TASTyTruffle: Just-in-Time Specialization of Parametric Polymorphism

MATT D'SOUZA, University of Waterloo, Canada
JAMES YOU, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada
ALEKSANDAR PROKOPEC, Oracle Labs, Switzerland

Parametric polymorphism allows programmers to express algorithms independently of the types of values that they operate on. The approach used to implement parametric polymorphism can have important performance implications. One popular approach, erasure, uses a uniform representation for generic data, which entails primitive boxing and other indirections that harm performance. Erasure destroys type information that could be used by language implementations to optimize generic code.

We present TASTyTruffle, an implementation for a subset of the Scala programming language. Instead of JVM bytecode, TASTyTruffle interprets Scala's TASTy intermediate representation, a typed representation wherein generic types are not erased. TASTy's precise type information empowers TASTyTruffle to implement generic code more effectively. In particular, it allows TASTyTruffle to reify types as run-time objects that can be passed around. Using reified types, TASTyTruffle supports heterogeneous box-free representations for generic values. TASTyTruffle also uses reified types to specialize generic code, producing monomorphic copies of generic code that can be easily and reliably optimized by its just-in-time (JIT) compiler.

Empirically, TASTyTruffle is competitive with standard JVM implementations on a small set of benchmark programs; when generic code is used with multiple types, TASTyTruffle consistently outperforms the JVM. The precise type information in TASTy enables TASTyTruffle to find additional optimization opportunities that could not be uncovered with erased JVM bytecode.

CCS Concepts: • **Software and its engineering** → **Polymorphism**; **Dynamic compilers**.

Additional Key Words and Phrases: parametric polymorphism, specialization, reified types, just-in-time compiler, Truffle, Scala

ACM Reference Format:
Matt D'Souza, James You, Ondřej Lhoták, and Aleksandar Prokopec. 2023. TASTyTruffle: Just-in-Time Specialization of Parametric Polymorphism. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 277 (October 2023), 28 pages. https://doi.org/10.1145/3622853

1 INTRODUCTION

Parametric polymorphism, commonly called *generics*, is a powerful abstraction technique for expressing algorithms and data structures independently of the types of values they operate on.

Consider a `swap` function that replaces an array element with a new value and returns the old value. Such a function can be generic over the element type, since the element type does not affect the description of the algorithm. Below is a Scala implementation of such a `swap` method.

Authors' addresses: Matt D'Souza, matt.dsouza@gmail.com, University of Waterloo, Canada; James You, j35you@uwaterloo.ca, University of Waterloo, Canada; Ondřej Lhoták, olhotak@uwaterloo.ca, University of Waterloo, Canada; Aleksandar Prokopec, aleksandar.prokopec@gmail.com, Oracle Labs, Switzerland.

2475-1421/2023/10-ART277
https://doi.org/10.1145/3622853

```
def swap[T](arr: Array[T], i: Int, v: T): T = {
  val result: T = arr(i)
  arr(i) = v
  result
}
```

Note the use of the type parameter `T` to indicate the types of generic values.

Parametric polymorphism enables code reuse and improves the maintainability of software projects, but it poses a challenge for the language implementation: how should generic values be represented at run time? Several implementation approaches exist, each with inherent strengths and limitations, but most approaches fall broadly into one of two categories.

The first approach, *monomorphization* (heterogeneous translation), uses a different data representation for each set of type arguments (an *instantiation*). For each instantiation, a specialized copy of the code that operates on those representations is created. Below are some monomorphized copies of the `swap` method.

```
def swap$int(arr: Array[Int], i: Int,
             v: Int): Int = {
  val result: Int = arr(i)
  arr(i) = v
  result
}
```

```
def swap$string(arr: Array[String], i: Int,
                v: String): String = {
  val result: String = arr(i)
  arr(i) = v
  result
}
```

Monomorphization is used in the implementations of C++ templates [Stroustrup 1999] and Rust [Klabnik and Nichols 2023]. It 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.

The second approach, *erasure* (homogeneous translation), uses a single uniform data representation. This approach was pioneered by Java [Bracha et al. 1998]. In an erasure scheme, each generic method is translated to a single method that treats generic data uniformly. The type of this uniform representation is often some universal supertype; for Java, it is `Object`. The code below depicts a typical erased representation for `swap`.

```
def swap(arr: Array[Object], i: Int,
         v: Object): Object = {
  val result: Object = arr(i)
  arr(i) = v
  result
}
```

The erased representation replaces `T` with the universal supertype `Object`. Erasure produces less code than monomorphization, but forces generic data to conform to a single representation. When a primitive argument is passed for `v`, it must undergo an expensive boxing operation in order to fit the uniform `Object` representation. Moreover, since `arr` is erased to `Array[Object]`, an array of primitive values can simply never be passed for `arr`. Erasure also loses the information that the method is generic and the requirement that the array element type, the type of `v`, and the method return type must be consistent — information that could be used by a just-in-time (JIT) compiler to produce more efficient code at run time.

Scala is a statically-typed programming language that takes its own erasure approach. Notably, and unlike Java, Scala allows arrays of primitives like `Array[Int]` to be used where a generic array type `Array[T]` is expected. Thus, Scala's generic array has a non-uniform representation — it can be an array of reference types or an array of any of Scala's eight primitive types. In order to support this non-uniform representation, every generic array access gets implicitly rewritten to use run-time accessor methods. The code below depicts the `swap` method under Scala's erasure scheme. Notice that array operations get rewritten to `apply` and `update` calls. Also, since the common supertype for the different representations of `Array[T]` is `Object`, Scala's erased representation loses the information that a generic array is an `Array` at all.

```
def swap(arr: Object, i: Int,
         v: Object): Object = {
  val result: Object = apply(arr, i)
  update(arr, i, v)
  return result
}
```

These array accessor methods are expensive, requiring a cascading series of type tests to identify the underlying type of a generic array. Consider a snippet of the `apply` method below. Since these methods use `Object` types in their signatures, primitive values *remain* boxed when they enter and leave the accessors. For example, since `apply` returns an `Object`, when the underlying `array` is an `Array[Int]`, the `Int` it returns is implicitly boxed up.

```
def apply(array: Object, i: Int): Object =
  if (array.isInstanceOf[Array[Int]])
    return array.asInstanceOf[Array[Int]](i)
  else if (array.isInstanceOf[Array[Double]])
    return array.asInstanceOf[Array[Double]](i)
  else ...
```

Erasure imposes significant performance overheads at run time. Forcing a uniform representation introduces expensive extra memory indirection, which also inhibits classic compiler optimizations. Just-in-time (JIT) compilation can mitigate some of the performance overhead, for instance, by eliding box-unbox sequences and redundant type checks, but these transformations are driven by imperfect heuristics that can fail.

Dynamic type profiling can help recover the type information lost by erasure, but when type profiles are polymorphic (which is often the entire point of generics), they are less useful, especially in recovering relationships between different generic values. Consider the following Scala method with two `Array[T]` parameters.

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

The Scala source guarantees that both arrays have the same element type for any given invocation. If the method is called separately with integer and double arrays, type profiles will record that it was called with arguments of types `Array[Int]` and `Array[Double]`, but the correlation that `src` is `Array[Int]` *if and only if* `dst` is `Array[Int]` is lost. Without such correlations, generating efficient code becomes much more difficult.

Proposed solution. Instead of monomorphizing code at compile time, or erasing generic code to a uniform representation, this paper investigates how generic type information can be retained until execution and then exploited by a just-in-time compiler to generate efficient code. Since the input program is not monomorphized, this approach avoids the code bloat penalty associated with monomorphization. Further, since precise type information is available at run time, the implementation can use precise data representations that avoid the performance penalties associated with erasure.

To validate our solution, we implemented TASTyTruffle, a new interpreter for a subset of Scala that reifies generic types and uses them to generate specialized generic code. Scala is typically compiled to JVM bytecode, where generic type information is erased. Instead, TASTyTruffle interprets TASTy, Scala 3's compact binary representation of Scala programs. TASTy retains all of the type information from the source program, including generic types, which makes it a valuable alternative to JVM bytecode as an interpretation target.

This paper investigates the potential of the precise generic type information in TASTy for generating efficient code in a JIT compiler. We make the following contributions:

- A high-level description of TASTyTruffle, an interpreter for TASTy written using Truffle [Würthinger et al. 2013, 2012] (Section 2). Truffle is a framework built on top of the Graal VM [Duboscq et al. 2013] that reduces the effort required to create high-performance language implementations by partially evaluating an interpreter into a JIT compiler. Graal is especially effective at performing speculative optimizations, which make it possible to achieve good performance even with highly polymorphic code.

