Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization

Heather Miller
EPFL, Switzerland
heather.miller@epfl.ch

Philipp Haller
Typesafe, Inc.
philipp.haller@typesafe.com

Eugene Burmako
EPFL, Switzerland
eugene.burmako@epfl.ch

Martin Odersky
EPFL, Switzerland
martin.odersky@epfl.ch

Abstract

As more applications migrate to the cloud, and as "big data" edges into even more production environments, the performance and simplicity of exchanging data between compute nodes/devices is increasing in importance. An issue central to distributed programming, yet often under-considered, is serialization or pickling, *i.e.,* persisting runtime objects by converting them into a binary or text representation. Pickler combinators are a popular approach from functional programming; their composability alleviates some of the tedium of writing pickling code by hand, but they don't translate well to object-oriented programming due to qualities like open class hierarchies and subtyping polymorphism. Furthermore, both functional pickler combinators and popular, Java-based serialization frameworks tend to be tied to a specific pickle format, leaving programmers with no choice of how their data is persisted. In this paper, we present object-oriented pickler combinators and a framework for generating them at compile-time, called *scala/pickling*, designed to be the default serialization mechanism of the Scala programming language. The static generation of OO picklers enables significant performance improvements, outperforming Java and Kryo in most of our benchmarks. In addition to high performance and the need for little to no boilerplate, our framework is extensible: using the type class pattern, users can provide both (1) custom, easily interchangeable pickle formats and (2) custom picklers, to override the default behavior of the pickling framework. In benchmarks, we compare scala/pickling with other popular industrial frameworks, and present results on time, memory usage, and size when pickling/unpickling a number of data types used in real-world, large-scale distributed applications and frameworks.

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509547

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – multiparadigm languages, object-oriented languages, applicative (functional) languages; D.3.3 [*Programming Languages*]: Language Constructs and Features – input/output

Keywords Serialization, pickling, meta-programming, distributed programming, Scala

1. Introduction

With the growing trend towards cloud computing and mobile applications, distributed programming has entered the mainstream. As more and more traditional applications migrate to the cloud, the demand for interoperability between different services is at an all-time high, and is increasing. At the center of it all is communication. Whether we consider a cluster of commodity machines churning through a massive data-parallel job, or a smartphone interacting with a social network, all are "distributed" jobs, and all share the need to communicate in various ways, in many formats, even within the same application.

A central aspect to this communication that has received surprisingly little attention in the literature is the need to serialize, or *pickle* objects, *i.e.,* to persist in-memory data by converting them to a binary, text, or some other representation. As more and more applications evolve the need to communicate with different machines, providing abstractions and constructs for easy-to-use, typesafe, and performant serialization is more important than ever.

On the JVM, serialization has long been acknowledged as having a high overhead [7, 41], with some estimates purporting object serialization to account for 25-65% of the cost of remote method invocation, and which go on to observe that the cost of serialization grows with growing object structures up to 50% [18, 27]. Due to the prohibitive cost of using Java Serialization in high-performance distributed applications, many frameworks for distributed computing, like Akka [37], Spark [42], SCADS [3], and others, provide support for higher-performance alternative frameworks such as Google's Protocol Buffers [13], Apache Avro [1], or Kryo [21]. However, the higher efficiency typically comes at

the cost of weaker or no type safety, a fixed serialization format, more restrictions placed on the objects to-be-serialized, or only rudimentary language integration.

This paper takes a step towards more principled open programming through a new foundation for pickling in object-oriented languages. We present object-oriented picklers and *scala/pickling*, a framework for their generation either at runtime or at compile time. The introduced notion of object-oriented pickler combinators extends pickler combinators known from functional programming [17] with support for object-oriented concepts such as subtyping, mix-in composition, and object identity in the face of cyclic object graphs. In contrast to pure functional-style pickler combinators, we employ static, type-based meta programming to compose picklers at compile time. The resulting picklers are efficient, since the pickling code is generated statically as much as possible, avoiding the overhead of runtime reflection [9, 12].

Furthermore, the presented pickling framework is extensible in several important ways. First, building on an object-oriented type-class-like mechanism [8], our approach enables retroactively adding pickling support to existing, unmodified types. Second, our framework provides pluggable pickle formats which decouple type checking and pickler composition from the lower-level aspects of data formatting. This means that the type safety guarantees provided by type-specialized picklers are "portable" in the sense that they carry over to different pickle formats.

The design of our framework has been guided by the following principles:

- **Ease of use**. The programming interface aims to require as little pickling boilerplate as possible. Thanks to dedicated support by the underlying virtual machine, Java's serialization [25] requires only little boilerplate, which mainstream Java developers have come to expect. Our framework aims to be usable in production environments, and must, therefore, be able to integrate with existing systems with minimal changes.
- **Performance**. The generated picklers should be efficient enough so as to enable their use in high-performance distributed, "big data", and cloud applications. One factor driving practitioners away from Java's default serialization mechanism is its high runtime overhead compared to alternatives such as Kryo, Google's Protocol Buffers or Apache's Avro serialization framework. However, such alternative frameworks offer only minimal language integration.
- **Extensibility**. It should be possible to add pickling support to existing types retroactively. This resolves a common issue in Java-style serialization frameworks where classes have to be marked as serializable upfront, complicating unanticipated change. Furthermore, type-class-like extensibility enables pickling also for types provided by the underlying runtime environment (including built-in types), or types of third-party libraries.
- **Pluggable Pickle Formats**. It should be possible to easily swap target pickle formats, or for users to provide their own customized format. It is not uncommon for a distributed application to require multiple formats for exchanging data, for example an efficient binary format for exchanging system messages, or JSON format for publishing feeds. Type-class-like extensibility makes it possible for users to define their own pickle format, and to easily *swap it in* at the use-site.
- **Type safety**. Picklers should be type safe through (a) type specialization and (b) dynamic type checks when unpickling to transition unpickled objects into the statically-typed "world" at a well-defined program point.
- **Robust support for object-orientation**. Concepts such as subtyping and mix-in composition are used very commonly to define regular object types in object-oriented languages. Since our framework does without a separate data type description language (*e.g.,* a schema), it is important that regular type definitions are sufficient to describe the types to-be-pickled. The Liskov substitution principle is used as a guidance surrounding the substitutability of both objects to-be-pickled and first-class picklers. Our approach is also general, supporting object graphs with cycles.

1.1 Selected Related Work

Some OO languages like Java and runtime environments like the JVM or .NET provide serialization for arbitrary types, provided entirely by the underlying virtual machine. While this approach is very convenient for the programmer, there are also several issues: (a) the pickling format cannot be exchanged (Java), (b) serialization relies on runtime reflection which hits performance, and (c) existing classes that do not extend a special marker interface are not serializable, which often causes oversights resulting in software engineering costs. In functional languages, pickler combinators [10, 17] can reduce the effort of manually writing pickling and unpickling functions to a large extent. However, existing approaches do not support object-oriented concepts such as subtyping polymorphism. Moreover, it is not clear whether local type inference as required in OO languages would yield a comparable degree of conciseness, acceptable to programmers used to Java-style serialization. Nonetheless, our approach builds on pickler combinators, capitalizing on their powerful composability.

