle without specialization.

Specialized TastyTruffle (T_S): This configuration is identical to T_U, except specialization is enabled.

Benchmark	Description
ArrayCopy	Copies the contents of one Array[T] to another.
Checksum	Computes a checksum of an Array[T], invoking the ## operator to hash each element.
InsertionSort	Performs insertion sort over an Array[T] using an Ordering[T] type class.
QuickSort	Performs quicksort over an Array[T] using an Ordering[T] type class.
StdDev	Computes the standard deviation of an Array[T] where T is a numeric type. Uses a Fractional[T] to perform mathematical operations, and uses fold and reduce to compute the result in a functional programming style.
ArrayDeque	Defines a generic ArrayDeque[T] that mirrors the standard library. Repeatedly appends elements to a dynamically-resizable buffer.
HashMap	Defines a generic HashMap[K,V] backed by generic key and value arrays. Constructs a map from a set of inputs, then looks up and removes each result.

Table 6.1: Table of benchmarks.

All benchmarks are run on a Ubuntu 22.04.2 system with four 16-core AMD Opteron 6380 processors and 512 GiB of memory. All of the configurations use a Java 17 build of GraalVM Enterprise Edition 22.2.0 with a fixed heap size of 8 GiB.

G is included in the evaluation to assess how TastyTruffle behaves compared to a state-of-the-art implementation. It is worth emphasizing that the comparison serves to validate TastyTruffle's design, not to make an argument that TastyTruffle is a "better" implementation. The benchmarks do not represent real workloads, so it is meaningless to extrapolate about real workloads based on the quantitative results. However, quantitative performance differences between Graal and TastyTruffle usually indicate differences in the way programs are compiled. These differences *are* meaningful and are discussed in the evaluation.

Each benchmark has two different workloads:

Monomorphic: In the monomorphic workload, the benchmark is invoked with a single concrete type. Such a benchmark workload is denoted with a concrete type, like ArrayCopy[Int]. The monomorphic workload is intended to measure best-case performance, where the compiler can often speculatively monomorphize the generic code over the concrete type.

Polymorphic: In the polymorphic workload, the benchmark is invoked with three different concrete types: Int, Double, and a simple BoxedInt class that wraps a primitive Int. A polymorphic workload is denoted with the benchmark name, like ArrayCopy. Since generic code is written to be used with multiple different concrete types, the polymorphic workload gives a more realistic assessment of how the compiler optimizes generic code.

The benchmarks are executed using the Java Microbenchmark Harness (JMH)[1]. The harness method that invokes each benchmark is excluded from compilation so that the generic code is not inlined into a call site with concrete type arguments.

6.3 Throughput

TastyTruffle was designed to enable high performance generic code. To assess whether TastyTruffle achieves this goal, the evaluation primarily focuses on the peak throughput of the benchmarks (i.e., performance after the benchmark code reaches a stable just-in-time compiled state).

Each benchmark is first invoked for several warmup iterations (at least ten iterations at ten seconds each) to ensure that the benchmarks get compiled by Graal. Dry runs were performed before the actual evaluation, wherein the compiler logs and graphs were manually inspected to ensure that benchmarks consistently reached a stable, warmed-up state during the warmup iterations. After warmup, each benchmark is run for five measurement iterations at ten seconds each. Since JIT compilation is so dependent on heuristics and dynamic measurements, warmed-up code can often reach different steady states, and so this process is repeated for five different forked runs. In total, 25 throughput measurements are collected for each benchmark.

Table 6.2 depicts the mean throughput for each benchmark. It includes 99.9% confidence intervals for comparison.[2] For the TastyTruffle configurations, the throughput relative to G is included; it is bolded when the confidence intervals do not overlap (i.e., they are statistically significant).

The following sections discuss the throughput results on the polymorphic and monomorphic workloads. Differences in the throughput can arise for multiple reasons, and the reasons are often interrelated (and challenging to disentangle), so the goal of the discussion is to identify patterns that likely play a role in the results. Additionally, these patterns can help to demonstrate the innate limitations or effectiveness of each implementation strategy.

6.3.1 Polymorphic workloads

In general, T_S runs faster than G on the polymorphic workloads. The performance of T_U sometimes matches T_S, but other times does much worse. There are a few common factors that likely contribute to the observed differences.

Loop unswitching

Recall that T_S avoids polymorphic code by making a copy of the AST for different concrete generic types. It appears that Graal's *loop unswitching* transformation provides similar benefits for G and T_U. If a loop contains a conditional branch, but the branch condition is loop-invariant, loop unswitching "extracts" the conditional branch outside of the loop and

[1] https://github.com/openjdk/jmh

[2] The confidence intervals are constructed using a t-distribution, assuming the measurements are normally distributed.

	G		T_U			T_S		
	ops/s	± CI	ops/s	± CI	rel	ops/s	± CI	rel
ArrayCopy	48.829	± 3.527	162.529	± 42.915	**3.33**	191.058	± 1.712	**3.91**
Checksum	98.055	± 1.710	141.794	± 1.796	**1.45**	143.893	± 6.382	**1.47**
InsertionSort	8.139	± 0.070	23.221	± 0.347	**2.85**	23.405	± 0.130	**2.88**
QuickSort	81.816	± 5.976	39.884	± 3.550	**0.49**	104.348	± 3.326	**1.28**
StdDev	534.339	± 8.638	60.142	± 0.972	**0.11**	829.088	± 15.951	**1.55**
ArrayDeque	283.872	± 6.634	79.583	± 6.105	**0.28**	370.483	± 14.954	**1.31**
HashMap	45.430	± 0.599	47.445	± 1.451	1.04	101.233	± 2.812	**2.23**
ArrayCopy[Int]	1105.282	± 31.261	1030.287	± 15.655	**0.93**	999.140	± 36.414	**0.90**
Checksum[Int]	2291.861	± 19.363	2254.107	± 8.401	**0.98**	2249.684	± 6.365	**0.98**
InsertionSort[Int]	91.153	± 0.895	94.323	± 0.585	**1.03**	90.933	± 0.756	1.00
QuickSort[Int]	526.492	± 12.303	504.695	± 8.985	**0.96**	490.084	± 13.127	**0.93**
StdDev[Double]	3385.695	± 0.592	3121.858	± 12.142	**0.92**	3120.272	± 12.494	**0.92**
ArrayDeque[Int]	1516.108	± 28.420	1523.908	± 80.445	1.01	1480.079	± 67.556	0.98
HashMap[Int,Int]	225.298	± 2.964	568.631	± 5.306	**2.52**	567.485	± 4.819	**2.52**

Table 6.2: Throughput of each benchmark (in operations per second) with 99.9% confidence intervals. For TastyTruffle configurations, performance relative to G is also included (higher is better).

```
1 def copy[T](src: Array[T], dst: Array[T]): Unit = {
2   var idx = 0
3   while (idx < src.length) {
4     dst(idx) = src(idx)
5     idx += 1
6   }
7 }
```

Figure 6.1: Source code for ArrayCopy.

duplicates the loop for each branch. Since most of the work in each benchmark happens inside of a loop, loop unswitching can effectively monomorphize polymorphic benchmark code in the same way as T_S's duplication.

Consider ArrayCopy as an example. The source code for this method is depicted in Figure 6.1. Recall that on the JVM, Scala supports generic arrays using runtime methods that switch over the run-time type of the array (for example, Figure 2.4). The type of the arrays is invariant within the loop, so in theory the loop can be unswitched.

Figure 6.2 depicts a simplified subgraph of the Graal IR produced by the ArrayCopy benchmark on G. In this graph, it appears that Graal only partially unswitches the loop with respect to the `src` argument: the `int[]` case is extracted out of the loop, but the conditional jumps for `double[]` and `Object[]` remain inside the loop. When ArrayCopy is invoked with non-integer arrays, it suffers a performance hit because it must check the argument types on each iteration.