- A strategy to *reify* generic types using TASTy and use those reified types to implement generic methods and classes (Section 3). TASTyTruffle passes reified types into generic code, which dynamically switches over the types to implement accesses to specialized data representations. Having precise knowledge about the types used in a generic context allows TASTyTruffle to avoid the indirection of boxing and gives the JIT compiler more opportunities to optimize generic code.
- A technique to *specialize* generic code in TASTyTruffle to handle specific representations (Section 4). In the reified interpreter, generic code exhibits performance inconsistencies on polymorphic workloads. TASTyTruffle creates specialized copies of generic code that are monomorphic and more readily optimized by TASTyTruffle's JIT compiler. Unlike monomorphization, TASTyTruffle's specialization is dynamic and performed on-demand.
- An empirical evaluation of TASTyTruffle on seven small benchmark programs (Section 5). Compared to a standard HotSpot implementation, TASTyTruffle achieves higher average throughput across all benchmarks (3.32× geometric mean). Compared to Graal, which uses the same compiler (and is thus a more fair comparison), TASTyTruffle exhibits higher throughput for polymorphic workloads (1.94× geometric mean) and performs comparably on monomorphic workloads (1.09× geometric mean). We investigate the compiler IR for the benchmarks and explore how reified types in TASTyTruffle enable the compiler to more effectively optimize the benchmarks. We also discuss the warm-up performance and memory usage of our approach.

2 SYSTEM OVERVIEW

A challenge with implementing languages efficiently is that they can be highly dynamic. An interpreter may need to support a range of expensive operations, but if only some specific code paths are taken during execution, it is more efficient to only handle the cases observed. The Truffle framework [Würthinger et al. 2013] aims to make high-performance language implementations possible with "modest effort". The Truffle ecosystem comprises several components, including (but not limited to) a domain-specific language (DSL) for writing interpreters [Humer et al. 2014], a set of general-purpose libraries, and a custom front-end to the Graal compiler, which plays an important role in Truffle's performance.

2.1 The Truffle Ecosystem

Truffle interpreters take the form of abstract syntax trees (ASTs) written in Java. AST interpreters are relatively intuitive to implement, but ASTs can be difficult to optimize due to their indirection. Truffle compiles hot ASTs using Graal, using partial evaluation [Würthinger et al. 2017] to remove these indirections (and perform other optimizations), which allows Graal to generate highly efficient code. Truffle interpreters can change their behaviour depending on the run-time characteristics of the interpreted program, for example, by rewriting an AST node based on specific types of inputs.

2.1.1 Partial Evaluation. Normally, AST interpreters are difficult to compile, since they consist of highly-polymorphic `execute` call sites. Truffle uses partial evaluation (PE) to make AST interpreters more amenable to compilation [Futamura 1999; Würthinger et al. 2017]. 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 and devirtualizes as many calls (to `execute` and other methods) as it can. It's possible for such assumptions to be wrong, in which case any partially-evaluated code becomes invalid. When control reaches an invalid code path, the code is deoptimized and resumes execution in the Truffle interpreter.

2.1.2 Self-Optimizing ASTs. Truffle nodes are *self-optimizing* [Würthinger et al. 2012]: a node can have multiple *specializations*[1], each defining different semantics. By profiling the behaviour 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.

2.1.3 Truffle's Object Model. Interpreters often need to support object types (structures, class instances, etc.). An object is logically a collection of named properties. Truffle offers two object models (or shapes): a *dynamic* model for objects whose properties (and their types) can change during execution [Wöß et al. 2014], and a newer *static* model for objects with a fixed set of properties [Ansaloni 2022]. The static object model is suitable for languages where objects have a statically known set of properties, such as C, Java, or Scala. Static shapes can be built from a parent shape. This is convenient for implementing inheritance — the resultant shape contains all of its parent's properties.

2.2 TASTyTruffle: A Truffle Interpreter for Scala

TASTyTruffle is an interpreter built using the Truffle framework [Würthinger et al. 2013]. The primary goal of TASTyTruffle is to demonstrate how language implementations can benefit from generic type information. Thus, rather than implementing the entire Scala language, it implements a core subset of it; namely, TASTyTruffle supports: primitives, classes with single inheritance, and singleton objects; if-statements, while loops, and early returns; and both direct and indirect method dispatch. More complicated features like pattern matching, multiple inheritance, and Java interoperation are outside of the scope of the project.

Scala is traditionally compiled to Java Virtual Machine (JVM) bytecode and executed on a JVM. TASTyTruffle instead interprets TASTy, a lossless representation of Scala code introduced in Scala 3.[2] Unlike JVM bytecode, wherein generic types from Scala source code are erased, TASTy preserves all generic type information, which allows us to reify type information in the interpreter.

2.2.1 Data Representations. An important design consideration is how to represent program data in TASTyTruffle. Scala's type system is similar to Java's, so we can represent many Scala values 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[]`.

```
class Shape {
  Map<Symbol, Field> fields;
  Map<MethodSignature, Method> methods;
  Map<MethodSignature, Symbol> vtable;
  Symbol parent;
  StaticShape staticShape;
}
```

Fig. 1. Data definition for Shape.

2.2.2 Shapes. What remains are class instances (i.e., objects) which contain fields and can have methods invoked on them. TASTyTruffle uses Truffle's static object model to implement Scala objects. The parser creates a static shape for each Scala class, mapping each field to a property of the static shape. Truffle uses these field properties to synthesize a JVM class with storage for each field. In the interpreter, each Scala object is an instance of this synthetic class.

TASTyTruffle uses its own Shape abstraction to represent all of the information about a Scala class (Figure 1). A Shape has tables for its declared fields and methods, a virtual method table, a symbolic reference to its parent and a static shape to create object instances.

[1] A Truffle specialization is conceptually different from a generic specialization.
[2] https://docs.scala-lang.org/scala3/guides/tasty-overview.html

```
class ReadLocal extends Node {
  @CompilationFinal Local local;

  Object execute(VirtualFrame frame) {
    return switch(local.getRepresentation()) {
      case BOOL -> frame.getBoolean(local.getIndex());
      case INT -> frame.getInt(local.getIndex());
      case LONG -> frame.getLong(local.getIndex());
      ... // other primitives
      default -> frame.getObject(local.getIndex());
    }
  }
}
```

Fig. 2. Source code for a ReadLocal node before partial evaluation.

Shape properties can be computed during parsing. The fields and methods declared inside the class are used to contruct the field and method tables. To construct the virtual method table, we use Scala compiler APIs to determine a class's declared and inherited methods. A symbolic reference to the parent class can also be obtained from the AST. 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.

2.2.3 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, we store the value's representation on the node itself, so it knows how to properly handle the value.

Locals. Local access nodes require both the frame index of a variable and its representation. For example, the `ReadLocal` node (Figure 2) first switches over the local's representation to determine how to access it; then, once the representation is known, it reads the local using the appropriate type and index.

This design interacts well with partial evaluation. The partial evaluator treats fields marked `@CompilationFinal` as compilation constants, so during compilation, the switch can be replaced with the branch to handle the appropriate representation. 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).[3] The branching and indirect calls are eliminated, and what remains is a single frame read. When Graal performs scalar replacement, whereby the `frame` object is elided and frame slots are replaced with local variables, the frame read simplifies to a regular local variable read.

```
class ReadLocal extends Node {
  @CompilationFinal Local local;

  Object execute(VirtualFrame frame) {
    return frame.getInt(2);
  }
}
```

Fig. 3. Source code for a ReadLocal node after partial evaluation (when local has type INT and index 2).

[3]Partial evaluation happens during bytecode parsing, but we present source code for the sake of illustration.

Fields. TASTyTruffle has field access nodes to implement direct field accesses.[4] These nodes work analogously to local accessors. Each node stores 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[]`. TASTyTruffle's array nodes store the array's component type and switch over it to implement each operation; when the nodes are compiled, partial evaluation replaces each switch with the branch that handles the statically-known component type.

3 USING REIFIED TYPES IN TASTYTRUFFLE

Recall the `swap` method from Section 1. When compiled to JVM bytecode, the type parameter `T` of `swap` is erased: both `array` and `value` are given type `Object`, and the bytecode operates on the values in a uniform way. Forcing generic values into a uniform representation introduces boxing overheads and impedes efficient compilation.

Unlike JVM bytecode, TASTy has complete type information, which can be used to reify types. By reifying types, we can pass type information into generic contexts at run time. Generic code can use reified types to dynamically support different data representations for generic values. For example, if `swap` takes a reified type for `T`, when `T` is `Int` it can store `value` without boxing; it can also determine that `array` is an `Array[Int]` and avoid expensive generic array type switches.

3.1 Reifying Types

To reify types in TASTyTruffle, we extend the AST with `TypeNodes`. `TypeNodes` model a subset of Scala's rich type system: class types (e.g., `Foo`) are modeled by `NamedType`; each primitive type is modeled by a respective primitive `TypeNode` (e.g., `IntType`); type parameters are modeled by `MethodTypeParam` and `ClassTypeParam`; and generic type applications (e.g., `List[Int]`) are modeled with `AppliedType`. The `AppliedType` node represents a generic type applied to concrete type arguments, which are themselves `TypeNodes` (i.e., `TypeNodes` can form trees).