Our approach of retrofitting existing types with pickling support builds on implicits in Scala [8] and is reminiscent of other type-class-like mechanisms, such as JavaGI [40] or C++ Concepts [29].

Additionally, in an effort to further reduce the boilerplate required to define or compose picklers using existing picklers, we present a framework for automatically generating picklers for compound types based on picklers for their component types. Given the close relationship of our implicit

picklers to type classes, this generation mechanism is related to Haskell's *deriving* mechanism [19]. One of the main differences is that our mechanism is faithful to subtyping. So far, our mechanism is specialized for pickling; an extension to a generic mechanism for composing type class instances is left for future work.

We discuss other related work in Section 7.

1.2 Contributions

This paper makes the following contributions:

- An extension to pickler combinators, well-known in functional programming, to support the core concepts of object-oriented programming, namely subtyping polymorphism, open class hierarchies, and object identity.
- A framework based on object-oriented pickler combinators which (a) enables retrofitting existing types with pickling support, (b) supports automatically generating picklers at compile time and at runtime, (c) supports pluggable pickle formats, and (d) does not require changes to the host language or the underlying virtual machine.
- A complete implementation of the presented approach in and for Scala.[1]
- An experimental evaluation comparing the performance of our framework with Java serialization and Kryo on a number of data types used in real-world, large-scale distributed applications and frameworks.

2. Overview and Usage

2.1 Basic Usage

Scala/pickling was designed so as to require as little boilerplate from the programmer as possible. For that reason, pickling or unpickling an object `obj` of type `Obj` requires simply,

```
import scala.pickling._
val pickle = obj.pickle
val obj2 = pickle.unpickle[Obj]
```

Here, the `import` statement imports scala/pickling, the method `pickle` triggers static pickler generation, and the method `unpickle` triggers static unpickler generation, where `unpickle` is parameterized on `obj`'s precise type `Obj`. Note that not every type has a `pickle` method; it is implemented as an extension method using an *implicit conversion*. This implicit conversion is imported into scope as a member of the `scala.pickling` package.

Implicit conversions. Implicit conversions can be thought of as methods which can be implicitly invoked based upon their type, and whether or not they are present in implicit scope. Implicit conversions carry the `implicit` keyword before their declaration. The `pickle` method is provided using the following implicit conversion (slightly simplified):

```
implicit def PickleOps[T](picklee: T) =
  new PickleOps[T](picklee)

class PickleOps[T](picklee: T) {
  def pickle: Pickle = ...
  ...
}
```

In a nutshell, the above implicit conversion is implicitly invoked, passing object `obj` as an argument, whenever the `pickle` method is invoked on `obj`. The above example can be written in a form where all invocations of implicit methods are explicit, as follows:

```
val pickle = PickleOps[Obj](obj).pickle
val obj2 = pickle.unpickle[Obj]
```

Optionally, a user can import a `PickleFormat`. By default, our framework provides a Scala Binary Format, an efficient representation based on arrays of bytes, though the framework provides other formats which can easily be imported, including a JSON format. Furthermore, users can easily extend the framework by providing their own `PickleFormats` (see Section 4.3.1).

Typically, the framework generates the required pickler itself inline in the compiled code, using the `PickleFormat` in scope. In the case of JSON, for example, this amounts to the generation of string concatenation code and field accessors for getting runtime values, all of which is inlined, generally resulting in high performance (see Section 6).

In rare cases, however, it is necessary to fall back to runtime picklers which use runtime reflection to access the state that is being pickled and unpickled. For example, a runtime pickler is used when pickling instances of a generic subclass of the static class type to-be-pickled.

Using scala/pickling, it's also possible to pickle and unpickle subtypes, even if the pickle and unpickle methods are called using supertypes of the type to-be-pickled. For example,

```
abstract class Person {
  def name: String
}

case class Firefighter(name: String, since: Int)
     extends Person

val ff: Person = Firefighter("Jim", 2005)
val pickle = ff.pickle
val ff2 = pickle.unpickle[Person]
```

In the above example, the runtime type of `ff2` will correctly be `Firefighter`.

This perhaps raises an important concern– what if the type that is passed as a type argument to method `unpickle` is incorrect? In this case, the framework will fail with a runtime exception at the call site of `unpickle`. This is an improvement

[1] See `http://github.com/scala/pickling/`

over other frameworks, which have less type information available at runtime, resulting in wrongly unpickled objects often propagating to other areas of the program before an exception is thrown.

Scala/pickling is also able to unpickle values of static type `Any`. Scala's pattern-matching syntax can make unpickling on less-specific types quite convenient, for example:

```
pickle.unpickle[Any] match {
  case Firefighter(n, _) => println(n)
  case _ => println("not a Firefighter")
}
```

Beyond dealing with subtypes, our pickling framework supports pickling/unpickling most Scala types, including generics, case classes, and singleton objects. Passing a type argument to pickle, whether inferred or explicit, which is an unsupported type leads to a compile-time error. This avoids a common problem in Java-style serialization where non-serializable types are only discovered at runtime, in general. Function types, however, are not yet supported, and are planned future work.

2.2 Advanced Usage

@pickleable Annotation. To handle subtyping correctly, the pickling framework generates dispatch code which delegates to a pickler specialized for the runtime type of the object to-be-pickled, or, if the runtime type is unknown, which is to be expected in the presence of separate compilation, to a generic, but slower, runtime pickler.

For better performance, scala/pickling additionally provides an annotation which, at compile-time, inserts a runtime type test to check whether the runtime class extends a certain class/trait. In this case, a method that returns the pickler specialized for that runtime class is called. If the class/trait has been annotated, the returned pickler is guaranteed to have been generated statically. Furthermore, the `@pickleable` annotation (implemented as a macro annotation) is expanded transitively in each subclass of the annotated class/trait.

This `@pickleable` annotation enables:

- library authors to guarantee to their clients that picklers for separately-compiled subclasses are fully generated at compile-time;
- faster picklers in general because one need not worry about having to fallback on a runtime pickler.

For example, assume the following class `Person` and its subclass `Firefighter` are defined in separately-compiled code.

```
// Library code
@pickleable class Person(val name: String)

// Client code
class Firefighter(override val name: String, salary: Int)
  extends Person(name)
```

Note that class `Person` is annotated with the `@pickleable` annotation. `@pickleable` is a *macro annotation* which generates additional methods for obtaining type-specialized picklers (and unpicklers). With the `@pickleable` annotation expanded, the code for class `Person` looks roughly as follows:

```
class Person(val name: String)
    extends PickleableBase {
  def pickler: SPickler[_] =
    implicitly[SPickler[Person]]
  ...
}
```

First, note that the supertypes of `Person` now additionally include the trait `PickleableBase`; it declares the abstract methods that the expansion of the macro annotation "fills in" with concrete methods. In this case, a `pickler` method is generated which returns an `SPickler[_]`.[2] Note that the `@pickleable` annotation is defined in a way where pickler generation is triggered in both `Person` and its subclasses.