The compiler often has many loop unswitching candidates. Deciding which conditional branches are worth unswitching is driven by branch profiles and other heuristics, which can lead to unpredictable and sub-optimal results. On G, all of the benchmarks except InsertionSort and QuickSort produce graphs with generic array type checks inside

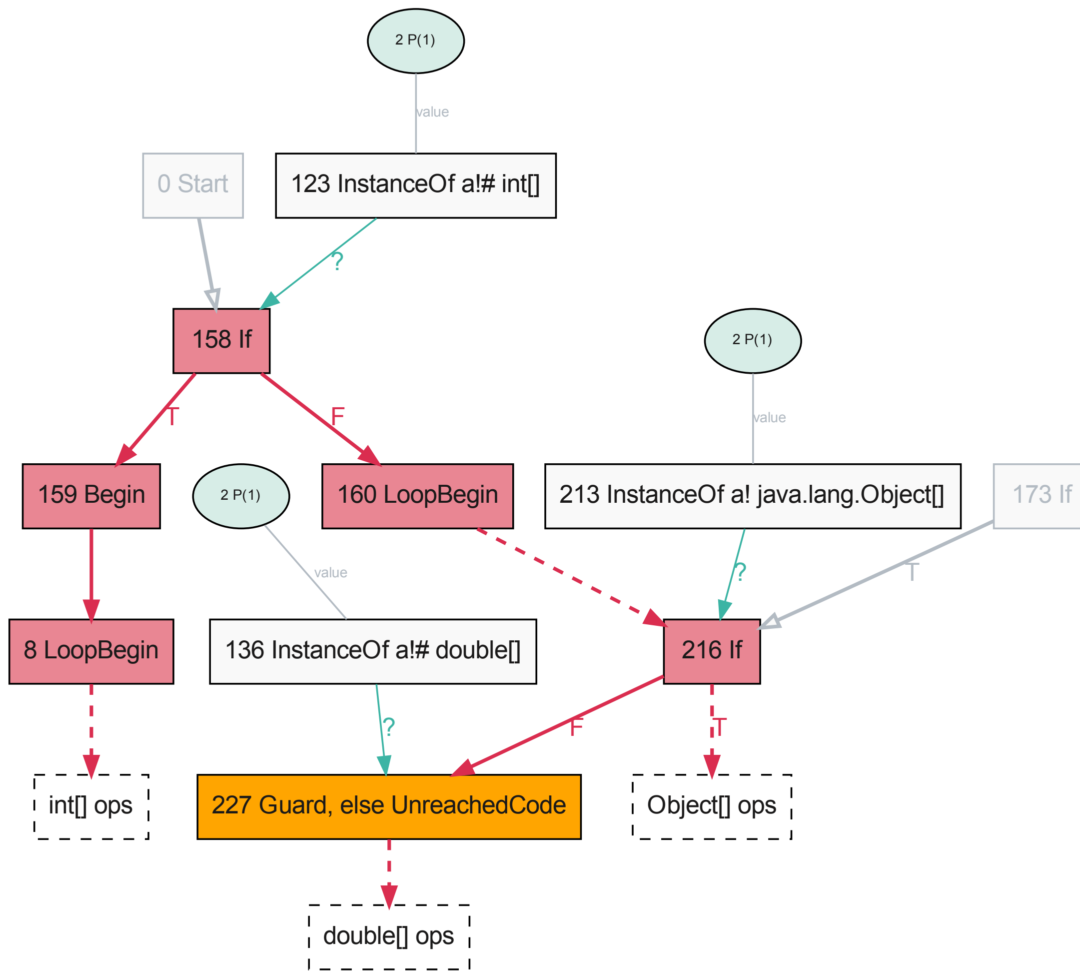


Figure 6.2: Graal IR subgraph for ArrayCopy on G.

loops; sometimes the loops get partially unswitched, like with ArrayCopy, but in other cases there is no unswitching at all.

On T_U, a benchmark's throughput tends to correlate with Graal's ability to unswitch the benchmark's generic type checks. For ArrayCopy, Checksum, and InsertionSort, T_U matches the performance of T_S. In each case, the generic type switches get unswitched out of the loops, resulting in loops that are more monomorphic and amenable to optimization. For the other benchmarks, on which T_U performs much worse than T_S, the type switches are either partially unswitched or not unswitched at all.

The duplication performed by T_S thus appears to be a significant reason for its performance on the polymorphic workloads. Performing code duplication at the AST level, in effect, simplifies the compiler's job: it does not need to guess whether a type check is worth unswitching from a loop, since the transformation is performed automatically.

Explicit type associations

An interesting consequence of reifying types is that it makes type associations among different values more apparent to the compiler. Consider two values of type `Array[T]`. Statically, the Scala type system guarantees that these values have the same type. However, during translation to JVM bytecode, both values are erased to type `Object`—the fact that they have the same concrete type is lost. On G, this leads to code containing redundant type checks and impossible branches.

Figure 6.3 demonstrates an example of this on the ArrayCopy benchmark (Figure 6.1). Both `src` and `dst` are of type `Array[T]`, but in the unswitched branch where `src` is `int[]` (the `T` branch of node `158`) the compiler does not infer that `dst` is also `int[]`. Instead, it dynamically checks whether `dst` (`P(2)`) is an `Object[]` (node `50`) or `int[]` (node `299`). Not only is the former case impossible, but the code that follows it is nonsensical: if `dst` is an instance of `Object[]`, the code then expects it to be a `BoxedInt[]` (node `54`) and tries to store a boxed `java.lang.Integer` into it (node `56`), which would throw an `ArrayStoreException` at run time.

The nonsensical branch is eliminated in later passes (presumably Graal detects the type mismatch), but the extra polymorphism is still undesirable. Graal uses graph size as a heuristic for many optimizations including inlining and loop unswitching. Even if Graal eventually removes the impossible branches, they inflate the graph size and could prevent Graal from performing important optimizations.

In contrast, on T_U and T_S, generic array accesses are implemented not by switching over the array type, but by switching over the reified type parameter. Since both arrays share the same type parameter `T`, after switching over `T` to determine the type of one array, the type of the other array is immediately inferred, and no further type switches are required.

Implicit type associations

A similar type association problem arises in the benchmarks that use type classes. In Scala, type classes are implemented with object instances that get implicitly passed around and can be invoked by client code. All of the benchmarks except ArrayCopy and Checksum use type classes to perform operations over polymorphic values. For example, the sorting

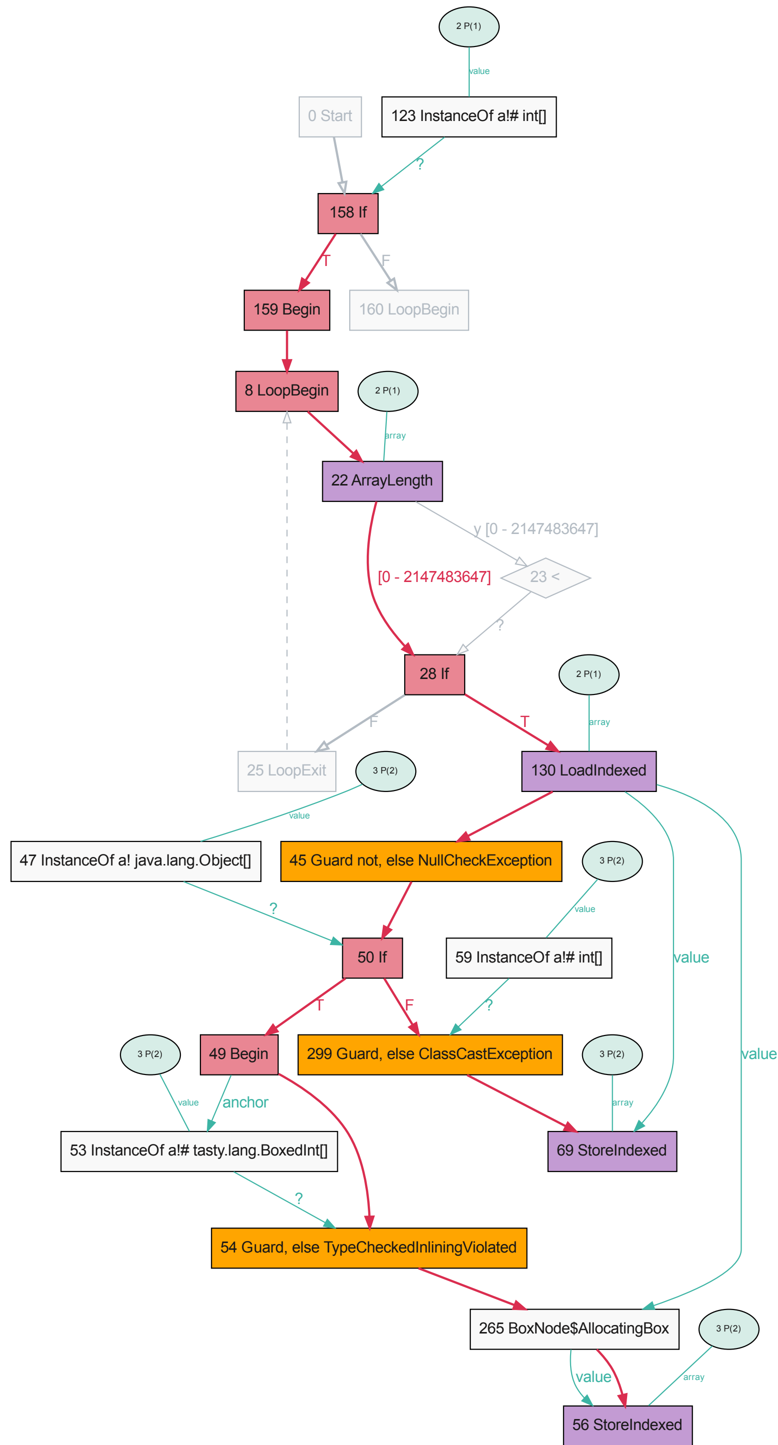


Figure 6.3: Graal IR subgraph for ArrayCopy on G where the source array is `int[]`.

```
1  def insertionSort[T](a: Array[T])(using ord: Ordering[T]): Unit = {
2    var i = 1
3    while (i < a.length) {
4      val next = a(i)
5      // if a(i) < a(i-1), shuffle elements until j s.t. a(j) <= a(i)
6      if (ord.lt(next, a(i - 1))) {
7        var j = i
8        while (j > 0 && ord.lt(next, a(j - 1))) {
9          a(j) = a(j - 1)
10         j -= 1
11       }
12       a(j) = next
13     }
14     i += 1
15   }
16 }
```

Figure 6.4: Simplified source code for InsertionSort.

benchmarks use an `Ordering[T]` instance to compare elements. A simplified version of the InsertionSort benchmark is provided in Figure 6.4.[3] Note the calls to `lt` used to compare the generic elements.