A `TypeNode` is a Truffle tree that can be executed to obtain a `Shape` (Section 2.2.2). For any TASTyTruffle node that needs a `TypeNode`, we extract the type information from TASTy during parsing and convert it to a `TypeNode`. At run time, a node can evaluate its `TypeNode`(s) to compute reified types. To evaluate primitive and class type nodes, we simply look up their `Shapes` by name from a global table. Evaluating type parameters and type applications can depend on local context (e.g., type arguments passed to a generic method); their implementations are discussed in the subsequent sections.

3.2 Generic Methods

A generic method uses type parameters to model the types of its generic values. With reified types, a generic method can dynamically support heterogeneous representations for its generic values. We extend the TASTyTruffle AST in a few ways to make this possible.

3.2.1 Invoking Generic Methods. In TASTyTruffle, type arguments are reified and passed to generic methods at each call site. When the parser encounters a generic call, it supplies the call node with additional `TypeNodes` for its type arguments. These type arguments are evaluated and

[4]Non-private field accesses are *indirect* accesses proxied through accessor methods.

```
@NodeChild("typeNode")
class GenericReadLocal extends Node {
  final int index;
  @Specialization
  int readInt(VirtualFrame frame, IntShape shape) {
    return frame.getIntStatic(index + INT.ordinal());
  }
  @Specialization
  double readDouble(VirtualFrame frame, DoubleShape shape) {
    return frame.getDoubleStatic(index + DOUBLE.ordinal());
  }
  ... // other specializations
}
```

Fig. 4. Source code for a `GenericReadLocal` node.

passed during the call. The generic callee can read these type parameters using `MethodTypeParam` nodes, which load type parameters directly from the frame (just like regular parameters).

3.2.2 Generic Locals. When a local variable is generic, the way to access it from the frame changes dynamically. For example, if a local has type `T`, it may be accessed from a different location depending on the concrete value of `T` (e.g., in a primitive or object section).

TASTyTruffle has generic variants of its local accessors that use `TypeNodes` to dynamically determine how a generic local should be accessed. For instance, the `GenericReadLocal` node defines a different specialization for each data representation in TASTyTruffle (Figure 4). It evaluates its `typeNode` child to a `Shape`, which Truffle uses to dispatch to the appropriate strategy.

For example, if the `typeNode` evaluates to a `DoubleShape`, the generic local has type `double`, and Truffle will invoke the `readDouble` specialization. This specialization reads the local from the primitives section of the frame, where it is stored as a primitive `double` without boxing.

Dynamically changing the representation of locals has subtle consequences for JIT compilation. If the same frame slot is used for every representation of a generic local, the compiler models the local's type with the common `Object` supertype. Even though the interpreter design ensures that a single type is used consistently (e.g., a local is always `int` for a given invocation), the compiler cannot easily infer this, which can introduce boxing and run-time type checks. TASTyTruffle works around this problem by allocating a unique frame slot for each possible representation of a local. For example, index `n` may be used for reference types, `n+1` for `int`s, `n+2` for `double`s, and so on. Observe how each representation's ordinal (e.g., `INT.ordinal()`) is added to a base index to obtain a unique index. Using multiple slots can sometime allocate unnecessary frame space, but during compilation Graal can elide storage for representations that are not observed at run time.

3.2.3 Generic Arrays. Since TASTyTruffle supports multiple array representations, code that operates on generic arrays must dynamically support different representations. For example, to read from a generic array, we must first know whether it is an `int[]`, `double[]`, or some other array type. TASTyTruffle defines generic variants of its array accessors that model the array component type with a `TypeNode`. These accessors evaluate the `TypeNode` to decide how to cast and access the array.

3.3 Generic Classes

Like generic methods, generic classes model generic values using type parameters. Generic classes support generic locals and arrays using the same AST nodes as generic methods — however, now the TypeNodes stored on those nodes may be ClassTypeParams.

A generic class can also define generic fields. Since fields are stored in class instances, the representation of a generic class instance itself depends on the values of its type parameters. For example, if a generic class defines a field of generic type T, the representation of a class instance — in particular, the field layout — is different when T is Int or Long. The code that operates on generic class instances needs to account for the heterogeneous ways they can be represented.

3.3.1 Modeling Applied Generic Classes. To support generic classes, the interpreter needs a runtime representation for a generic class applied to concrete type arguments (an *applied generic class*). Recall that TASTyTruffle uses a Shape to represent non-generic classes. A Shape contains an object layout and tables for its fields and methods (Figure 1). To model a generic class, we simply extend Shape with an additional field mapping each type parameter to a concrete type argument (Figure 5). Each unique set of type arguments supplied to a generic class corresponds to a unique GenericShape with its own object layout. How GenericShapes are actually created will be discussed shortly.

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

Fig. 5. Data definition for GenericShape.

Like with method type parameters, class type parameters can be directly referenced in method bodies. The TASTyTruffle parser creates ClassTypeParams when parsing TypeNodes. Whereas MethodTypeParams are stored in the frame, ClassTypeParams are stored on the GenericShape of a generic class instance. To evaluate a ClassTypeParam, we evaluate the receiver object, obtain its GenericShape, and look up the parameter from its typeArgMap. Since the map lookup is expensive, and generic class methods will observe different GenericShapes for each instantiation of the class, ClassTypeParam uses a polymorphic inline cache indexed on the receiver's shape.

3.3.2 Instantiating a GenericShape*.* The interpreter needs a mechanism to actually apply a generic class to concrete type arguments. Since different type arguments impose different object layouts, and a class is not applied to concrete types until execution, TASTyTruffle must retain enough metadata about a generic class to dynamically instantiate specialized object layouts.

We define a GenericShapeTemplate for this purpose. When the parser encounters a generic class, it creates a GenericShapeTemplate that stores essentially the same information as a Shape. However, since each Field's representation and offset depends on the concrete type arguments, GenericShapeTemplate also stores TypeNodes for each field. If the field is generic, its TypeNode will contain ClassTypeParam(s).

Specializing TypeNode*s.* An important part of the application process is TASTyTruffle's TypeNode specialization. TASTyTruffle defines a TypeNodeSpecializer visitor that can traverse an AST and replace type parameters with a set of constant types; for example, a ClassTypeParam could be replaced by IntType. The visitor transforms subtrees, so ClassTypeParams nested inside AppliedTypes can also be replaced with constant types.

The application process. To apply a generic class to type arguments, we:

(1) First create a typeArgMap mapping each type parameter to its concrete type argument.
(2) Then, construct the Fields. For each field in the GenericShapeTemplate, transform its TypeNode into a non-generic TypeNode using the TypeNodeSpecializer. The resultant

```
class AppliedType extends TypeNode {
  final Symbol templateSymbol;
  @Children TypeNode[] typeArgs;

  @ExplodeLoop
  Object execute(VirtualFrame frame) {
    Shape[] concreteArgs = new Shape[typeArgs.length];
    for (int i = 0; i < typeArgs.length; i++) {
      concreteArgs[i] = typeArgs[i].execute(frame);
    }
    return Globals.lookupTemplate(templateSymbol).apply(concreteArgs);
  }
}
```

Fig. 6. Source code for an AppliedType node.

TypeNode evaluates to a fixed Shape; use these Shapes to create Fields with specialized representations.

(3) Use the Fields to synthesize a Truffle StaticShape.
(4) Create a new GenericShape with the computed Fields, StaticShape, and typeArgMap. Reuse the methods and parent information stored on the GenericShapeTemplate.

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

3.3.3 Modeling Generic Classes in the AST. The interpreter models generic applications (such as Map[Int, Double]) in the AST using the aforementioned AppliedType. AppliedType is a node in the TypeNode hierarchy that computes a GenericShape (Figure 6).

An AppliedType contains child TypeNodes for each of its type arguments. To evaluate an AppliedType, we evaluate all of the type arguments, look up the GenericShapeTemplate from a global table, and then invoke apply on it with the type arguments. The resultant GenericShape can be used to create new generic class instances.

If two AppliedType nodes apply the same GenericShapeTemplate to the same type arguments, they should produce the same GenericShape. Idempotence is important not only for correctness — two Map[Int, Double] instances should have the same Shape for type comparisons — but also for performance, since TASTyTruffle's inline caches are keyed on the receiver's Shape. On each GenericShapeTemplate, we store a cache that maps a list of Shapes (the concrete type arguments) to computed GenericShapes. When apply encounters new type arguments, it instantiates a new GenericShape and caches it; on subsequent invocations, it returns the cached GenericShape.

3.3.4 Generic Fields. For a given generic class, each instantiated GenericShape can have a different field layout. These fields may themselves have different representations. When a class is generic, the TASTyTruffle parser generates GenericFieldRead and GenericFieldWrite nodes for 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.