Here, we obtain an instance of `SPickler[Person]` by means of implicits. The `implicitly` method, part of Scala's standard library, is defined as follows:

```
def implicitly[T](implicit e: T) = e
```

Annotating the parameter (actually, the parameter list) using the `implicit` keyword means that in an invocation of `implicitly`, the implicit argument list may be omitted if, for each parameter of that list, there is exactly one value of the right type in the *implicit scope*. The implicit scope is an adaptation of the regular variable scope; imported implicits, or implicits declared in an enclosing scope are contained in the implicit scope of a method invocation.

As a result, `implicitly[T]` returns the uniquely-defined implicit value of type `T` which is in scope at the invocation site. In the context of picklers, there might not be an implicit value of type `SPickler[Person]` in scope (in fact, this is typically only the case with custom picklers). In that case, a suitable pickler instance is generated using a macro def.

Macro defs. Macro defs are methods that are transparently loaded by the compiler and executed (or expanded) during compilation. A macro is defined as if it is a normal method, but it is linked using the `macro` keyword to an additional method that operates on abstract syntax trees.

```
def assert(x: Boolean, msg: String): Unit =
  macro assert_impl
def assert_impl(c: Context)
  (x: c.Expr[Boolean], msg: c.Expr[String]):
                            c.Expr[Unit] = ...
```

In the above example, the parameters of `assert_impl` are syntax trees, which the body of `assert_impl` operates on,

[2] The notation `SPickler[_]` is short for the existential type `SPickler[t] forSome { type t }`. It is necessary here, because picklers must be invariant in their type parameter, see Section 3.1.4.

itself returning an AST of type `Expr[Unit]`. It is `assert_impl` that is expanded and evaluated at compile-time. Its result is then inlined at the call site of `assert` and the inlined result is typechecked. It is also important to note that implicit defs as described above can be implemented as macros.

Scala/pickling provides an implicit macro def returning picklers for arbitrary types. Slightly simplified, it is declared as follows:

```
implicit def genPickler[T]: SPickler[T]
```

This macro def is expanded when invoking `implicitly[SPickler[T]]` if there is no implicit value of type `SPickler[T]` in scope.

Custom Picklers. It is possible to use manually written picklers in place of generated picklers. Typical motivations for doing so are (a) improved performance through specialization and optimization hints, and (b) custom pre-pickling and post-unpickling actions; such actions may be required to re-initialize an object correctly after unpickling. Creating custom picklers is greatly facilitated by modular composition using object-oriented pickler combinators. The design of these first-class object-oriented picklers and pickler combinators is discussed in detail in the following Section 3.

3. Object-Oriented Picklers

In the first part of this section (3.1) we introduce picklers as first-class objects, and, using examples, motivate the contracts that valid implementations must guarantee. We demonstrate that the introduced picklers enable modular, object-oriented pickler combinators, *i.e.,* methods for composing more complex picklers from simpler primitive picklers.

In the second part of this section (3.2) we present a formalization of object-oriented picklers based on an operational semantics.

3.1 Picklers in Scala

In scala/pickling, a *static pickler* for some type `T` is an instance of trait `SPickler[T]` which has a single abstract method, `pickle`:

```
trait SPickler[T] {
  def pickle(obj: T, builder: PBuilder): Unit
}
```

For a concrete type, say, class `Person` from Section 2, the `pickle` method of an `SPickler[Person]` converts `Person` instances to a pickled format, using a *pickle builder* (the `builder` parameter). Given this definition, picklers "are type safe in the sense that a type-specialized pickler can be applied only to values of the specialized type" [10]. The pickled result is not returned directly; instead, it can be requested from the `builder` using its `result()` method. Example:

```
val p = new Person("Jack")
...
val personPickler   = implicitly[SPickler[Person]]
val builder         = pickleFormat.createBuilder()
personPickler.pickle(p, builder)
val pickled: Pickle = builder.result()
```

In the above example, invoking `implicitly[SPickler[Person]]` either returns a regular implicit value of type `SPickler[Person]` that is in scope, or, if it doesn't exist, triggers the (compile-time) generation of a type-specialized pickler (see Section 4). To use the pickler, it is also necessary to obtain a pickle builder of type `PBuilder`. Since pickle formats in scala/pickling are exchangeable (see Section 4.3.1), the pickle builder is provided by the specific pickle format, through builder factory methods.

The `pickled` result has type `Pickle` which wraps a concrete representation, such as a byte array (*e.g.,* for binary formats) or a string (*e.g.,* for JSON). The abstract `Pickle` trait is defined as follows:

```
trait Pickle {
  type ValueType
  type PickleFormatType <: PickleFormat
  val value: ValueType
  ...
}
```

The type members `ValueType` and `PickleFormatType` abstract from the concrete representation type and the pickle format type, respectively. For example, scala/pickling defines a `Pickle` subclass for its default binary format as follows:

```
case class BinaryPickle(value: Array[Byte]) extends Pickle {
  type ValueType = Array[Byte]
  type PickleFormatType = BinaryPickleFormat
  override def toString = ...
}
```

Analogous to a pickler, an unpickler for some type `T` is an instance of trait `Unpickler[T]` that has a single abstract method `unpickle`; its (simplified) definition is as follows:

```
trait Unpickler[T] {
  def unpickle(reader: PReader): T
}
```

Similar to a pickler, an unpickler does not access pickled objects directly, but through the `PReader` interface, which is analogous to the `PBuilder` interface. A `PReader` is set up to read from a pickled object as follows. First, we need to obtain an instance of the pickle format that was used to produce the pickled object; this format is either known beforehand, or it can be selected using the `PickleFormatType` member of `Pickle`. The pickle format, in turn, has factory methods for creating concrete `PReader` instances:

```
val reader = pickleFormat.createReader(pickled)
```

The obtained reader can then be passed to the `unpickle` method of a suitable `Unpickler[T]`. Alternatively, a macro def on trait `Pickle` can be invoked directly for unpickling:

```
trait Pickle {
  ...
  def unpickle[T] = macro ...
}
```

It is very common for an instance of `SPickler[T]` to also mix in `Unpickler[T]`, thereby providing both pickling and unpickling capabilities.

3.1.1 Pickling and Subtyping

So far, we have introduced the trait `SPickler[T]` to represent picklers that can pickle objects of type `T`. However, in the presence of subtyping and open class hierarchies providing correct implementations of `SPickler[T]` is quite challenging. For example, how can an `SPickler[Person]` know how to pickle an arbitrary, unknown subclass of `Person`? Regardless of implementation challenges, picklers that handle arbitrary subclasses are likely less efficient than more specialized picklers.