On the polymorphic workloads, InsertionSort observes multiple different `Ordering`s during interpretation. When invoked with `T` being `Int`, it observes an `Ordering[Int]`; when invoked with `T` being `Double`, it observes an `Ordering[Double]`, and so on. In many cases, there is an implicit relation between the concrete type argument and the specific type class instance supplied.

When InsertionSort is compiled on G, the JVM type profiles provide Graal with a list of possible type class instances. Graal can speculatively check for these instances, which makes it easier to inline type class code. However, since types are erased on G, Graal does not understand the correlation between different `Ordering` instances and `T`, which leads to the same sorts of unnecessary type checks as in the previous section.

Though types are reified on T_U, it also suffers from this problem. Unlike with array representations, TastyTruffle does not use type arguments to determine which type class instance to use. Instead, type class instances are passed as extra parameters; the Scala compiler resolves a type class instance for each call site during compilation, and then TastyTruffle uses this information in the TASTy trees to compute and pass type class instances during calls. The generic callee then invokes methods on the type class instances using regular virtual dispatch. When the benchmarks that use type classes are compiled, the inline caches for each virtual method call are used to infer possible types for the type class instances. On T_U, these caches are shared across different type arguments.

A subgraph for InsertionSort on T_U is depicted in Figure 6.5. This graph represents part of an `ord.lt` method invocation. It first performs a type switch over the `shape` of the `ord` instance (nodes `2592`, `2597`, and `4597`). This type switch comes from the inline cache

[3] The actual implementation of InsertionSort, taken from the Scala standard library, is more complicated. The simplified version is presented for the sake of discussion.

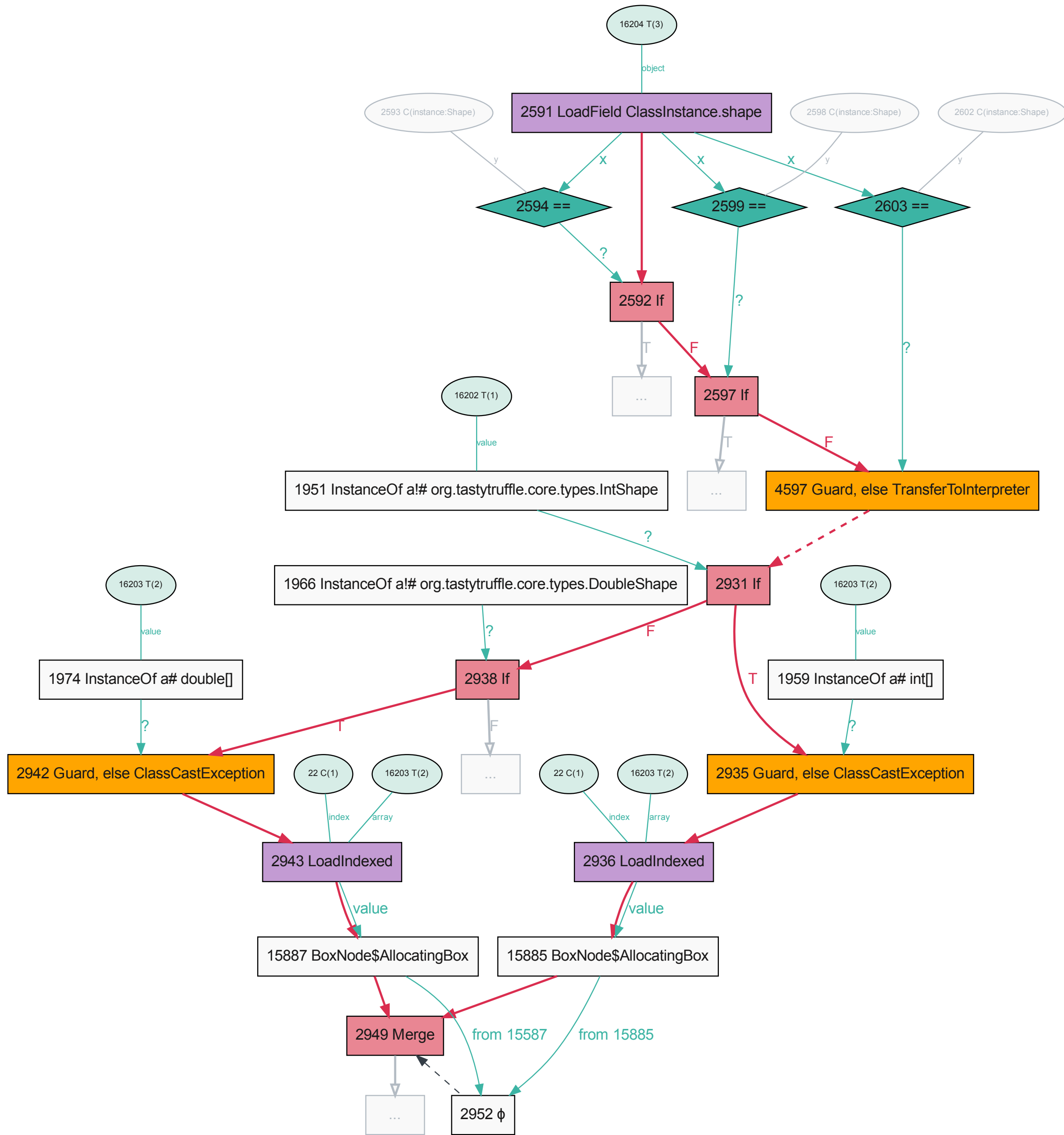


Figure 6.5: Graal IR subgraph for INSERTIONSORT on T_U where `ord` is the `Ordering[Int]` instance.

of an `IndirectCall` node.

In the case where `ord` is the `Ordering[Int]` instance (node `4597`), the code eventually reaches a point where it reads from the array `a`. Another type switch is performed: this time, over the type parameter `T` to determine how to access the array. When the parameter is `Int` (the `T` branch of node `2931`) it casts `a` to `int[]` and loads an element from it. However, the graph also checks whether `T` is `Double` or another type (omitted from the graph for simplicity) and loads an element from the appropriate array representation. Graal cannot determine that `T` must be `Int` on the branch with an `Ordering[Int]`, which leads to unnecessary type switch cases.

Observe also that, due to these extra type switches, intermediate values get modeled by ϕs (e.g., node `2952`). These ϕs are `Object`-typed, which forces primitives to be boxed up unnecessarily (e.g., nodes `15887` and `15885`). These boxing operations can introduce additional performance overhead on the G and T_U configurations.

Implicit associations can be automatically inferred on T_S. Since T_S invokes a different copy of the AST for each concrete value of `T`, each copy has its own, separate set of inline caches. When `T` is `Int`, `ord` is always an `Ordering[Int]` instance; there is no sharing of the type profiles, and Graal can often infer the exact type class instance to use (unless multiple different `Ordering[Int]`s are used). This separation of type profiles is an especially useful consequence of T_S's tree duplication.

Inlining

Whether calls are inlined into a call site is another important factor in performance. Each configuration successfully inlines every method invoked by the ArrayCopy, CheckSum, InsertionSort, and ArrayDeque benchmarks. With QuickSort, StdDev, and HashMap, which are more complicated, the configurations have varying degrees of success with inlining.

The main source of methods not being inlined appears to be type class methods on G. For example, on QuickSort, the calls to `Ordering` methods do not always get inlined. G can successfully inline these methods on the monomorphic workloads (e.g., QuickSort[Int]), so the polymorphic call sites likely pose a challenge for the inliner. T_U and T_S are always able to inline type class methods.

Another interesting case where inlining fails is StdDev on T_U. The source code for StdDev is depicted in Figure 6.6. The benchmark is written in a functional programming style with generic `reduce` and `fold` methods. The `fold` method does not get inlined into the main benchmark method on T_U, but it does on G. This is counter-intuitive because Truffle's inlining policy is generally more aggressive than Graal's. It appears that the size of generic methods on T_U can grow significantly when the type profiles are polymorphic: on T_U, `fold`'s initial graph from the monomorphic workload has 178 nodes, whereas the graph from the polymorphic workload has 4351 nodes (a 24× increase). T_U may not scale well with larger programs because of this limitation.