A generic field accessor node is evaluated in much the same way as a non-generic field accessor node. For example, to evaluate a generic field read, we compute the receiver, look up the Field from the receiver's Shape, and then switch over the field's representation to decide how to read the field.

[5]In Scala, indirect field accesses are proxied through accessor methods, which use these nodes in their implementations.

However, non-generic field accessors only ever observe a single `Field`, so they can compute it once and reuse it for every access. In contrast, a generic field accessor may need to use a different `Field` depending on the receiver's `Shape`. Performing a field lookup on every execution would be expensive and inhibit optimization, so generic field accessors use a polymorphic cache indexed on the receiver's `Shape` to remember previous computations.

3.3.5 Method Dispatch. Generic classes affect the method dispatch algorithm. Each `Shape` has a `vtable` mapping each method signature to the name of the `Shape` that implements it. Normally, calls look up the implementing `Shape` from a global `Shape` table, but a generic class does not have a `Shape`.[6] Instead, we modify the algorithm to search the receiver's `Shape` hierarchy for a `Shape` with the expected name. For example, if the `vtable` indicates that the generic class `Foo[T]` implements a method, we search the receiver's `Shape` hierarchy for a `Shape` named `Foo`, such as `Foo[Int]`, that will contain the method implementation.

3.3.6 Generic Parent Classes. To support generic parent classes in TASTyTruffle, the implementation is modified in a couple of ways:

- The existing `Shape` uses a symbolic name to refer to its `parent` `Shape`, but if the parent is generic, a symbolic name is not enough to determine the `Shape` — the type arguments matter. We change the `parent` field to be a `TypeNode` so that a precise parent `Shape` can be computed. It is possible for the parent `TypeNode` to reference type arguments from the child; for example, `Foo[T]` **`extends`** `Bar[T]`. When instantiating `Foo` with concrete types, we execute the parent `TypeNode` to get the parent `Shape`, but the `TypeNode` for `Bar[T]` cannot be directly executed without a receiver object to resolve `T`. Instead, when a generic child class is applied to concrete types, we first use the `TypeNodeSpecializer` to fill in any type parameters in the parent `TypeNode` so that it can be evaluated.
- The existing approach to evaluate a `ClassTypeParam` is to compute the receiver, obtain its `GenericShape`, and then look up the type argument from the `typeArgMap`. With generic parent classes, the receiver's base `Shape` may not contain the type parameter — the parameter can be defined elsewhere in the type hierarchy. We change `ClassTypeParam` to search through the `Shape` hierarchy for a `GenericShape` defining the type parameter. Since this search can be expensive, we add an inline cache indexed on the receiver's `Shape`.

3.4 Compilation

Reifying types means extra information must be computed and passed around the interpreter. Dynamically supporting heterogeneous representations also leads to extra run-time type checks and cache lookups. TASTyTruffle's implementation of reified types is carefully designed to avoid these overheads (as much as possible) in compiled code.

Since `TypeNodes` are Truffle trees, partial evaluation can determine when a `TypeNode` returns a constant `Shape`. It can elide the code required to evaluate the `TypeNode` and replace it with a constant `Shape`. These constant `Shapes` can be used to drive further optimizations, for example, by removing unnecessary run-time type tests altogether.

Some of the generic AST nodes define different Truffle specializations for each data representation. For example, `GenericReadLocal` evaluates a `TypeNode` to determine what representation to use when reading a local. By virtue of the Truffle DSL, the compiled code for these nodes will only support the representations actually seen at run time. For instance, if `GenericReadLocal` only encounters `Int` and `Double` locals at run time, its compiled code will only handle those two

[6] A generic class has potentially many instantiated `GenericShapes`, but no `Shape` representing the uninstantiated generic class.

representations. The resultant code is compact and can give the compiler more precise type information for optimizations.

The `AppliedType` node is also carefully written with partial evaluation in mind. Executing an `AppliedType` requires several loops to evaluate its type arguments and compare the result against each entry in the `GenericShapeTemplate`'s cache. Loops are troublesome for compiler optimizations, since the program state after each loop is difficult to determine. Through careful use of Truffle's compiler directives (such as `@CompilationFinal` and `@ExplodeLoop`), we ensure that all of these loops get unrolled. In the best case, where every type argument is a constant, partial evaluation can completely replace the type application code with the constant `Shape`.

4 SPECIALIZING GENERIC CODE

Experimentally, the implementation of generics described in the previous section can exhibit inconsistent performance. This section explains the performance problem and extends the scheme with specialization to achieve more reliable performance.

4.1 Motivation

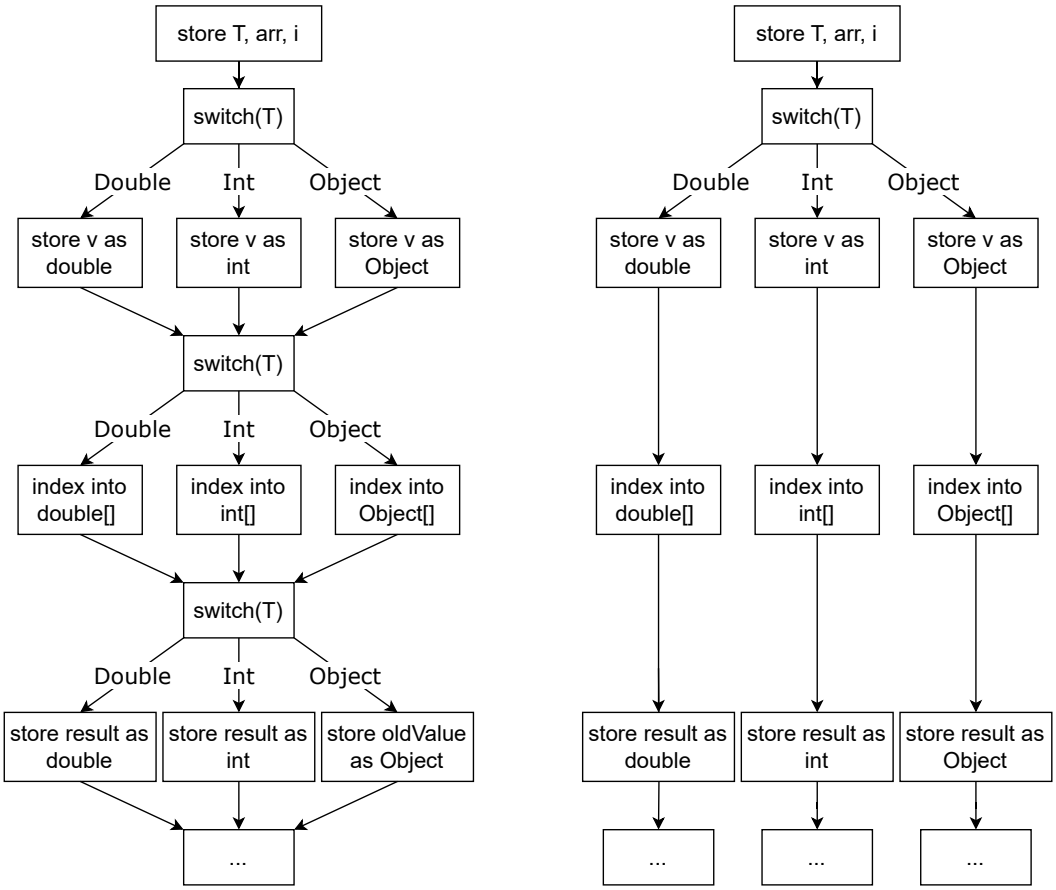
During interpretation, a variety of different data representations can flow through a generic AST. The AST dynamically changes its behaviour for each representation, but the type profiles and specialization states of the nodes are shared, which can lead to megamorphic code that is poorly optimized by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

Consider again the `swap` method from Section 1. Suppose that during interpretation, the program invokes `swap` with `T` being `Double`, `Int`, and a reference type. Each generic node in the AST will specialize itself to handle each representation. After specialization, each node effectively performs a switch over the type parameter `T` to decide how to execute. When Graal compiles `swap`, the resultant control flow graph looks something like Figure 7a. Control splits at each generic node and then merges after the operation. These frequent control flow splits inhibit optimization, since Graal cannot infer a precise state after each merge.

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 will eliminate 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 performs poorly.

The control flow splits are especially unfortunate because they switch over the same type parameters. When `swap` is invoked with `T` being `Int`, the second branch in Figure 7a 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 is the idea behind the *replication* or *tail duplication* [Mueller and Whalley 1995] 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`, rather than merge control after the first `switch`, the compiler could duplicate the remainder of the graph (starting from the second `switch`) 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 7b. 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.



(a) Before optimization

(b) After optimization

Fig. 7. Part of the control flow graph for swap.

The motivation for TASTyTruffle's specialization is to manually perform a tail-duplication-like transformation at the AST level. Graal supports tail duplication, but deciding when to perform it is an imperfect process driven by heuristics; improving Graal's tail duplication decisions is an active area of research [Leopoldseder et al. 2018]. 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.