To provide flexibility while enabling optimization opportunities, scala/pickling introduces two different traits for picklers: the introduced trait `SPickler[T]` is called a static pickler; it does not have to support pickling of subclasses of `T`. In addition, the trait `DPickler[T]` is called a dynamic pickler; its contract requires that it is applicable also to subtypes of `T`. The following section motivates the need for dynamic picklers, and shows how the introduced concepts enable a flexible, object-oriented form of pickler combinators.

3.1.2 Modular Pickler Combinators

This section explores the composition of the pickler abstractions introduced in the previous section by means of an example. Consider a simple class `Position` with a field of type `String` and a field of type `Person`, respectively:

```
class Position(val title: String, val person: Person)
```

To obtain a pickler for objects of type `Position`, ideally, existing picklers for type `String` and for type `Person` could be combined in some way. However, note that the `person` field of a given instance of class `Position` could point to an instance of a subclass of `Person` (assuming class `Person` is not final). Therefore, a modularly re-usable pickler for type `Person` must be able to pickle all possible subtypes of `Person`.

In this case, the contract of static picklers is too strict, it does not allow for subtyping. The contract of dynamic picklers on the other hand does allow for subtyping. As a result, *dynamic picklers are necessary so as to enable modular composition in the presence of subtyping*.

Picklers for final class types like `String`, or for primitive types like `Int` do not require support for subtyping. Therefore, static picklers are sufficient to pickle these *effectively final types*. Compared to dynamic picklers, static picklers benefit from several optimizations.

3.1.3 Implementing Object-Oriented Picklers

The main challenge when implementing OO picklers comes from the fact that a dynamic pickler for type `T` must be able to pickle objects of any subtype of `T`. Thus, the implementation of a dynamic pickler for type `T` must, in general, dynamically dispatch on the runtime type of the object to-be-pickled to take into account all possible subtypes of `T`. Because of this dynamic dispatch, manually constructing dynamic picklers can be difficult. It is therefore important for a framework for object-oriented picklers to provide good support for realizing this form of dynamic dispatching.

There are various ways across many different object-oriented programming languages to handle subtypes of the pickler's static type:

- Data structures with shallow class hierarchies, such as lists or trees, often have few final leaf classes. As a result, manual dispatch code is typically simple in such cases. For example, a manual pickler for Scala's `List` class does not even have to consider subclasses.
- Java-style runtime reflection can be used to provide a generic `DPickler[Any]` which supports pickling objects of any type [25, 27]. Such a pickler can be used as a fallback to handle subtypes that are unknown to the pickling code; such subtypes must be handled in the presence of separate compilation. In Section 4.4 we present Scala implementations of such a generic pickler.
- Java-style annotation processing is commonly used to trigger the generation of additional methods in annotated class types. The purpose of generated methods for pickling would be to return a pickler or unpickler specialized for an annotated class type. In C#, the Roslyn Project [22] allows augmenting class definitions based on the presence of annotations.
- Static meta programming [5, 34] enables generation of picklers at compile time. In Section 4 we present an approach for generating object-oriented picklers from regular (class) type definitions.

3.1.4 Supporting Unanticipated Evolution

Given the fact that the type `SPickler[T]`, as introduced, has a type parameter `T`, it is reasonable to ask what the variance of `T` is. Ruling out covariance because of `T`'s occurrence in a contravariant position as the type of a method parameter, it remains to determine whether `T` can be contravariant.

For this, it is useful to consider the following scenario. Assume `T` is declared to be contravariant, as in `SPickler[-T]`. Furthermore, assume the existence of a public, non-final class `C` with a subclass `D`:

```
class C {...}
class D extends C {...}
```

Initially, we might define a generic pickler for `C`:

```
implicit val picklerC = new SPickler[C] {
  def pickle(obj: C): Pickle = { ... }
}
```

Because `SPickler`$[T]$ is contravariant in its type parameter, instances of `D` would be pickled using `picklerC`. There are several possible extensions that might be *unanticipated* initially:

- Because the implementation details of class `D` change, instances of `D` should be pickled using a dedicated pickler instead of `picklerC`.
- A subclass `E` of `C` is added which requires a dedicated pickler, since `picklerC` does not know how to instantiate class `E` (since class `E` did not exist when `picklerC` was written).

In both cases it is necessary to add a new, dedicated pickler for either an existing subclass (`D`) or a new subclass (`E`) of `C`:

```
implicit val picklerD = new SPickler[D] { ... }
```

However, when pickling an instance of class `D` this new pickler, `picklerD`, would not get selected, even if the type of the object to-be-pickled is statically known to be `D`. The reason is that `SPickler[C]` $<:$ `SPickler[D]` because of contravariance which means that `picklerC` is more specific than `picklerD`. As a result, according to Scala's implicit look-up rules `picklerC` is selected when an implicit object of type `SPickler[D]` is required. (Note that this is the case even if `picklerD` is declared in a scope that has higher precedence than the scope in which `picklerC` is declared.)

While contravariant picklers do not support the two scenarios for unanticipated extension outlined above, invariant picklers do, in combination with type bounds. Assuming invariant picklers, we can define a generic method `picklerC1` that returns picklers for all subtypes of class `C`:

```
implicit def picklerC1[T <: C] = new SPickler[T] {
  def pickle(obj: T): Pickle = { ... }
}
```

With this pickler in scope, it is still possible to define a more specific `SPickler[D]` (or `SPickler[E]`) as required:

```
implicit val picklerD1 = new SPickler[D] { ... }
```

However, the crucial difference is that now `picklerD1` is selected when an object of static type `D` is pickled, since `picklerD1` is more specific than `picklerC1`.

In summary, the combination of invariant picklers and generics (with upper type bounds) is flexible enough to support some important scenarios of unanticipated evolution. This is not possible with picklers that are contravariant. Consequently, in scala/pickling the `SPickler` trait is invariant in its type parameter.

3.2 Formalization

To define picklers formally we use a standard approach based on an operational semantics for a core object-oriented language. Importantly, our goal is not a full formalization of a core language; instead, we (only) aim to provide a precise definition of *object-oriented picklers*. Thus, our core language simplifies our actual implementation language in several ways. Since our basic definitions are orthogonal to the type system of the host language, we limit types to non-generic classes with at most one superclass. Moreover, the core language does not have first-class functions, or features like pattern matching. The core language without picklers is a simplified version of a core language used in the formal development of a uniqueness type system for Scala [14].