There is a similar code size concern for T_S. Since generic methods use type switches with calls to different specialized versions of the AST, the entire graph for a generic method may be too big to inline. However, since the type parameter(s) to a generic method call

```
1 def computeStdDev[T](array: Array[T])(using fractional: Fractional[T
     ]): Double = {
2   val N = fractional.fromInt(array.length)
3   val sum = reduce[T](new Add[T], array)
4   val mean = fractional.div(sum, N)
5   val numerator = fold[T, T](
6       fractional.zero, new AddSquareDistance[T](mean), array)
7   val variance = fractional.div(numerator, N)
8   Math.sqrt(fractional.toDouble(variance))
9 }
```

Figure 6.6: Source code for StdDev.

are statically known to the inliner[4], it can (with the help of partial evaluation) detect that it only needs to inline one specialization. For example, when `T` is `Int`, a call to `fold[T]` only needs to inline the `fold$Int` specialization rather than the entire `fold[T]` method and all of its specializations.

6.3.2 Monomorphic workloads

On all of the monomorphic workloads besides HashMap[Int,Int], the three configurations perform comparably. Sometimes the TastyTruffle configurations perform marginally worse than G, but the code produced is structurally similar. In fact, many of the hot loops are compiled to identical low-level code. It is possible that the way Truffle code is invoked in the benchmarks—through the polyglot API, which coerces each host (Java) value to a guest (TastyTruffle) value—introduces an extra overhead on the Truffle configurations.

The fact that the results are so similar suggests that the presence of type information in T_U and T_S does not give them much of an advantage on monomorphic workloads. In G, though the JVM does not do the same degree of profiling as Truffle interpreters, it does collect type profiles at virtual method call sites and `instanceof` checks. On monomorphic workloads, these profiles observe just a single type, so Graal can speculatively compile the benchmarks to monomorphic code that handles the single type observed by the profile. This hypothesis is consistent with the graphs produced on G.

Scala runtime methods

Unlike the other benchmarks, G is not able to monomorphize HashMap[Int,Int], which leads to polymorphic code that gets poorly optimized. The reason it is not monomorphized has to do with Scala's runtime accessor methods.

Recall that operations on generic arrays are proxied through runtime methods like `ScalaRunTime.array_apply`. Since the JVM's type profiling is limited to virtual method

[4] On T_S, type parameters at a call site are always statically known during inlining. Only specializations of generic code are compiled, so any type parameter `T` used at a generic call site is replaced by a constant type during specialization. For example, `fold[T]` is not inlined into `computeStdDev[T]`, but into a particular specialization like `computeStdDev$Int` where `T` is known.

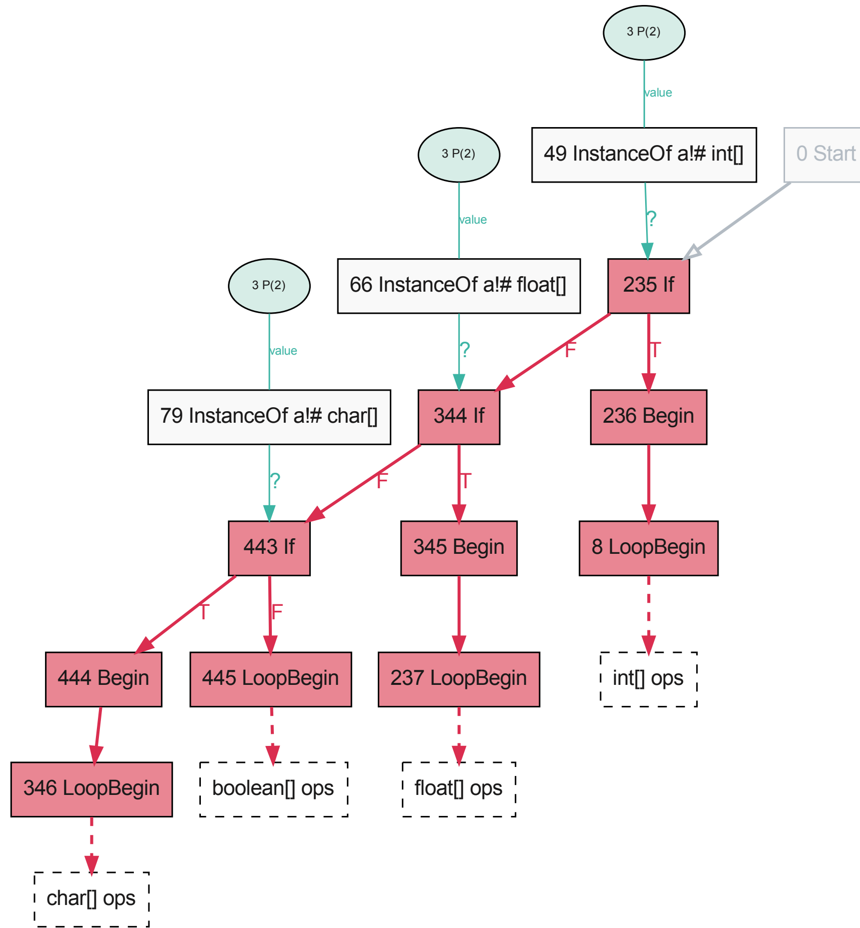


Figure 6.7: Graal IR subgraph for ArrayCopy on G when the generic array accessors are used with other types.

calls and `instanceof` checks—neither of which is performed by the benchmarks—the benchmarks themselves do not collect any profiles about the types of arrays they encounter. It is the type profiles of the runtime methods that get used during compilation to infer possible types for the generic arrays. These type profiles are *shared globally* by all code that accesses generic arrays.

For example, Figure 6.7 depicts a subgraph for the ArrayCopy[Int] workload when it is run on a JVM where the generic array accessors are also heavily used with `Float`, `Char`, and `Boolean` arrays. Even though `copy` is only ever invoked with `Int`, the polluted type profiles on the accessor methods lead to extra branches for `float[]` (node `344`), `char[]`, (node `443`) and `boolean[]` (inside the loop at node `445`).

Thus, the performance results on G are somewhat of a fluke. The performance can degrade arbitrarily depending on how frequently the generic accessors are used with other types. In the case of HashMap[Int,Int], the type profiles for these accessor methods are

```
 1 def put(key: T, value: U): Unit = {
 2   ...
 3   var idx = hash(key)
 4   while (keys(idx) != /* default value */) {
 5     // linear probing
 6     if (keys(idx) == key) {
 7       // overwrite value
 8       values(idx) = value
 9       return
10     }
11     idx = idx + 1
12   }
13   ...
14 }
```

Figure 6.8: Source code for the `put` method of HashMap.

heavily polluted with non-`int` arrays, so the benchmark code could not be monomorphized. Even after carefully hand-writing the benchmark harness code in Java to avoid using the array accessors, the profiles were still polymorphic, so it is unlikely that real programs would encounter monomorphic profiles for the array accessors.

Instead of runtime accessor methods, TastyTruffle uses intrinsic nodes to implement generic array accesses. Each node collects its own type profile, which prevents this global type pollution issue and allows PE to know the precise set of array types that flow into each generic node.

Case study: `HashMap.put`

This section explores a specific example from the HashMap[Int,Int] workload on G. Due to an interesting combination of factors, the compiler is forced to introduce boxing operations even when it knows that it is working with `int`s.

The boxing occurs inside a subsection of the `put` method listed in Figure 6.8. The code performs linear probing to find an index to store a new table entry. On line 6, it compares the input `key` against a value stored in the `keys` array. Because of Scala's erasure, `key` automatically gets boxed to an `Integer`. In the other monomorphic benchmarks, Graal is able to elide the boxing, but it fails on this workload.

The initial part of the graph for this comparison is depicted in Figure 6.9. The compiler has unswitched the main benchmark loop, so it knows that `keys` is an `int[]` and `key` comes from an `int` value. In spite of this knowledge, Graal is unable to remove the automatic `Integer` boxing. There are a few factors that introduce this issue:

1. The left-hand side of the comparison, `keys(idx)`, is implemented with a call to `ScalaRunTime.array_apply`. The inliner decides, based on its heuristics, to not inline this call (node `10534`), so the result has type `Object`.