4.2 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. 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 `swap` method once again. 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 `v` 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 simply replaces a generic method's body with a `TypeSwitch` node. The `TypeSwitch` node maintains a mapping from each set of type arguments to a unique specialization that gets invoked for those type arguments. When it encounters a new set of type arguments, the `TypeSwitch` node creates a fresh copy of the AST that will self-optimize itself for that specific set of type arguments. This approach to specialization is purely dynamic. We do not perform any static transformations on the generic AST to convert it to a monomorphic form; each copy of the AST simply self-optimizes itself into a monomorphic state.

As with `AppliedType`, `TypeSwitch` is carefully written to work well with partial evaluation. When some or all of the type parameters are PE-constant, the type switch can be simplified or even removed, and the specialized ASTs can be inlined directly into the compiled code.

4.3 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.
- TASTyTruffle creates a unique `GenericShape` for each unique set of type arguments supplied to a generic class (Section 3.3).

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, as was done in Section 3.3). 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 will only specialize themselves over these type arguments, and the resultant code will be monomorphic.

Unlike method specialization, methods specialized over class type parameters do not require `TypeSwitch` nodes. In effect, TASTyTruffle performs the type switching at class application time: the generic ASTs that should be executed are predetermined as soon as a `GenericShape` is resolved.

This specialization scheme works even for methods that have both class and method type parameters. Such a method will first be duplicated when the generic class is instantiated; then, when it is invoked with concrete type arguments, it will be duplicated again within the `TypeSwitch`. The net effect is that TASTyTruffle uses a unique AST for each unique combination of class and method type parameters, and so these ASTs will self-optimize into monomorphic states.

5 EVALUATING TASTYTRUFFLE

In this section, we evaluate the implementation of TASTyTruffle on a series of small Scala programs that use generics. These are programs of up to a few hundred lines of code; they are not full-sized Scala programs. The motivation for evaluating with these benchmarks is to validate the implementation and explore the interactions between TASTyTruffle and the Graal JIT. Many of the benchmarks are based on existing Scala standard library code that makes extensive use of generics, so efficiently implementing the generic idioms used in these benchmarks is a first step toward efficiently implementing more complex generic programs.

Table 1. Table of benchmarks.

Benchmark	Description	Input size
ArrayCopy	Copies the contents of one `Array[T]` to another.	1,000,000
Checksum	Computes a checksum of an `Array[T]`, invoking the **##** operator to hash each element.	1,000,000
InsertionSort	Performs insertion sort over an `Array[T]` using an `Ordering[T]` type class.	10,000
QuickSort	Performs quicksort over an `Array[T]` using an `Ordering[T]` type class.	20,000
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.	100,000
ArrayDeque	Defines a generic `ArrayDeque[T]` that mirrors the standard library. Repeatedly appends elements to a dynamically-resizable buffer.	100,000
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.	10,000

5.1 Benchmarks

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.

Note: TASTyTruffle can *always* allocate unboxed generic arrays (e.g., `int[]`) because it has type information, whereas JVM implementations require an extra `ClassTag` object to supply the component type at run time. Since our main goal in this evaluation is to compare how each configuration compiles code, it would be an unfair comparison to have one configuration use unboxed generic arrays and for another to use boxed arrays. Thus, the benchmarks are written with `ClassTag`s so that the JVM configurations also allocate unboxed generic arrays.

The full list of benchmarks is in Table 1. There are seven benchmarks loosely ordered by increasing complexity. Later benchmarks generally use a superset of the generic idioms used by earlier ones (e.g., in addition to type classes, StdDev also uses higher-order functions). The input sizes were selected (through trial and error) so that benchmark invocations would run long enough to avoid measurement error (e.g., from the benchmarking infrastructure) but not so long that few invocations would complete during a measurement iteration.

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.

5.2 Setup

The benchmarks are run in four different configurations:

HotSpot (H): The Scala benchmarks are compiled to JVM bytecode using the standard Scala compiler and executed by the HotSpot JVM (the typical execution environment for Scala programs).

Graal (G): This configuration is like H, but a Graal-equipped JVM is used instead.

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

Specialized TASTyTruffle (T_S): This configuration is like T_U, but specialization is enabled.

All benchmarks are run on a Ubuntu 22.04.2 system with four 16-core AMD Opteron 6380 processors and 512 GiB of memory. The HotSpot configuration uses OpenJDK 17.0.4; all other configurations use GraalVM Enterprise Edition 22.2.0 (which is built off of OpenJDK 17.0.4). The heap size is fixed to 8 GiB in all cases.

The benchmarks are executed using the Java Microbenchmark Harness (JMH)[7]. 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.

The primary goal of the evaluation is to assess how TASTyTruffle's design enables Graal's JIT to more effectively optimize generic code. Quantitative performance differences between G, T_U, and T_S generally indicate differences in the way programs are compiled; when they occur, we discuss these differences. HotSpot, being the "standard" implementation of Scala, is included in the evaluation as a performance baseline, but we do not analyze the code it produces.

5.3 Throughput

TASTyTruffle was designed to enable high-performance compiled code for generic workloads. To assess whether it achieves this goal, our primary focus in this evaluation is on the peak throughput of each benchmark after warm-up.

We run each benchmark for ten warmup iterations at ten seconds each to ensure that the benchmark method gets compiled by Graal. Prior to the actual evaluation, we performed dry runs, inspecting the compiler logs to ensure that each benchmark consistently reached a stable, warmed-up state during the warmup iterations. After warmup, we run each benchmark 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.

Figure 8 depicts the mean throughput for each benchmark. In the following subsections, we discuss some reasons for these results. Differences in peak throughput can occur for many reasons, and the reasons can be interrelated (and challenging to disentangle). Even *within* a particular configuration, the throughput can be highly variable due to the non-determinism of JIT compilation (we see this with QuickSort and StdDev, where G exhibits different steady-state throughputs in different forks). The goal in these sections is to identify features of the compiled code that likely play a role in the throughput results.

5.3.1 Polymorphic Workloads. We first consider the throughput results for the polymorphic workloads (the top row of Figure 8). In general, T_S achieves the highest throughput on the polymorphic workloads. Compared to H, T_S achieves a 4.42× speedup, G achieves a 2.27× speedup, and T_U

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

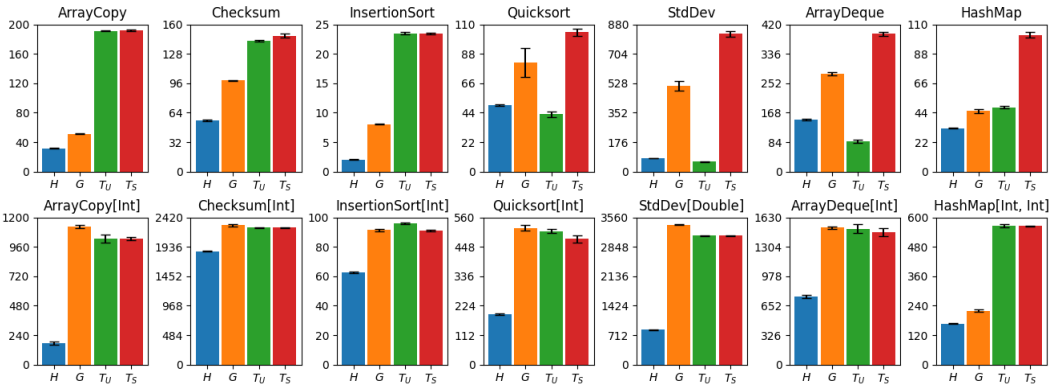


Fig. 8. Throughput of each benchmark (in operations per second) with 99.9% confidence intervals.

achieves a 1.92× speedup (geometric mean). We observe a few interesting features of the compiled code that likely contribute to the observed differences.

```
def copy[T](src: Array[T],
            dst: Array[T]) = {
  var idx = 0
  while (idx < src.length) {
    dst(idx) = src(idx)
    idx += 1
  }
}
```

Loop unswitching. Recall that T_S avoids polymorphic code by making a copy of the AST for different concrete generic types. Graal's *loop unswitching* transformation provides similar benefits for G and T_U. When a loop contains a conditional branch and the branch condition is loop-invariant, loop unswitching can extract the conditional branch outside of the loop, duplicating the loop for each branch. Within each copy of the loop, the outcome of the branch condition is statically known. Since a generic type check (e.g., checking whether an array is an `int[]`) is loop-invariant, and each benchmark executes some sort of loop, loop unswitching can effectively monomorphize polymorphic benchmark code in the same way as T_S's duplication.