$P ::=$	$\overline{cdef}\ t$	program
$cdef ::=$	`class` C `extends` $D\ \{\overline{fld}\ \overline{meth}\}$	class
$fld ::=$	`var` $f : C$	field
$meth ::=$	`def` $m(\overline{x : C}) : D = e$	method
$t ::=$	`let` $x = e$ `in` t	let binding
	$\mid x.f := y$	assignment
	$\mid x$	variable
$e ::=$	`new` $C(\overline{x})$	instance creation
	$\mid x.f$	selection
	$\mid x.m(\overline{y})$	invocation
	$\mid t$	term

Figure 1: Core language syntax. C, D are class names, f, m are field and method names, and x, y are names of variables and parameters, respectively.

$H ::=$	$\emptyset \mid (H, r \mapsto v)$	heap
$V ::=$	$\emptyset \mid (V, y \mapsto r)$	environment ($y \notin dom(V)$)
$v ::=$	$o \mid \rho$	value
$o ::=$	$C(\overline{r})$	object
$\rho ::=$	(C_p, m, C)	pickler
$r \in$	$RefLocs$	reference location

Figure 2: Heaps, environments, objects, and picklers.

Figure 1 shows the core language syntax. A program is a sequence of class definitions followed by a (main) term. (We use the common over-bar notation [16] for sequences.) Without loss of generality, we use a form where all intermediate terms are named (A-normal form [11]). The language does not support arbitrary mutable variables (*cf.* [28], Chapter 13); instead, only fields of objects can be (re-)assigned.

We assume the existence of two pre-defined class types, `AnyRef` and `Pickle`. All class hierarchies have `AnyRef` as their root. For the purpose of our core language, `AnyRef` is simply a member-less class without a superclass. `Pickle` is the class type of objects that are the result of pickling a regular object.

We define the standard auxiliary functions $mtype$ and $mbody$ as follows. Let `def` $m(\overline{x : C}) : D = e$ be a method defined in the most direct superclass of C that defines m. Then $mbody(m, C) = (\overline{x}, e)$ and $mtype(m, C) = \overline{C} \to D$.

$$\frac{\begin{array}{c}V(x) = r_p \qquad H(r_p) = (C_p, s, C) \\ V(y) = r \qquad H(r) = C(\_) \\ mbody(p, C_p) = (z, e)\end{array}}{\begin{array}{l}H, V, \texttt{let}\ x' = x.p(y)\ \texttt{in}\ t \\ \longrightarrow\ H, (V, z \mapsto r), \texttt{let}\ x' = e\ \texttt{in}\ t\end{array}}\ \text{(R-Pickle-S)}$$

$$\frac{\begin{array}{c}V(x) = r_p \qquad H(r_p) = (C_p, d, C) \\ V(y) = r \qquad H(r) = D(\_) \qquad D <: C \\ mbody(p, C_p) = (z, e)\end{array}}{\begin{array}{l}H, V, \texttt{let}\ x' = x.p(y)\ \texttt{in}\ t \\ \longrightarrow\ H, (V, z \mapsto r), \texttt{let}\ x' = e\ \texttt{in}\ t\end{array}}\ \text{(R-Pickle-D)}$$

$$\frac{\begin{array}{c}V(x) = r \qquad H(r) = C(\_) \\ V(\overline{y}) = r_1 \ldots r_n \\ mbody(m, C) = (\overline{x}, e)\end{array}}{\begin{array}{l}H, V, \texttt{let}\ x' = x.m(\overline{y})\ \texttt{in}\ t \\ \longrightarrow\ H, (V, \overline{x \mapsto r}), \texttt{let}\ x' = e\ \texttt{in}\ t\end{array}}\ \text{(R-Invoke)}$$

Figure 3: Reduction rules for pickling.

3.2.1 Dynamic semantics

We use a small-step operational semantics to formalize the dynamic semantics of our core language. Reduction rules are written in the form $H, V, t \longrightarrow H', V', t'$. That is, terms t are reduced in the context of a heap H and a variable environment V. Figure 2 shows their syntax. A heap maps reference locations to values. In our core language, values can be either objects or picklers. An object $C(\overline{r})$ stores location r_i in its i-th field. An environment maps variables to reference locations r. Note that we do not model explicit stack frames. Instead, method invocations are "flattened" by renaming the method parameters before binding them to their argument values in the environment (as in LJ [35]).

A pickler is a tuple (C_p, m, C) where C_p is a class that defines two methods p and u for pickling and unpickling an object of type C, respectively, where $mtype(p, C_p) = C \rightarrow \texttt{Pickle}$ and $mtype(u, C_p) = \texttt{Pickle} \rightarrow C$. The second component $m \in \{s, d\}$ is the pickler's *mode*; the operational semantics below explains how the mode affects the applicability of a pickler in the presence of subtyping.

As defined, picklers are first-class, since they are values just like objects. However, while picklers are regular objects in our practical implementation, picklers are different from objects in the present formal model. The reason is that a pickler has to contain a type tag indicating the types of objects that it can pickle (this is apparent in the rules of the operational semantics below); however, the alternative of adding parameterized types (as in, *e.g.,* FGJ [16]) is beyond the scope of the present paper.

According to the grammar in Figure 1, expressions are always reduced in the context of a let-binding, except for field assignments. Each operand of an expression is a variable y that the environment maps to a reference location r. Since the environment is a flat list of variable bindings, let-bound variables must be alpha-renamable: $\texttt{let}\ x = e\ \texttt{in}\ t \equiv \texttt{let}\ x' = e\ \texttt{in}\ [x'/x]t$ where $x' \notin FV(t)$. (We omit the definition of the FV function to obtain the free variables of a term, as it is standard [28].)

In the following we explain the subset of the reduction rules suitable to formalize the properties of picklers. We start with the reduction rule for method invocations, since the reduction rules pertinent to picklers are variants of that rule.

Figure 3 shows the reduction rules for pickling and unpickling an object.

Rule (R-Pickle-S) is a refinement of rule (R-Invoke) for method invocations. When using a pickler x to pickle an object y such that the pickler's mode is s (static), the type tag C of the pickler indicating the type of objects that it can pickle must be equal to the dynamic class type of the object to-be-pickled (the object at location r). This expresses the fact that a static pickler can only be applied to objects of a precise statically-known type C, but not a subtype thereof.

In contrast, rule (R-Pickle-D) shows the invocation of the pickling method p for a pickler with mode d (dynamic). In this case, the type tag C of the pickler must not be exactly equal to the dynamic type of the object to-be-pickled (the object at location r); it is only necessary that $D <: C$.

Property. The pickling and unpickling methods of a pickler must satisfy the property that "pickling followed by unpickling generates an object that is structurally equal to the original object". The following definition captures this formally:

Definition 3.1. Given variables x, x', y, y', heaps H, H', variable environments V, V', and a term t such that

$$\begin{array}{ll} V(y) = r & H(r) = C(\overline{r}) \\ V(x) = r_p & H(r_p) = (C_p, m, D) \\ \begin{cases} D = C & \text{if } m = s \\ D <: C & \text{if } m = d \end{cases} & \\ V'(y') = r' & \end{array}$$

and

$$H, V, \texttt{let}\ x' = x.u(x.p(y))\ \texttt{in}\ t \longrightarrow^* H', V', \texttt{let}\ x' = y'\ \texttt{in}\ t$$

Then r and r' must be structurally equivalent in heap H', written $r \equiv_{H'} r'$.