2. In Scala, generic `==` uses a pointer comparison as a quick check (node `10536`) followed by a slower fallback call to the first operand's `equals` method (not depicted). If

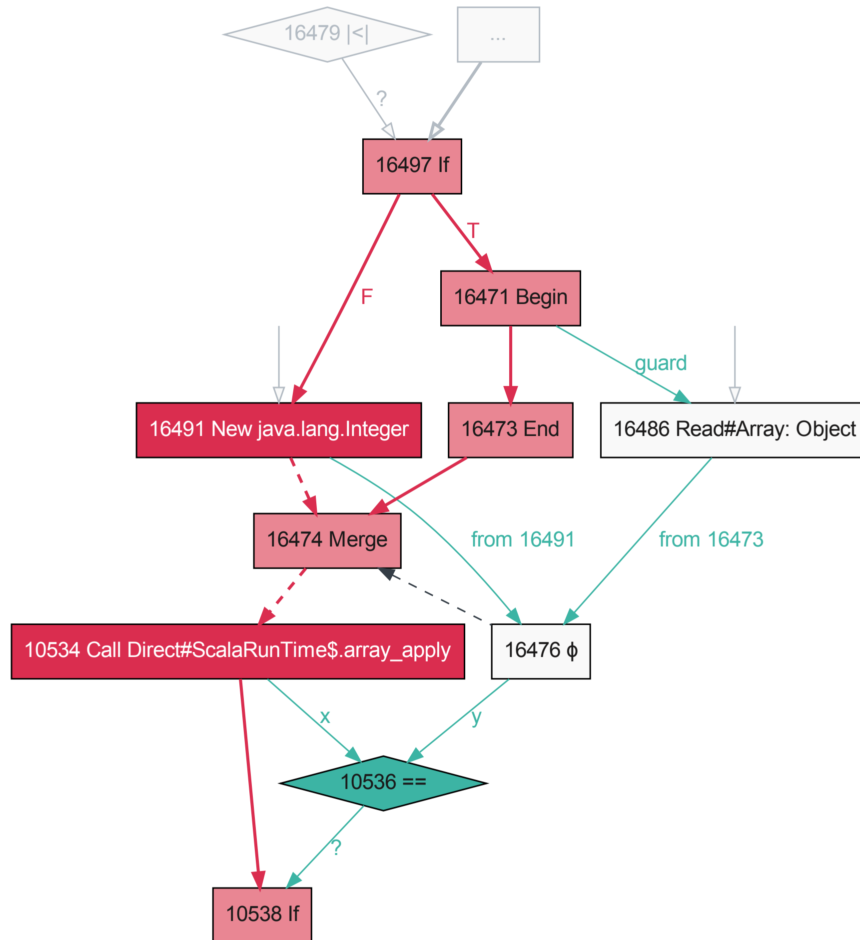


Figure 6.9: Graal IR subgraph for the `put` method of HashMap[Int,Int] on G.

	$T_{S(\text{static})}$		$T_{S(\text{dynamic})}$		
	ops/s	± CI	ops/s	± CI	rel
ArrayCopy	191.058	± 1.712	190.160	± 9.040	1.00
Checksum	143.893	± 6.382	131.556	± 5.357	**0.91**
InsertionSort	23.405	± 0.130	23.046	± 0.308	0.98
QuickSort	104.348	± 3.326	97.459	± 5.561	0.93
StdDev	829.088	± 15.951	825.759	± 18.975	1.00
ArrayDeque	370.483	± 14.954	378.473	± 16.001	1.02
HashMap	101.233	± 2.812	101.130	± 2.365	1.00
ArrayCopy[Int]	999.140	± 36.414	1021.750	± 18.477	1.02
Checksum[Int]	2249.684	± 6.365	2253.436	± 10.291	1.00
InsertionSort[Int]	90.933	± 0.756	90.782	± 1.015	1.00
QuickSort[Int]	490.084	± 13.127	489.101	± 18.242	1.00
StdDev[Double]	3120.272	± 12.494	3118.788	± 11.106	1.00
ArrayDeque[Int]	1480.079	± 67.556	1558.046	± 52.762	1.05
HashMap[Int,Int]	567.485	± 4.819	566.154	± 4.556	1.00

Table 6.3: Throughput of each benchmark using static and dynamic specialization (in operations per second) with 99.9% confidence intervals. The performance of dynamic specialization relative to static specialization is also included (higher is better).

`array_apply` were inlined, the compiler could determine that both operands were `int`s and hence that the pointer comparison would be unnecessary. It could instead elide the boxing and perform a simpler integer comparison. Since the call is not inlined, `key` on the right-hand side must remain a boxed `Integer` so that the pointer comparison can be performed.

3. Normally, a pointer comparison between one object and a freshly-created object always evaluates to `false` (since they must be different objects), so there is still an opportunity for Graal to elide the boxing by completely removing the reference-equality check (node `10536`). However, part of the Java Language Specification (JLS) requires `Integer`s between -127 and 128 to be interned and reused to reduce the performance penalty of boxing. Therefore, the boxing operation has two branches: one where a new `Integer` is allocated (node `16491`), and another where an interned value is read from a cache (node `16471`). Since the resultant value (node `16476`) is not necessarily a fresh object, Graal cannot elide the pointer comparison, and so the `int` value that gets boxed in the call to `put` must remain boxed.

This example demonstrates a limitation of G. Even when Graal can infer the types of generic values, things can still go awry because of unfortunate inlining decisions and unexpected interactions between the type system and library code.

6.3.3 Comparing dynamic and static specialization

Two approaches to specialization are presented in Chapter 5. The first, dynamic specialization, simply duplicates the generic AST for each set of type arguments. The AST self-optimizes itself to a monomorphic representation during interpretation. The second approach, static specialization, actually transforms the generic AST to a non-generic AST.

Since generic code that uses dynamic specialization must perform the occasional type check—specifically, to guard on the generic receiver's shape—it was suggested that the static approach might be marginally more performant.

Table 6.3 depicts the throughput of the dynamic approach ($T_{S(\text{dynamic})}$) relative to the static approach ($T_{S(\text{static})}$). There appears to be no significant difference in throughput between the two configurations (the Checksum difference is the result of an unrelated bug).

Inspecting the graphs for $T_{S(\text{dynamic})}$, the occasional type check does appear in the graphs for the generic class methods. However, T_S's duplication ensures that each AST is monomorphic, so the type check only amounts to a single field load and pointer comparison. The check is a guard rather than a branch, so if the pointer comparison fails, the code deoptimizes (type information is not lost by branching). In the graphs for the actual benchmark methods these type checks often get elided because the receiver's type is statically known.

Since there seems to be no significant difference between the two approaches with respect to throughput, the choice between them is more of a software engineering concern. Static specialization requires more implementation effort, since the developer must define a rewrite rule for each generic node in the AST, but it also provides better guarantees that the AST does not silently enter a polymorphic state.

6.4 Transient behaviour

Though the focus of TastyTruffle's design is on steady-state performance, it is useful to understand the transient behaviour as well. This section performs a smaller experiment with a subset of the benchmarks to assess how TastyTruffle performs as it warms up.

In this experiment, Checksum, HashMap, and QuickSort are used. Both monomorphic and polymorphic workloads are tested. Each benchmark is run for 30 seconds, split across 150 JMH iterations of 200 ms each. This process is repeated for ten different forked runs.

6.4.1 Warmup

Figure 6.10 plots the average throughput over time for each benchmark. The 99% confidence interval for the throughput in each 200 ms interval is depicted in a translucent colour. In general, the throughput is highly variable at the start as the benchmarks warm up. Over time, the throughput tends to stabilize after the JIT compiles the hot code.

Generally, G warms up much faster than either TastyTruffle configuration. For each benchmark, the first iteration on G has a non-negligible throughput, but the TastyTruffle throughput is nearly zero. The G throughput ramps up earlier than TastyTruffle as well. This makes sense, because on G the benchmarks are JVM bytecode. The JVM can achieve at least *some* performance while interpreting them, and it can compile the bytecode directly. In contrast, the Scala programs implemented by TastyTruffle are one layer of abstraction above the JVM. The Truffle interpreters, interpreted on the JVM, are too slow to achieve much throughput, so the host code (e.g., the nodes and their `execute`