Consider the ArrayCopy benchmark depicted above. It takes two arrays `src` and `dst` of type `Array[T]` and copies the contents of `src` into `dst`. Recall that on the JVM, Scala implements generic array accesses using runtime methods that switch over the run-time type of the array. The type of the arrays is invariant within the loop, so in theory the loop can be unswitched over the array type check. The code produced by G behaves like the pseudocode to the right. Graal uses type profiling on the `instanceof` check to determine that the arrays will likely be `int[]`, `double[]`, or `Object[]`. It successfully unswitches the case where `src` is an instance of `int[]`, but the two type tests for `double[]` and `Object[]` remain inside a loop together. When ArrayCopy is invoked with non-integer arrays, it suffers a performance hit because it must perform these type tests on each iteration of the loop.

```
idx = 0
if (src instanceof int[]) {
  while (idx < src.length) { ... }
} else {
  while (idx < src.length) {
    if (src instanceof Object[]) {
      ...
    }
    else {
      assert src instanceof double[]
      ...
    }
    idx += 1
  }
}
```

The compiler has many loop unswitching candidates, and deciding which conditional branches are worth unswitching is driven by heuristics, so the results can be unpredictable. On G, all of the benchmarks except InsertionSort and QuickSort produce IR graphs with generic array type

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

T_U matches T_S's throughput on the ArrayCopy, Checksum, and InsertionSort benchmarks. For each of these benchmarks, the type checks get unswitched from the loop, resulting in highly-optimized monomorphic code. Conversely, on the other benchmarks, wherein 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 why it achieves the highest throughput on the polymorphic workloads. Though T_U matches T_S on the simple benchmarks, Graal fails to make optimal loop unswitching decisions on the more complex benchmarks. Our takeaway is that performing code duplication at the AST level, in effect, makes it so that the compiler does not have to guess whether a type check is worth unswitching from a loop.

Explicit type associations. An interesting consequence of reifying types is that it makes type associations between different values more apparent to the compiler. Consider the two parameters to the ArrayCopy benchmark. Both parameters have type `Array[T]`, so 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`, so the fact that they have the same concrete type is lost. On G, this leads to code containing redundant type checks and impossible branches.

```
idx = 0
if (src instanceof int[]) {
  while (idx < src.length) {
    if (dst instanceof BoxedInt[]) {
      dst(idx) = Integer.valueOf(src(idx))
      ...
    } else {
      assert dst instanceof int[]
      ...
    }
  }
} else { ... }
```

We again consider the ArrayCopy benchmark. In an early phase of compilation, inside the copy of the loop dominated by the unswitched `src instanceof int[]` guard, Graal does not infer that `dst` must also be `int[]`. The compiler graph, depicted in pseudocode to the left, contains an impossible code path (introduced by type profiles) where `dst` is a `BoxedInt[]`. The code then attempts to store a boxed `Integer` into `dst`. In this case, a later optimization phase eliminates the nonsensical branch (presumably Graal detects the type-incompatibility), but in other cases the compiler may not be so lucky. Graal uses graph size as a heuristic for many optimizations including inlining and loop unswitching, so if Graal cannot remove the impossible branches, or can only do so in a later phase, the extra code could inflate the graph size and prevent Graal from performing 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 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 benchmarks take an `Ordering[T]` instance and invoke its `lt` ("less than") method to compare elements. On the polymorphic workloads, InsertionSort observes a different type class instance for each instantiation of `T` during interpretation: when `T` is `Int`, it observes an `Ordering[Int]`; when `T` is `Double`, it observes an `Ordering[Double]`, and so on. In such 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 type class instances observed during interpretation. Graal uses these profiles to speculatively inline methods from the observed type class instances. However, since types are erased on G, Graal does not understand the correlation between different `Ordering` instances and `T`, which leads to unnecessary type checks.

```
object IntOrdering
  extends Ordering[Int]
object DoubleOrdering
  extends Ordering[Double]
object BoxedIntOrdering
  extends Ordering[BoxedInt]

if (ord instanceof IntOrdering) {
  ...
  if (T == Int) {
    // read from int[]
  } else if (T == Double) {
    // read from double[]
  }
} else { ... }
```

Though types are reified on T_U, it also suffers from this problem. Consider the pseudocode to the right, which depicts part of the IR graph for InsertionSort on T_U. Unlike array representations, TASTyTruffle does not use reified type arguments to determine which type class instance to use. Instead, the Scala compiler injects type class instances at each call site during compilation. During execution, each type class method call (e.g., `ord.lt(...)`) profiles its receiver type; then, during compilation, Graal can use the profile to speculate over the type class instance.

On T_U, these profiles are shared between different type arguments, so the implicit association is lost. For a call to `ord.lt`, the type profile observes the three types for `Ordering` depicted in the code above. In the branch where `ord` is known to be `IntOrdering`, the compiler cannot infer that `T` must be `Int`, leading to more unnecessary type checks.

T_S does not exhibit this problem. Conveniently, 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 a convenient benefit 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 on G appears to be type class methods. For example, on QuickSort, the calls to `Ordering` methods do not always get inlined. G successfully inlines 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 StdDev 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. Though we have not confirmed the precise reason `fold` does not get inlined on T_U (inlining decisions can be somewhat opaque), we suspect code size to be at fault. It appears that generic methods on T_U can grow significantly in code size on polymorphic workloads: `fold`'s initial IR 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 copies of the AST, the entire IR graph for a generic method may contain too many copies of ASTs to be inlined. However, since the type parameters to a generic method call are

statically known to the inliner[8], 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.

5.3.2 Monomorphic Workloads. On all of the monomorphic workloads besides HashMap[Int,Int], the three Graal-based configurations perform comparably: T_U achieves a 2.54× speedup, T_S achieves a 2.49× speedup, and G achieves a 2.28× speedup over H (geometric mean).[9]

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 over G. 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 IR graphs produced on G.

Scala Runtime Methods. Unlike the other benchmarks, G is not able to perform monomorphization on 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 that switch over the type of the array. Since the JVM's type profiling is limited to virtual method 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.

```
idx = 0
if (src instanceof int[]) { ... }
else if (src instanceof float[]) { ... }
else if (src instanceof char[]) { ... }
else { ... }
```

For example, we ran a variant of the monomorphic ArrayCopy[Int] workload on a JVM where the generic array accessors were heavily used with `float`, `char`, and `boolean` arrays. The pseudocode to the left represents the compiled code produced for the `copy` method. Even though the ArrayCopy[Int] benchmark is only ever invoked with `int[]`, the polluted type profiles on the accessor methods lead to extra branches for `float[]`, `char[]`, and `boolean[]` (inside the **`else`**). 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 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.

[8]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 gets replaced by a constant type during specialization. For example, `fold[T]` will not get inlined into `computeStdDev[T]`, but into a particular specialization like `computeStdDev$Int` where `T` is known.

[9]Excluding HashMap[Int,Int] (which G optimizes poorly because of runtime method type pollution), G achieves a 2.52× speedup over H.

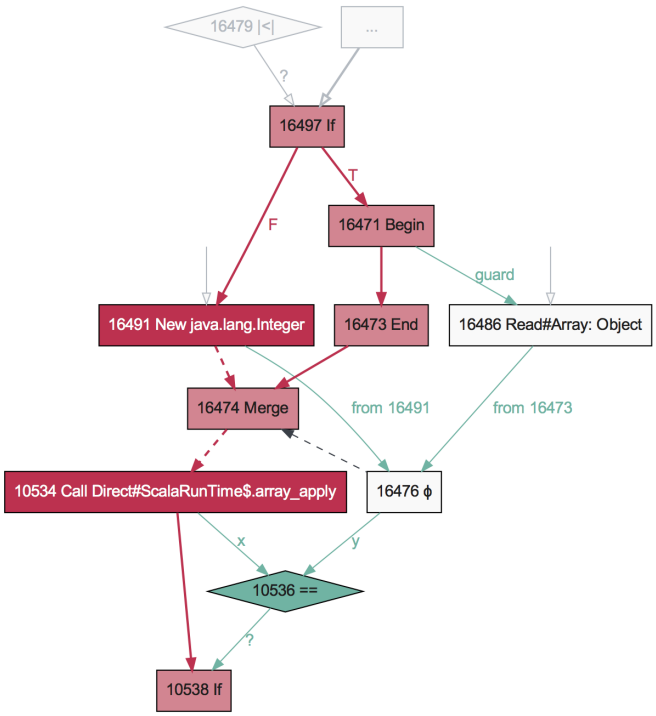


Fig. 9. Graal IR subgraph for the put method of HashMap[Int,Int] on *G*.