Note that in the above definition we assume that references in heap H are not garbage collected in heap H'. The definition of structural equivalence is straight-forward.

Definition 3.2. (Structural Equivalence)
Two picklers r_p, r'_p are structurally equal in heap H, written $r_p \equiv_H r'_p$ *iff*

$$H(r_p) = (C_p, m, C) \wedge H(r'_p) = (C'_p, m', C') \Rightarrow m = m' \wedge C <: C' \wedge C' <: C \quad (1)$$

Two reference locations r, r' are structurally equal in heap H, written $r \equiv_H r'$ *iff*

$$H(r) = C(\overline{r}) \wedge H(r') = C'(\overline{p}) \Rightarrow C <: C' \wedge C' <: C \wedge \forall r_i \in \overline{r}, p_i \in \overline{p}.\ r_i \equiv_H p_i \quad (2)$$

Note that the above definition considers two picklers to be structurally equal even if their implementation classes C_p and C'_p are different. In some sense, this is consistent with our practical implementation in the common case where picklers are only resolved using implicits: Scala's implicit resolution enforces that an implicit pickler of a given type is uniquely determined.

3.3 Summary

This section has introduced an object-oriented model of first-class picklers. Object-oriented picklers enable modular pickler combinators with support for subtyping, thereby extending a well-known approach in functional programming. The distinction between static and dynamic picklers enables optimizations for final class types and primitive types. Object-oriented picklers can be implemented using various techniques, such as manually written picklers, runtime reflection, or Java-style annotation processors. We argue that object-oriented picklers should be invariant in their generic type parameter to allow for several scenarios of unanticipated evolution. Finally, we provide a formalization of a simple form of OO picklers.

4. Generating Object-Oriented Picklers

An explicit goal of our framework is to require little to no boilerplate in client code, since practitioners are typically accustomed to serialization supported by the underlying runtime environment like in Java or .NET. Therefore, instead of requiring libraries or applications to supply manually written picklers for all pickled types, our framework provides a component for *generating picklers* based on their required static type.

Importantly, compile-time pickler generation enables *efficient picklers* by generating as much pickling code as possible statically (which corresponds to a partial evaluation of pickler combinators). Section 6 reports on the performance improvements that our framework achieves using compile-time pickler generation, compared to picklers based on runtime reflection, as well as manually written picklers.

4.1 Overview

Our framework generates type-specialized, object-oriented picklers using compile-time meta programming in the form of *macros*. Whenever a pickler for static type T is required but cannot be found in the implicit scope, a macro is expanded which generates the required pickler step-by-step by:

- Obtaining a type descriptor for the static type of the object to-be-pickled,
- Building a static *intermediate representation* of the object-to-be-pickled, based on the type descriptor, and
- Applying a pickler generation algorithm, driven by the static pickler representation.

In our Scala-based implementation, the static type descriptor is generated automatically by the compiler, and passed as an implicit argument to the pickle extension method (see Section 2). As a result, such an implicit `TypeTag`[1] does not require changing the invocation in most cases. (However, it is impossible to generate a `TypeTag` automatically if the type or one of its components is abstract; in this case, an implicit `TypeTag` must be in scope.)

Based on the type descriptor, a static representation, or model, of the required pickler is built; we refer to this as the *Intermediate Representation* (IR). The IR specifies precisely the set of types for which our framework can generate picklers automatically. Furthermore, these IRs are composable.

We additionally define a model for composing IRs, which is designed to capture the essence of Scala's object system as it relates to pickling. The model defines how the IR for a given type is composed from the IRs of the picklers of its supertypes. In Scala, the composition of an IR for a class type is defined based on the linearization of its supertraits.[2] This model of inheritance is central to the generation framework, and is formally defined in the following Section 4.2

4.2 Model of Inheritance

The goal of this section is to define the IR, which we'll denote Υ, of a static type T as it is used to generate a pickler for type T. We start by defining the syntax of the elements of the IR (see Def. 4.1).

[1] TypeTags are part of the mainline Scala compiler since version 2.10. They replace the earlier concept of Manifests, providing a faithful representation of Scala types at runtime.

[2] Traits in Scala can be thought of as a more flexible form of Java-style interfaces that allow concrete members, and that support a form of multiple inheritance (mix-in composition) that is guaranteed to be safe based on a linearization order.

Definition 4.1. (Elements of IR)
We define the syntax of values of the IR types.

$$F ::= \overline{(f_n, T)}$$
$$\Upsilon ::= (T, \Upsilon_{opt}, F)$$
$$\Upsilon_{opt} ::= \epsilon \mid \Upsilon$$

F represents a sequence of *fields*. We write $\overline{X}$ as shorthand for sequences, $X_1, \ldots, X_n$, and we write tuples $(X_1, \ldots, X_n)$. f_n is a string representing the name of the given field, and T is its type.

Υ represents the pickling information for a class or some other object type. That is, an Υ for type T contains all of the information required to pickle instances of type T, including all necessary static info for pickling its fields provided by F.

Υ_{opt} is an optional Υ; a missing Υ is represented using ϵ.

In our implementation the IR types are represented using case classes. For example, the following case class represents Υs:

```
case class ClassIR(
  tpe: Type,
  parent: ClassIR,
  fields: List[FieldIR]
) extends PickleIR
```

We go on to define a number of useful IR combinators, which form the basis of our model of inheritance.

Definition 4.2. (IR Combinators - Type Definitions)
We begin by defining the types of our combinators before we define the combinators themselves.

Type Definitions

$$\textit{concat} : (F, F) \Rightarrow F$$
$$\textit{extended} : (\Upsilon, \Upsilon) \Rightarrow \Upsilon$$
$$\textit{linearization} : T \Rightarrow \overline{T}$$
$$\textit{superIRs} : T \Rightarrow \overline{\Upsilon}$$
$$\textit{compose} : \Upsilon \Rightarrow \Upsilon$$
$$\textit{flatten} : \Upsilon \Rightarrow \Upsilon$$

We write function types $X \Rightarrow Y$, indicating a function from type X to type Y.

The *linearization* function represents the host language's semantics for the linearized chain of supertypes.[3]

[3] For example, in Scala the linearization is defined for classes mixing in multiple traits [23, 24]; in Java, the linearization function would simply return the chain of superclasses, not including the implemented interfaces.