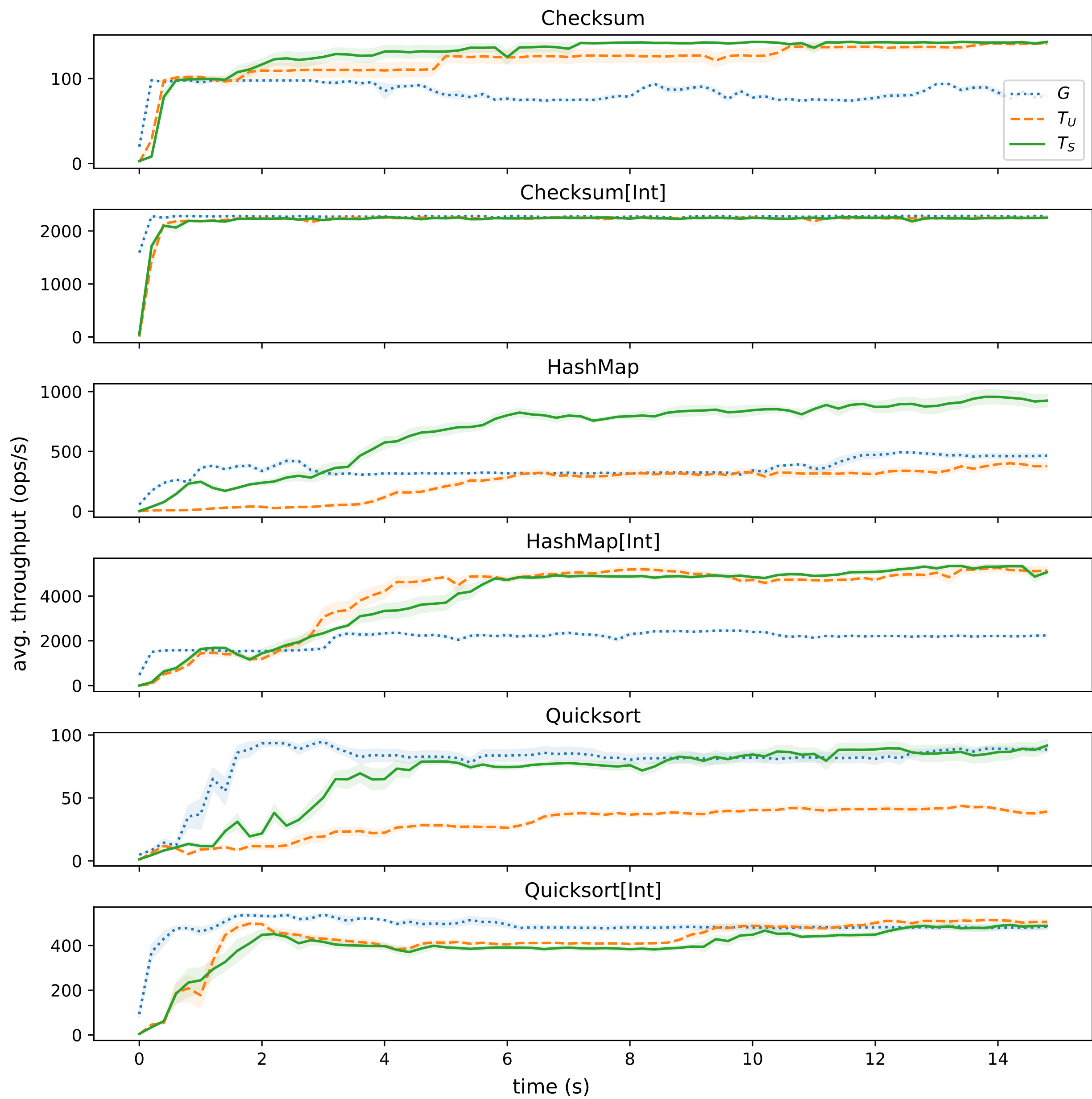


Figure 6.10: Average throughput over time for CHECKSUM, HASHMAP, and QUICKSORT on monomorphic and polymorphic workloads.

methods) must usually be compiled first by the JVM. Then, once the host code is fast enough to interpret the guest code for enough iterations, the guest code can be compiled by Truffle.

The effect of tiered compilation

Graal uses two tiers of compilation. The tiered compilation is visible in some of the warmup graphs. For example, HashMap on T_S jumps up to around 250 ops/s in the first couple of seconds, then later jumps up to over 800 ops/s. Since different methods within the system get compiled concurrently, the separation between these tiers is not always so apparent.

Sometimes, T_S seems to reach peak performance before T_U. For example, on the QuickSort benchmark, T_S reaches peak at around 5 seconds, whereas T_U peaks after 7 seconds. One possible explanation for the difference lies in Graal's tiered compilation. Since second-tier compilation cannot occur until the code exceeds an invocation threshold, how long it takes to execute the first-tier version of a method may affect the time to trigger second-tier compilation. Since T_U is highly polymorphic, Graal may not be able to optimize first-tier compilations very much. In contrast, on T_S each specialized AST can be first-tier compiled separately, and each AST is monomorphic, so the compiler may be able to perform more optimization. The first-tier compiler graphs for QuickSort lend credence to this hypothesis. On T_S, the graph for each specialization loops over a single array type, but on T_U the graph is polymorphic over the three array types used in the benchmark.

Thus, it appears the code duplication of T_S may enable better first-tier compilation, which in turn may lead to faster warmup than T_U.

6.4.2 Compilation

One risk in duplicating generic ASTs (as T_S does) is that it could put additional pressure on the compiler: instead of having a single call target to compile, there is one call target per set of type arguments.

Truffle's `--engine.CompilationStatistics` flag was used to collect compiler during the warmup experiment. A summary of the compilation counts and time spent compiling averaged across the ten runs is depicted in Table 6.4.[5] The counts refer to the number of guest language Scala methods compiled by Graal. Based on this sample of benchmarks, it appears that T_S does cause a modest increase in Truffle compilations (at worst, 29% more for Checksum). The compiler also generally spends more time compiling on T_S than T_U.

Table 6.5 depicts summary statistics about the size of Truffle compilations. For the polymorphic workloads, T_S produces smaller compilations than T_U on average. T_S also produces less compiled code overall, despite compiling more methods. Significantly, in the case of HashMap, the amount of code installed on T_S is nearly half that of T_U. This code size reduction is likely a result of the monomorphism of T_S's generic ASTs. Since the ASTs are monomorphic, the compiler can produce simpler, smaller compilations. On the monomorphic workloads, there is generally not much of a difference in code size.

An exception to these trends is HashMap. For both the monomorphic and polymorphic workloads, the maximum tier-two compilation size is noticeably larger on T_S than T_U.

[5] The JVM does not expose compilation statistics, so only Truffle compilations are included.

	T_U				T_S			
	tier 1	tier 2	total	time (s)	tier 1	tier 2	total	time (s)
Checksum	8.0	13.0	21.0	1.3	10.0	17.0	27.0	1.1
HashMap	55.1	50.7	105.8	35.4	63.2	61.5	124.7	49.1
QuickSort	26.6	26.3	52.9	9.8	30.1	25.0	55.1	12.7
Checksum[Int]	5.0	8.0	13.0	0.7	6.0	9.0	15.0	0.7
HashMap[Int,Int]	26.3	25.8	52.1	12.3	27.3	26.8	54.1	13.4
QuickSort[Int]	13.4	13.3	26.7	2.6	16.4	16.1	32.5	4.2

Table 6.4: Truffle compilation counts and timing for Checksum, HashMap, and QuickSort averaged over ten forks.

	T_U					T_S				
	t1 avg	t1 max	t2 avg	t2 max	total	t1 avg	t1 max	t2 avg	t2 max	total
Checksum	2.9	9.7	2.7	7.4	58.1	1.9	9.7	1.8	7.4	48.7
HashMap	10.4	65.8	4.6	30.8	806.7	3.0	33.7	4.1	39.4	439.7
QuickSort	7.6	30.6	5.1	27.7	335.6	2.4	15.2	3.8	18.0	166.6
Checksum[Int]	1.6	4.8	1.8	5.9	22.5	1.4	4.8	1.6	5.9	23.2
HashMap[Int,Int]	2.4	11.3	3.1	26.4	144.3	2.3	11.3	3.4	37.0	154.4
QuickSort[Int]	2.2	10.4	3.2	11.2	72.4	2.0	10.4	3.8	10.9	94.1

Table 6.5: Truffle compilation size data (in KiB) for Checksum, HashMap, and QuickSort averaged over ten forks.

This requires more investigation, but based on the compiler graphs, it appears that T_S is able to inline more methods directly into these compilations than T_U.

Chapter 7

Related Work

The design and implementation of TastyTruffle is heavily informed by past work. This chapter discusses some of this work.

7.1 Implementing parametric polymorphism

Parametric polymorphism, first distinguished from other forms of polymorphism in [44], can be implemented in a variety of ways. Morrison et al. [34] distinguished among three main categories of implementations:

- *Uniform polymorphism*, wherein generic functions operate on generic data using a single uniform representation. This approach, also known as *erasure*, was popularized by Java [9]; mitigating its limitations is a major reason for the TastyTruffle project.
- *Textual polymorphism*, wherein the compiler generates specialized functions and representations for each set of type arguments a generic definition is used with. The template system of C++ [5] uses this approach, also known as *monomorphization*.
- *Tagged polymorphism*, wherein generic data uses specialized representations, but generic functions have one implementation. Generic data is tagged with extra metadata that allows code to dynamically determine the representation of the data. Tagged polymorphism fits somewhere in between erasure and monomorphization: only a single copy of code is generated, but it must dynamically support heterogeneous data representations.