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[Int,Int]. In this section we discuss a specific compilation issue *G* exhibits on the HashMap[Int,Int] workload. 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 the `put` method listed to the right. The code performs linear probing to find an index to store a new table entry. Inside the loop, 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.

```
def put(key: T, value: U) = {
  var idx = hash(key)
  while (/* keys(idx) is occupied */) {
    if (keys(idx) == key) {
      // overwrite value
    }
    idx = idx + 1
  }
  ...
}
```

We present a small portion of the Graal IR for the `==` comparison in Figure 9. This graph follows the "sea-of-nodes" design [Click and Paleczny 1995]; in brief, the red edges represent control flow, and the teal edges represent data flow.[10] The

[10]For a primer on understanding Graal IR, we recommend https://chrisseaton.com/truffleruby/basic-graal-graphs/.

compiler has inlined `put` into a context where it knows that `keys` is an `int[]` and that `key` comes from an `int` value. Despite this knowledge, Graal does not 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 the runtime accessor method `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 `array_apply` was inlined, the compiler could have determined that both operands were `int`s and hence that the pointer comparison was unnecessary. It could have instead elided the boxing and performed 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 an existing 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, 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 `16486`). Since the resultant value (node `16476`) is not necessarily a fresh object, Graal cannot elide the pointer comparison, and so the `int` value that was 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. In TASTyTruffle, the array accessors are interpreter intrinsics, so it can always infer that `keys(idx)` will return a primitive `int`.

5.4 Warmup

Although Truffle interpreters achieve high peak throughput, they are known to struggle with start-up performance [Humer and Bebić 2022; Marr et al. 2022]. It takes time for the JIT to compile Truffle ASTs, and in the meantime, Truffle implementations are stuck executing relatively-slow AST interpreters. Though start-up performance is not a primary research goal for TASTyTruffle, we briefly examine its warm-up behaviour in this section.

We measured, across ten runs, the average throughput of the Checksum, HashMap, and QuickSort benchmarks as the programs warmed up. Figure 10 plots the average throughput over time for each benchmark. We captured the average throughput over 200 ms intervals; the 99% confidence interval for the throughput in each interval is depicted in a translucent colour. In general, the throughput is unstable at the beginning as the benchmarks warm up, and it tends to stabilize over time as the JIT compiles hot code paths.

Overall, *H* and *G* warm up much faster than the TASTyTruffle configurations. At $t = 0$, *H* and *G* usually have non-zero throughput, but the TASTyTruffle throughput is very close to zero. We also see that *H* and *G* ramp up earlier than TASTyTruffle; for example, on QuickSort *H* and *G* reach their peak performance by around 2s, whereas T_U and T_S take much longer.

These observations makes sense, because *H* and *G* execute JVM bytecode, the native interpretation target of the JVM. TASTyTruffle executes ASTs that are *themselves* implemented in JVM bytecode. The Truffle-based interpreters, executed on the JVM, are too slow to achieve much throughput, so the host code (e.g., the nodes and their `execute` methods) must generally be compiled first by the

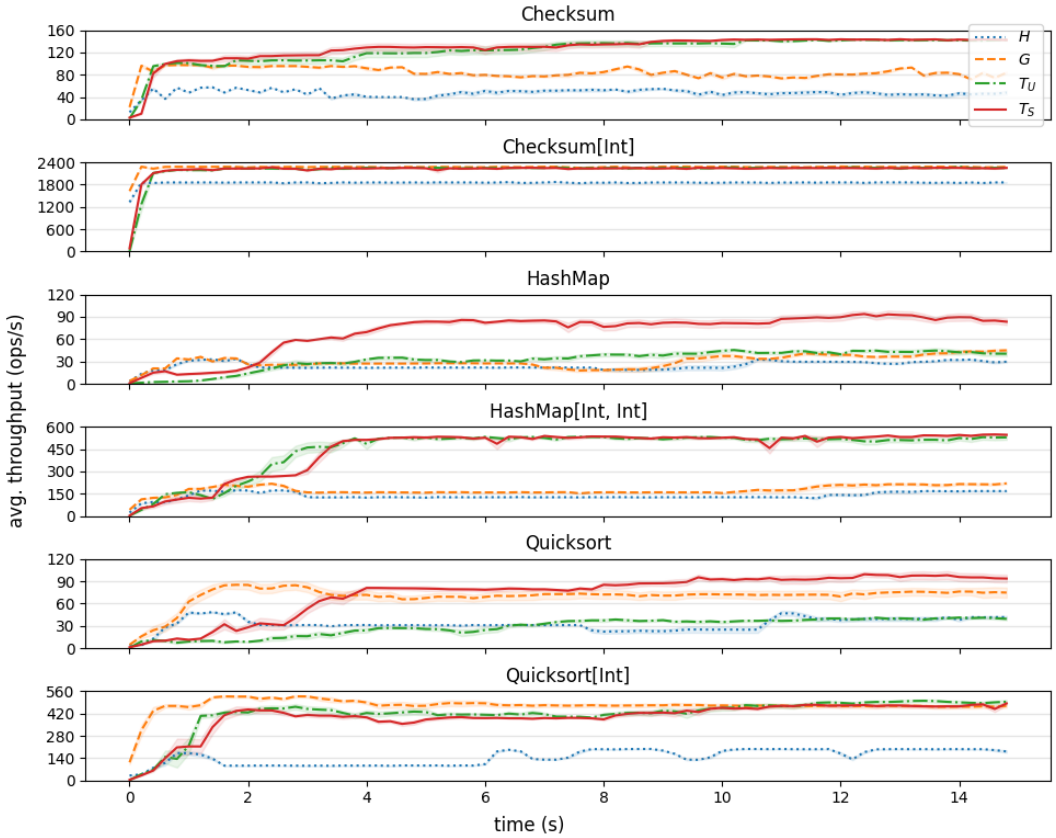


Fig. 10. Average throughput over time for Checksum, HashMap, and QuickSort on monomorphic and polymorphic workloads.

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 60 ops/s in the first few seconds, then later jumps up to around 90 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. The difference likely 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 cannot optimize first-tier compilations very much: the first-tier graphs show a loop that is polymorphic over the three array types. In contrast, on T_S each specialized AST is first-tier compiled separately, and each loop is monomorphic, so the compiler can perform more optimization.

Thus, despite the extra work required to duplicate ASTs, 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.

5.5 Memory

We performed a couple of experiments to assess the memory usage of TASTyTruffle.

5.5.1 Heap Size. By virtue of their representation, ASTs tend to consume more memory than other interpretation targets. For the Checksum, HashMap, and QuickSort benchmarks, we modified our benchmark scripts to dump the live (reachable) heap at the end of execution so we could measure this difference. Table 2 depicts the heap sizes across configurations. These measurements do not include Scala program data; the memory usage of Scala objects is discussed in the next section.

Across all workloads, the reachable heap is 3-4× larger on TASTyTruffle than the JVM configurations. Interestingly, only a small portion of the memory increase comes from ASTs. We programmatically compared the heap contents and found that the vast majority of extra memory usage comes from classes loaded by the TASTyTruffle configurations; TASTyTruffle loads almost five thousand more classes than the JVM configurations. Roughly half of these extra classes come from the TASTy parsing API (which itself hooks into the Scala compiler infrastructure), so there is room for improvement, but the remaining half appear to be classes needed by the Truffle runtime to invoke and compile Truffle ASTs. Thus, we believe some of this extra heap footprint is unavoidable with Truffle-based interpreters.

Recall that T_S creates duplicate ASTs to monomorphize generic code. We can see the impact this duplication has on the memory footprint by comparing the AST footprint between T_U and T_S. As expected, T_S uses more memory than T_U for its ASTs; this difference is greater on polymorphic workloads where more ASTs are duplicated. In the most extreme case (HashMap), the AST footprint increases by about 19%. In the future, there are strategies we can employ to reduce the memory footprint of T_S, for example by only duplicating the most frequently-used specializations.

5.5.2 Data Structure Size. We also examined the size of the data structures underlying the Scala objects in each implementation. TASTyTruffle implements Scala objects by dynamically synthesizing JVM classes containing the precise set of fields in each Scala class (Section 2.2.1). In other words, both TASTyTruffle and JVM-based object representations should occupy the same amount of memory as long as they use the same representations for those fields.

Recall from Section 5.1 that the benchmarks use `ClassTags` to allocate unboxed representations for generic arrays (to make throughput comparisons fair). In this case, the data structures in our benchmarks have exactly the same footprint. For example, across 5 executions, a `HashMap[Int, Int]` with 1,000,000 entries consistently occupied the same amount of memory on all configurations — 16.8 MB.

In reality, most data structures in the Scala standard library do not use `ClassTags`, so when they allocate generic arrays, they use `Object` arrays. We rewrote the HashMap benchmark to not use `ClassTags` and found that the same `HashMap[Int, Int]` instance with 1,000,000 entries occupied 31.1 MB on H and G — nearly double the size of TASTyTruffle's representation. The difference comes from the use of `Object` arrays that store boxed `Integers` (rather than primitive `ints`) for the key and value arrays. In effect, TASTyTruffle's reified types allow us to reduce the footprint of generic data structures *without* `ClassTags` (i.e., without making breaking changes to data structure APIs).

6 RELATED WORK

Parametric polymorphism, first distinguished from other forms of polymorphism by Strachey [2000], can be implemented in a variety of ways. Morrison et al. [1991] 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. This approach is more commonly known

Table 2. Heap size for Checksum, HashMap, and QuickSort benchmarks (in KB).