Definition 4.3. (IR Combinators - Function Defns)

Function Definitions

$$\textit{concat}(\overline{f}, \overline{g}) = \overline{f}, \overline{g}$$
$$\textit{extended}(C, D) = (T, C, \textit{fields}(T)) \quad \text{where } D = (T, \_, \_) \ \wedge T <: C.1$$
$$\textit{superIRs}(T) = [(S, \epsilon, \textit{fields}(S)) \mid S \in \textit{linearization}(T)]$$
$$\textit{compose}(C) = \textit{reduce}(\textit{superIRs}(C.1), \textit{extended})$$
$$\textit{flatten}(C) = \begin{cases} (C.1, C.2, \textit{concat}(C.3, \textit{flatten(C.2).3})), & \text{if } C.2 \neq \epsilon \\ C, & \text{otherwise} \end{cases}$$

The function *concat* takes two sequences as arguments. We denote concatenation of sequences using a comma. We introduce the *concat* function for clarity in the definition of *flatten* (see below); it is simply an alias for sequence concatenation.

The function *extended* takes two Υs, C and D, and returns a new Υ for the type of D such that C is registered as its super Υ. Basically, *extended* is used to combine a completed Υ C with an incomplete Υ D yielding a completed Υ for the same type as D. When combining the Υs of a type's supertypes, the *extended* function is used for reducing the linearization sequence yielding a single completed Υ.

The function *superIRs* takes a type T and returns a sequence of the IRs of T's supertypes in linearization order.

The function *compose* takes an Υ C for a type $C.1$ and returns a new Υ for type $C.1$ which is the composition of the IRs of all supertypes of $C.1$. The resulting Υ is a chain of super IRs according to the linearization order of $C.1$.

The function *flatten*, given an Υ C produces a new Υ that contains a concatenation of all the fields of each nested Υ. Given these combinators, the Υ of a type T to-be-pickled is obtained using $\Upsilon = flatten(compose((T, \epsilon, [])))$.

The above IR combinators have direct Scala implementations in scala/pickling. For example, function $superIRs$ is implemented as follows:

```
private val f3 = (c: C) =>
  c.tpe.baseClasses
       .map(superSym => c.tpe.baseType(superSym))
       .map(tp => ClassIR(tp, null, fields(tp)))
```

Here, method `baseClasses` returns the collection of superclass symbols of type `c.tpe` in linearization order. Method `baseType` converts each symbol to a type which is, in turn, used to create a `ClassIR` instance. The semantics of the `fields` method is analogous to the above $fields$ function.

4.3 Pickler Generation Algorithm

The pickler generation is driven by the IR (see Section 4.2) of a type to-be-pickled. We describe the generation algorithm in two steps. In the first step, we explain how to generate a pickler for static type T assuming that for the dynamic type S of the object to-be-pickled, $\mathit{erasure}(T) =:= S$. In the second step, we explain how to extend the generation to dynamic picklers which do not require this assumption.

4.3.1 Pickle Format

The pickling logic that we are going to generate contains calls to a pickle *builder* that is used to incrementally construct a pickle. Analogously, the unpickling logic contains calls to a pickle *reader* that is used to incrementally read a pickle. Importantly, the pickle format that determines the precise persisted representation of a completed pickle is not fixed. Instead, the pickle format to be used is selected at compile time– efficient binary formats, and JSON are just some examples. This selection is done via implicit parameters which allows the format to be flexibly selected while providing a default binary format which is used in case no other format is imported explicitly.

The pickle format provides an interface which plays the role of a simple, lower-level "backend". Besides a pickle template that is generated inline as part of the pickling logic, methods provided by pickle builders aim to do as little as possible to minimize runtime overhead. For example, the JSON `PickleFormat` included with scala/pickling simply uses an efficient string builder to concatenate JSON fragments (which are just strings) in order to assemble a pickle.

The interface provided by `PickleFormat` is simple: it basically consists of two methods (a) for creating an empty builder, and (b) for creating a reader from a pickle:[3]

```
def createBuilder(): PBuilder
def createReader(pickle: PickleType): PReader
```

The `createReader` method takes a pickle of a specific `PickleType` (which is an abstract type member in our implementation); this makes it possible to ensure that, say, a pickle encapsulating a byte array is not erroneously attempted to be unpickled using the JSON pickle format. Moreover, pickle builders returned from `createBuilder` are guaranteed to produce pickles of the right type.

```
class PBuilder {
  def beginEntry(obj: Any): PBuilder
  def putField(n: String, pfun: PBuilder => Unit): PBuilder
  def endEntry(): Unit
  def result(): Pickle
}
```

In the following we're going to show how the `PBuilder` interface is used by generated picklers; the `PReader` interface is used by generated unpicklers in an analogous way. The above example summarizes a core subset of the interface of `PBuilder` that the presented generation algorithm is going to use.[4] The `beginEntry` method is used to indicate the start of a pickle for the argument `obj`. The field values of a class instance are pickled using `putField` which expects both a field name and a lambda encapsulating the pickling logic for the object that the field points to. The `endEntry` method indicates the completion of a (partial) pickle of an object. Finally, invoking `result` returns the completed `Pickle` instance.

4.3.2 Tree Generation

The objective of the generation algorithm is to generate the body of `SPickler`'s `pickle` method:

```
def pickle(obj: T, builder: PBuilder): Unit = ...
```

As mentioned previously, the actual pickling logic is synthesized based on the IR. Importantly, the IR determines which fields are pickled and how. A lot of the work is already done when building the IR; therefore, the actual tree generation is rather simple:

- Emit `builder.beginEntry(obj)`.
- For each field `fld` in the IR, emit `builder.putField(${fld.name},b => pbody)` where `${fld.name}` denotes the splicing of `fld.name` into the tree. `pbody` is the logic for pickling `fld`'s value into the builder `b`, which is an alias of `builder`. `pbody` is generated as follows:
  1. Emit the field getter logic: `val v: ${fld.tpe} = obj.${fld.name}`. The expression `${fld.tpe}` splices the type of `fld` into the generated tree; `${fld.name}` splices the name of `fld` into the tree.
  2. Recursively generate the pickler for `fld`'s type by emitting either `val fldp = implicitly[DPickler[${fld.tpe}]]` or `val fldp = implicitly[SPickler[${fld.tpe}]]`, depending on whether `fld`'s type is effectively final or not.
  3. Emit the logic for pickling `v` into `b`: `fldp.pickle(v, b)`

A practical implementation can easily be refined to support various extensions of this basic model. For example, support for avoiding pickling fields marked as *transient* is easy with this model of generation– such fields can simply be left out of the IR. Or, based on the static types of the picklee and its fields, we can emit hints to the builder to enable various optimizations.

For example, a field whose type T is *effectively final*, *i.e.,* it cannot be extended, can be optimized as follows:

- Instead of obtaining an implicit pickler of type `DPickler[T]`, it is sufficient to obtain an implicit pickler of type `SPickler[T]`,

[3] In our actual implementation the `createReader` method takes an additional parameter which is a "mirror" used for runtime reflection; it is omitted here for simplicity.

[4] It is not necessary that `PBuilder` is a class. In fact, in our Scala implementation it is a trait. In Java, it could be an interface.

which is more efficient, since it does not require a dynamic dispatch step like `DPickler[T]`

- The field's type does not have to be pickled, since it can be reconstructed from its owner's type.