Morrison et al. introduce a form of tagged polymorphism that stores tags not on the generic data itself (which can be inefficient) but alongside generic data as first-class values in the language. These tags can be used to perform type-sensitive operations on generic data at run time. This approach is more commonly known today as *type reification*, which is at the heart of TastyTruffle's approach. The type-theoretic basis for reified types, sometimes called *intensional type analysis*, is well-studied in works such as [24] and [15].

TastyTruffle's implementation of parametric polymorphism is similar to the approach taken by the .NET Common Language Runtime (CLR) [27]. Unlike JVM bytecode,

the CLR intermediate language preserves generic type information, so the CLR can reify generic type arguments at run time. The CLR uses these reified types to dynamically specialize generic methods and classes, producing efficient, box-free code. The run-time type information is also accessible to languages hosted on the CLR (e.g., C# or F#); TastyTruffle does not support type introspection since that would alter the semantics of Scala programs. TastyTruffle and the CLR also differ in how they handle reference-type specializations. Since reference types have the same data representation, values with different reference types can be treated uniformly; the CLR uses a single specialization for all reference-type instantiations to avoid creating too many specializations. Though TastyTruffle creates a separate specialization for each reference-type instantiation, the type profiles from each specialization are independent, so the compiled code has less polymorphism. This tradeoff between creating additional specializations and producing less-polymorphic code deserves more investigation in future work.

7.2 Working around type erasure on the JVM

JVM language implementations can suffer from poor performance because of type erasure. Erasure also limits the expressivity of generic code, since certain operations (like type comparisons) are impossible. Some existing work aims to address these problems in a few different ways.

7.2.1 Static specialization

TastyTruffle is not the first project to specialize Scala programs. Dragos and Odersky [20] extended the compiler to support static specialization. By annotating type parameters with `@specialized`, programmers can direct the compiler to automatically generate specializations for each primitive type, leading to more optimized generic code. Since Scala has nine primitive types, `@specialized` methods can generate a lot of extra code, which has been a barrier to the adoption of `@specialized` in the standard library. Ureche et al. [45] curb the code size issue using "miniboxing", which creates a single specialization for 64-bit `Long`s. Since other primitive types can fit in a `Long`, they can use the `Long` specialization with the help of additional run-time conversions. Miniboxing was unfortunately never merged into the Scala project, likely due to its complexity. Since Truffle supports implicit conversions among data types, it is possible that TastyTruffle could incorporate a miniboxing-like approach into its specialization scheme.

Since Scala classes support separate compilation (i.e., Scala compiles under an open-world assumption), it is generally impossible for the compiler to know how a generic definition is instantiated at run time. Some works have found success in using whole-program static analyses (i.e., analysis under a closed-world assumption) to analyze and optimize Scala code [19, 37]. For example, [37] presents a context-sensitive call graph analysis that incorporates static information about type arguments. Including type arguments in the analysis increases the precision and enables the compiler to perform automatic specialization of generic methods.

Work has also been done to improve the usability of generics in Java code. Early

on in Java's adoption of generics, Cartwright and Steele [11] proposed NextGen, which uses an alternative compilation strategy for generics. The code compiled by NextGen makes generic types accessible at run time for type-related operations like `instanceof`. NextGen works by generating type-specialized classes that wrap generic class instances. This scheme is designed to improve the ergonomics of generics rather than to improve performance, but it is possible that an optimizing compiler like Graal could use the extra static type information to produce specialized code.

A major limitation of NextGen is that it does not support primitive type instantiations; the authors cite the type-incompatibility between primitives and reference types as the reason. This friction between Java's primitive and reference types is a long-standing area of research. Project Valhalla [4] is an ongoing project that aims to evolve the Java language to "heal the rift between primitives and objects." In particular, its Universal Generics proposal allows type parameters to range over both reference and primitive types, which is intended to enable run-time specialization of generic code in the future.

Some work on Java focuses on ahead-of-time (AOT) transformations to optimize generic code. Graur et al. [23] developed JCS, a tool to specialize some classes from the Java Collections library. JCS, invoked before regular Java compilation, can be used to generate type-specialized copies of Collections classes, like `HashMap<String, Int>`. These classes enjoy specialized representations for their generic fields and locals, and can be imported and referenced directly in Java applications.

A common idea with these static approaches is to circumvent the lack of run-time types in the JVM by generating specialized code during or before compilation. TastyTruffle avoids erasure altogether, using reified types to defer code specialization until run time. Whereas most of these approaches require the programmer to choose what code gets specialized, TastyTruffle performs specialization automatically.

7.2.2 Reifying Scala types

Schinz [39] designed a compilation strategy that reified all Scala types as JVM classes at run time. The reified types enable Scala code to perform type-specific operations like pattern matching and `isInstanceOf` checks over generic types. In an empirical evaluation, the approach introduces significant overheads with respect to running time and memory consumption. The overheads are somewhat expected, since the approach incurs the cost of reification but does not use the reified types for specialization. In contrast, TastyTruffle models only enough types to enable generic code specialization, and types are modeled with built-in interpreter intrinsics rather than language-level objects, which can be more amenable to optimization. The extra overhead of reifying types in TastyTruffle (much of which can be optimized away during JIT compilation) is outweighed by the performance gains it enables via specialization.

7.3 Just-In-Time (JIT) compilation of Scala programs

Since Scala typically runs on a JVM, research on its performance focuses primarily on static compilation strategies that enable the JVM to execute Scala code more effectively.

Nevertheless, there has been some work that explores JIT compilation of Scala and the transformations that are especially useful.

Stadler et al. [42] explored the relative impact of different Graal optimizations on Scala and Java benchmarks by selectively disabling them and comparing the performance. They observe that Graal more effectively optimizes the Scala benchmarks than the Java benchmarks because Scala code "contains more opportunities for optimization." In particular, they find profiling types and branches, inlining polymorphic call sites (based on receiver type profiles), and intrinsifying native methods (like `System.arraycopy`) to be especially effective. By virtue of running on top of Graal and being implemented with idiomatic Truffle code, TastyTruffle enjoys many of these optimizations. Interestingly, tail duplication has no impact on the Scala benchmarks. The authors speculate that the optimization is not "clever" enough since it only performs a local analysis. In a way, TastyTruffle circumvents this limitation by performing tail duplication in the interpreter itself.

A related work by Prokopec et al. [38] uses case studies to demonstrate how Graal optimizations can aggressively simplify Scala collections code. The authors cite the genericity of Scala's collections library code as a major reason for its performance overhead. They also remark that, while JITs can be very effective, they are fundamentally limited by the data representations chosen by the language. The authors encourage further research to improve data representations. TastyTruffle, which avoids the boxing of primitive values in generic contexts, aims to be a first step in this direction.

7.4 Truffle interpreters

The Truffle framework [49] is a general framework for writing high-performance language implementations. There are many well-established Truffle implementations, including TruffleRuby [40], Graal.js [3] and Espresso [31] (a JVM bytecode interpreter). Truffle interpreters make extensive use of type specialization at the AST node granularity, but TastyTruffle is the first Truffle interpreter (to our knowledge) to specialize entire generic ASTs using reified types.

Recent work has explored the specialization of data layouts within Truffle interpreters. Makor et al. [33] designed a technique to dynamically transform a JavaScript array of objects to a columnar layout (i.e., each object property is stored in a separate array) based on the access patterns observed at run time. In follow-up work, Kloibhofer et al. [29] introduced a code duplication transformation that improves the performance of columnar arrays. Data layout specialization is a possible avenue for future work in TastyTruffle.

Chapter 8

Conclusion

Many programming language implementations use erasure to support parametric polymorphism. Erasure is inherently limiting to performance: not only does it introduce boxing overheads and other run-time indirection, but it also destroys type information that could otherwise be used by the implementation to achieve high performance. Scala's standard implementation, which runs on the Java Virtual Machine (JVM), is no exception.

This thesis presented TastyTruffle, a Scala implementation that interprets TASTy IR instead of JVM bytecode. By taking advantage of the rich type information encoded in TASTy, TastyTruffle is able to achieve high performance on generic code. TastyTruffle uses TASTy's type information to reify types as first-class objects in the interpreter. These reified types enable TastyTruffle to dynamically select precise, box-free representations for generic values. TastyTruffle also uses TASTy's type information to determine which methods and classes are generic; it uses this information to specialize generic code on demand. The specialized code is much less polymorphic, which makes it significantly easier for the Graal just-in-time compiler (JIT) to reliably generate optimized generic code.

Through empirical evaluation, this thesis showed that TastyTruffle is competitive with a state-of-the-art JVM implementation configured to use the Graal compiler. In particular, while both implementations perform similarly on monomorphic workloads, when generic code is instantiated with multiple concrete types, TastyTruffle consistently outperforms the JVM. Performance on the JVM largely depends on the JIT compiler serendipitously choosing the right optimizations to perform; on large programs, this is infeasible. TastyTruffle performs these transformations directly in its ASTs, which reduces the guesswork required by the JIT. TastyTruffle also implicitly captures relationships among generic values by separating profiling information among specializations.