Benchmark	Conf.	Total (± SD)	AST (± SD)	# Classes (± SD)
Checksum	H	5160.52 (± 0.10)	0	2487.40 (± 1.20)
	G	5407.40 (± 0.30)	0	2551.80 (± 1.60)
	T_U	22257.77 (± 20.38)	305.40 (± 0.02)	7390.40 (± 1.62)
	T_S	22320.72 (± 1.67)	311.18 (± 0.02)	7413.80 (± 0.98)
Checksum[Int]	H	5168.43 (± 0.15)	0	2487.40 (± 1.20)
	G	5407.34 (± 0.17)	0	2551.80 (± 1.60)
	T_U	22190.46 (± 2.25)	294.42 (± 0.02)	7388.00 (± 1.79)
	T_S	22217.33 (± 5.16)	297.02 (± 0.02)	7410.00 (± 0.89)
HashMap	H	5868.21 (± 0.22)	0	2586.40 (± 1.20)
	G	6121.29 (± 0.88)	0	2648.80 (± 1.60)
	T_U	22787.60 (± 2.40)	379.76 (± 0.04)	7454.00 (± 2.76)
	T_S	22960.62 (± 2.01)	450.66	7472.20 (± 2.48)
HashMap[Int,Int]	H	5866.54 (± 0.03)	0	2584.40 (± 1.20)
	G	6119.48 (± 0.68)	0	2646.80 (± 1.60)
	T_U	22498.59 (± 18.05)	344.39	7431.40 (± 3.20)
	T_S	22567.28 (± 7.75)	369.40	7458.40 (± 4.63)
QuickSort	H	5836.74 (± 0.37)	0	2512.40 (± 1.20)
	G	6085.97 (± 5.23)	0	2574.80 (± 1.60)
	T_U	22648.55 (± 18.86)	377.65	7424.20 (± 2.23)
	T_S	22745.31 (± 16.70)	407.78 (± 0.02)	7447.20 (± 2.99)
QuickSort[Int]	H	5835.05 (± 0.22)	0	2510.40 (± 1.20)
	G	6074.69 (± 0.11)	0	2572.80 (± 1.60)
	T_U	22445.24 (± 13.94)	355.49 (± 0.02)	7408.20 (± 4.53)
	T_S	22491.78 (± 11.09)	366.87 (± 0.02)	7429.20 (± 2.71)

today as *type reification*, which is at the heart of TASTyTruffle's approach. TASTyTruffle's implementation of parametric polymorphism is similar to the approach taken by the .NET Common Language Runtime (CLR) [Kennedy and Syme 2001]. Unlike JVM bytecode, the CLR intermediate language preserves generic type information, so the CLR can reify generic type arguments at run time. TASTyTruffle and the CLR 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.

Dragos and Odersky [2009] extended the Scala compiler to support static specialization. Programmers can direct the compiler to automatically generate specializations for each primitive type, leading to more optimized generic code. Ureche et al. [2013] improved Scala's static specialization using "miniboxing", which allows sharing specializations among multiple types at the expense of additional run-time conversions. Since Truffle supports implicit conversions between data types, it is possible that TASTyTruffle could incorporate a miniboxing-like approach into its specialization scheme. 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 [Doeraene 2018; Petrashko 2017].

Early in Java's adoption of generics, Cartwright and Steele [1998] 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`. 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. Project Valhalla [Goetz 2014] is an ongoing project that aims to evolve the Java language to allow type parameters to range over both reference and primitive types, enabling run-time specialization of generic code in the future. Graur et al. [2021] developed JCS, a tool to specialize some classes from the Java Collections library. 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.

Schinz [2005] 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 type checks over generic types. The approach introduces significant overheads with respect to running time and memory consumption. 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.

Stadler et al. [2013] 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 to be especially effective. Prokopec et al. [2017] 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.

7 CONCLUSION

Many programming language implementations use erasure to support parametric polymorphism. While erasure reduces code size by avoiding code duplication, it is inherently limiting to performance because it introduces run-time indirection and destroys type information that could be used by the implementation to improve performance.

We presented TASTyTruffle, a Scala implementation that interprets TASTy IR instead of JVM bytecode. 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, and generate efficient code specialized for these representations. On our benchmarks, we found that TASTyTruffle achieves higher peak throughput than the traditional HotSpot JVM. TASTyTruffle is also competitive with a JVM equipped with the same Graal compiler, especially when generic code is instantiated with multiple concrete types.

TASTyTruffle demonstrates how precise 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.

DATA-AVAILABILITY STATEMENT

An archive containing the TASTyTruffle source and dependencies is available at [D'Souza et al. 2023].

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

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

Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. 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* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 183–200. https://doi.org/10.1145/286936.286957

Robert Cartwright and Guy L. Steele. 1998. 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* (Vancouver, British Columbia, Canada) *(OOPSLA '98)*. Association for Computing Machinery, New York, NY, USA, 201–215. https://doi.org/10.1145/286936.286958

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

Sébastien Jean R Doeraene. 2018. *Cross-Platform Language Design*. Technical Report. EPFL. https://doi.org/10.5075/epfl-thesis-8733

Iulian Dragos and Martin Odersky. 2009. 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*. 42–47. https://doi.org/10.1145/1565824.1565830

Matt D'Souza, James You, Ondřej Lhoták, and Aleksandar Prokopec. 2023. *Artifact for paper "TASTyTruffle: Just-in-time Specialization of Parametric Polymorphism"*. https://doi.org/10.5281/zenodo.8332577

Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*. 1–10. https://doi.org/10.1145/2542142.2542143

Yoshihiko Futamura. 1999. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation* 12, 4 (1999), 381–391. https://doi.org/10.1023/A:1010095604496

Brian Goetz. 2014. State of the Specialization. https://cr.openjdk.org/~briangoetz/valhalla/specialization.html

Dan Graur, Rodrigo Bruno, and Gustavo Alonso. 2021. Specializing generic Java data structures. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 45–53. https://doi.org/10.1145/3475738.3480718

Christian Humer and Nikola Bebić. 2022. Operation DSL: How We Learned to Stop Worrying and Love Bytecodes again. (2022). https://2022.ecoop.org/details/truffle-2022/3/Operation-DSL-How-We-Learned-to-Stop-Worrying-and-Love-Bytecodes-again Truffle/GraalVM Languages Workshop, part of European Conference on Object-Oriented Programming (ECOOP) 2022.

Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A domain-specific language for building self-optimizing AST interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. 123–132. https://doi.org/10.1145/2775053.2658776

Andrew Kennedy and Don Syme. 2001. Design and Implementation of Generics for the .NET Common Language Runtime. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 1–12. https://doi.org/10.1145/378795.378797

Steve Klabnik and Carol Nichols. 2023. *The Rust programming language*. No Starch Press.

David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 126–137. https://doi.org/10.1145/3168811

Stefan Marr, Octave Larose, Sophie Kaleba, and Chris Seaton. 2022. Truffle Interpreter Performance without the Holy Graal. (March 2022). https://kar.kent.ac.uk/93938/

Ronald Morrison, Alan Dearle, Richard C. H. Connor, and Alfred L. Brown. 1991. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 3 (1991), 342–371. https://doi.org/10.1145/117009.117017

Frank Mueller and David B Whalley. 1995. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. 56–66. https://doi.org/10.1145/223428.207116

Dmytro Petrashko. 2017. *Design and implementation of an optimizing type-centric compiler for a high-level language*. Technical Report. EPFL. https://doi.org/10.5075/epfl-thesis-7979

Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making collection operations optimal with aggressive JIT compilation. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*. 29–40. https://doi.org/10.1145/3136000.3136002

Michel Schinz. 2005. *Compiling Scala for the Java Virtual Machine*. Ph. D. Dissertation. École polytechnique fédérale de Lausanne. https://doi.org/10.5075/epfl-thesis-3302

Lukas Stadler, Gilles Duboscq, Hanspeter Mössenböck, Thomas Würthinger, and Doug Simon. 2013. An experimental study of the influence of dynamic compiler optimizations on Scala performance. In *Proceedings of the 4th Workshop on Scala*. 1–8. https://doi.org/10.1145/2489837.2489846

Christopher Strachey. 2000. Fundamental concepts in programming languages. *Higher-order and symbolic computation* 13, 1 (2000), 11–49. https://doi.org/10.1023/A:1010000313106

Bjarne Stroustrup. 1999. An overview of the C++ programming language. *Handbook of object technology* (1999), 72. https://doi.org/10.1201/9780849331350.sec3

Vlad Ureche, Cristian Talau, and Martin Odersky. 2013. 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*. 73–92. https://doi.org/10.1145/2544173.2509537

Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. 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*. 133–144. https://doi.org/10.1145/2647508.2647517

Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 662–676. https://doi.org/10.1145/3062341.3062381

Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204. https://doi.org/10.1145/2509578.2509581

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*. 73–82. https://doi.org/10.1145/2384577.2384587