TastyTruffle demonstrates how generic type information can empower a language implementation to achieve high performance on generic code. By reifying types and specializing generic code, TastyTruffle enjoys much of the same performance benefits as a monomorphization scheme. Unlike monomorphization, TastyTruffle's specialization is speculative and performed at run time, which leaves further opportunities to improve performance in the future; for example, by speculatively changing object layouts, or by mixing the execution of specialized and unspecialized code.

References

[1] An Overview of TASTy, Aug. 2022. URL: https://docs.scala-lang.org/scala3/guides/tasty-overview.html.

[2] Arrays, Oct. 2022. URL: https://docs.scala-lang.org/overviews/collections-2.13/arrays.html.

[3] A ECMAScript 2022 compliant JavaScript implementation built on GraalVM, Mar. 2023. URL: https://github.com/oracle/graaljs.

[4] Project Valhalla, Jan. 2023. URL: https://openjdk.org/projects/valhalla/.

[5] Templates, Mar. 2023. URL: https://en.cppreference.com/w/cpp/language/templates.

[6] Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in Python programs. In *Proceedings of the 11th Symposium on Dynamic Languages*, pages 114–128, 2015.

[7] Danilo Ansaloni. Static Object Model, Oct. 2022. URL: https://docs.oracle.com/en/graalvm/enterprise/22/docs/graalvm-as-a-platform/language-implementation-framework/StaticObjectModel/.

[8] Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten Rust's belt: shrinking embedded Rust binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 121–132, 2022.

[9] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 183–200, New York, NY, USA, 1998. Association for Computing Machinery. URL: https://doi.org/10.1145/286936.286957, doi:10.1145/286936.286957.

[10] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, oct 1998. URL: https://doi.org/10.1145/286942.286957, doi:10.1145/286942.286957.

[11] Robert Cartwright and Guy L. Steele. Compatible Genericity with Run-Time Types for the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, page 201–215, New York, NY, USA, 1998. Association for Computing Machinery. URL: https://doi.org/10.1145/286936.286958, doi:10.1145/286936.286958.

[12] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape analysis for Java. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999*, pages 1–19. ACM, 1999. URL: https://doi.org/10.1145/320384.320386, doi:10.1145/320384.320386.

[13] Cliff Click and Keith D Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, 1995.

[14] Cliff Click and Michael Paleczny. A Simple Graph-Based Intermediate Representation. In Michael D. Ernst, editor, *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, pages 35–49. ACM, 1995. URL: https://doi.org/10.1145/202529.202534, doi:10.1145/202529.202534.

[15] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional Polymorphism in Type-erasure Semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, page 301–312, New York, NY, USA, 1998. Association for Computing Machinery. URL: https://doi.org/10.1145/289423.289459, doi:10.1145/289423.289459.

[16] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[17] Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 341–360. ACM, 2010. URL: https://doi.org/10.1145/1869459.1869489, doi:10.1145/1869459.1869489.

[18] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[19] Sébastien Jean R Doeraene. Cross-platform language design. Technical report, EPFL, 2018.

[20] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47, 2009.

[21] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, pages 1–10, 2013.

[22] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

[23] Dan Graur, Rodrigo Bruno, and Gustavo Alonso. Specializing generic java data structures. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 45–53, 2021.

[24] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 130–141, 1995.

[25] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European conference on object-oriented programming*, pages 21–38. Springer, 1991.

[26] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*, pages 123–132, 2014.

[27] Andrew Kennedy and Don Syme. Design and Implementation of Generics for the .NET Common Language Runtime. *SIGPLAN Not.*, 36(5):1–12, may 2001. URL: https://doi.org/10.1145/381694.378797, doi:10.1145/381694.378797.

[28] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 1–12, New York, NY, USA, 2001. Association for Computing Machinery. URL: https://doi.org/10.1145/378795.378797, doi:10.1145/378795.378797.

[29] Sebastian Kloibhofer, Lukas Makor, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. Control flow duplication for columnar arrays in a dynamic compiler. *arXiv preprint arXiv:2302.10098*, 2023.

[30] Filip Křikava, Heather Miller, and Jan Vitek. Scala implicits are everywhere: a large-scale study of the use of Scala implicits in the wild. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[31] Oracle Labs. Java on Truffle: Introducing a New Way to Run Java, 2023. URL: https://www.graalvm.org/java-on-truffle/.

[32] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 126–137, 2018.

[33] Lukas Makor, Sebastian Kloibhofer, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. Automatic Array Transformation to Columnar Storage at Run Time. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, pages 16–28, 2022.

[34] Ronald Morrison, Alan Dearle, Richard C. H. Connor, and Alfred L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(3):342–371, 1991.

[35] Frank Mueller and David B Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 56–66, 1995.

[36] Michael Paleczny, Christopher Vick, and Cliff Click. The java HotSpot™ server compiler. In *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*, 2001.

[37] Dmytro Petrashko. Design and implementation of an optimizing type-centric compiler for a high-level language. Technical report, EPFL, 2017.

[38] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, pages 29–40, 2017.

[39] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École polytechnique fédérale de Lausanne, 2005.

[40] Chris Seaton. *Specialising dynamic techniques for implementing the Ruby Programming Language*. The University of Manchester (United Kingdom), 2015.

[41] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala*, pages 1–8, 2013.

[42] Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala*, pages 1–8, 2013.

[43] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 165–174, 2014.

[44] Christopher Strachey. Fundamental concepts in programming languages. *Higher-order and symbolic computation*, 13(1):11–49, 2000.

[45] Vlad Ureche, Cristian Talau, and Martin Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 73–92, 2013.

[46] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 133–144, 2014.

[47] 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*, pages 662–676, 2017.

[48] 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*, pages 187–204, 2013.

[49] 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*, pages 73–82, 2012.

[50] James You. Specializing Scala with Truffle. Master's thesis, University of Waterloo, 2022.

APPENDICES

Appendix A

Generic `swap` AST

This appendix presents the generic AST for the `swap` method used in Chapters 4 and 5. The AST, depicted in Figure A.1, contains several generic nodes that dynamically change their behaviour based on type arguments. Chapter 5 describes a technique to statically specialize this AST to a simpler non-generic AST (Figure 5.9).

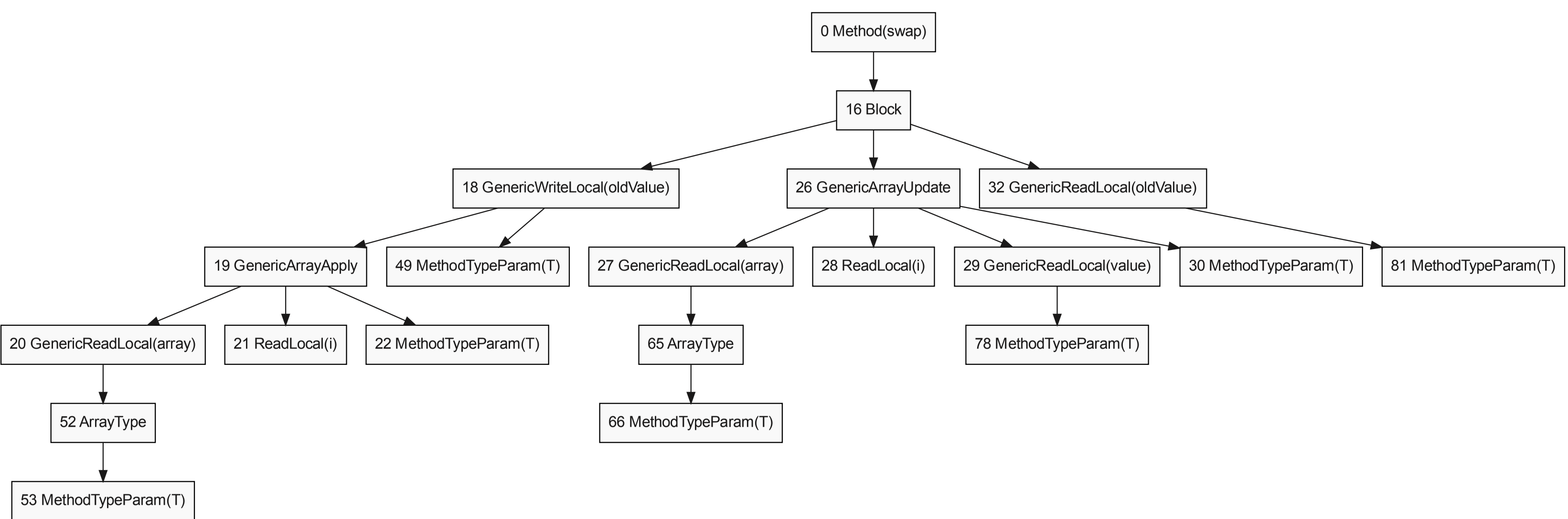


Figure A.1: Generic AST for `swap` (from Figure 5.1).