Pickler generation is compositional; for example, the generated pickler for a class type with a field of type `String` re-uses the `String` pickler. This is achieved by generating picklers for parts of an object type using invocations of the form `implicitly[DPickler[T]]`. This means that if there is already an implicit value of type `DPickler[T]` in scope, it is used for pickling the corresponding value. Since the lookup and binding of these implicit picklers is left to a mechanism outside of pickler generation, what's actually generated is a *pickler combinator* which returns a *pickler* composed of *existing picklers* for parts of the object to-be-pickled. More precisely, pickler generation provides the following composability property:

Property 4.1. (Composability) A generated pickler `p` is composed of implicit picklers of the required types that are in scope at the point in the program where `p` is generated.

Since the picklers that are in scope at the point where a pickler is generated are under programmer control, it is possible to import manually written picklers which are transparently picked up by the generated pickler. Our approach thus has the attractive property that it is an "open-world" approach, in which it is easy to add new custom picklers for selected types at exactly the desired places while integrating cleanly with generated picklers.

4.3.3 Dispatch Generation

So far, we have explained the generation of the pickling logic of static picklers. Dynamic picklers require an additional dispatch step to make sure subtypes of the static type to-be-pickled are pickled properly. The generation of a `DPickler[T]` is triggered by invoking `implicitly[DPickler[T]]` which tries to find an implicit of type `DPickler[T]` in the current implicit scope. Either there is already an implicit value of the right type in scope, or the only matching implicit is an implicit def provided by the pickling framework which generates a `DPickler[T]` on-the-fly. The generated dispatch logic has the following shape:

```
val clazz = if (picklee != null) picklee.getClass else null
val pickler = clazz match {
  case null => implicitly[SPickler[NullTpe]]
  case c1 if c1 == classOf[S1] => implicitly[SPickler[S1]]
  ...
  case cn if cn == classOf[Sn] => implicitly[SPickler[Sn]]
  case _ => genPickler(clazz)
}
```

The types $S_1, \ldots, S_n$ are known subtypes of the picklee's type T. If T is a sealed class or trait with final subclasses, this set of types is always known at compile time. However, in the presence of separate compilation it is, generally, possible that a picklee has an unknown runtime type; therefore, we include a default case (the last case in the pattern match) which dispatches to a runtime pickler that inspects the picklee using (runtime) reflection.

If the static type T to be pickled is annotated using the `@pickleable` annotation, all subclasses are guaranteed to extend the predefined `PickleableBase` interface trait. Consequently, a more optimal dispatch can be generated in this case:

```
val pickler =
  if (picklee != null) {
    val pbase = picklee.asInstanceOf[PickleableBase]
    pbase.pickler.asInstanceOf[SPickler[T]]
  }
  else implicitly[SPickler[NullTpe]]
```

4.4 Runtime Picklers

One goal of our framework is to generate as much pickling code at compile time as possible. However, due to the interplay of subclassing with both separate compilation and generics, we provide a runtime fall back capability to handle the cases that cannot be resolved at compile time.

Subclassing and separate compilation A situation arises where it's impossible to statically know all possible subclasses. In this case there are three options: (1) provide a custom pickler, and (2) use an annotation which is described in Section 2.2. In the case where neither a custom pickler nor an annotation is provided, our framework can inspect the instance to-be-pickled at runtime to obtain the pickling logic. This comes with some runtime overhead, but in Section 6 we present results which suggest that this overhead is not necessary in many cases.

For the generation of runtime picklers our framework supports two possible strategies:

- Runtime interpretation of a type-specialized pickler
- Runtime compilation of a type-specialized pickler

Interpreted runtime picklers. If the runtime type of an object is unknown at compile time, *e.g.,* if its static type is `Any`, it is necessary to carry out the pickling based on inspecting the type of the object to-be-pickled at runtime. We call picklers operating in this mode "interpreted runtime picklers" to emphasize the fact that the pickling code is not partially evaluated in this case. An interpreted pickler is created based on the runtime class of the picklee. From that runtime class, it is possible to obtain a runtime type descriptor:

- to build a static intermediate representation of the type (which describes all its fields with their types, etc.)

- to determine in which way the picklee should be pickled (as a primitive or not).

In case the picklee is of a primitive type, there are no fields to be pickled. Otherwise, the value and runtime type of each field is obtained, so that it can be written to the pickle.

4.5 Generics and Arrays

Subclassing and generics. The combination of subclassing and generics poses a similar problem to that introduced above in Section 4.4. For example, consider a generic class `C`,

```
class C[T](val fld: T) { ... }
```

A `Pickler[C[T]]` will not be able to pickle the field `fld` if its static type is unknown. To support pickling instances of generic classes, our framework falls back to using runtime picklers for pickling fields of generic type. So, when we have access to the runtime type of field `fld`, we can either look up an already-generated pickler for that runtime type, or we can generate a suitable pickler dynamically.

Arrays. Scala arrays are mapped to Java arrays; the two have the same runtime representation. However, there is one important difference: Java arrays are covariant whereas Scala arrays are invariant. In particular, it is possible to pass arrays from Java code to Scala code. Thus, a class `C` with a field `f` of type `Array[T]` may have an instance at runtime that stores an `Array[S]` in field `f` where `S` is a subtype of `T`. Pickling followed by unpickling must instantiate an `Array[S]`. Just like with other fields of non-final reference type, this situation requires writing the dynamic (array) type name to the pickle. This is possible, since array types are not erased on the JVM (unlike generic types). This allows instantiating an array with the expected dynamic type upon unpickling. At the time of writing only support for primitive arrays has been implemented in scala/pickling.

4.6 Object Identity and Sharing

Object identity enables the existence of complex object graphs, which themselves are a cornerstone of object-oriented programming. While in Section 6.7 we show that pickling *flat* object graphs is most common in big data applications, a general pickling framework for use with an object-oriented language must not only support flat object graphs, it must also support cyclic object graphs.

Supporting such cyclic object graphs in most object-oriented languages, however, typically requires sophisticated runtime support, which is known to incur a significant performance hit. This is due to the fact that pickling graphs with cycles requires tracking object identities at runtime, so that pickling terminates and unpickling can faithfully reconstruct the graph structure.

To avoid the overhead of tracking object identities unanimously for all objects, "runtime-based" serialization frameworks like Java or Kryo have to employ reflective/introspective checks to detect whether identities are relevant.[5]

Scala/pickling, on the other hand, employs a hybrid compile-time/runtime approach. This makes it possible to avoid the overhead of object identity tracking in cases where it is statically known to be safe, which we show in Section 6.7 is typically common in big data applications.

The following Section 4.6.1 outlines how object identity is tracked in scala/pickling. It also explains how the management of object identities enables a *sharing* optimization. This sharing optimization is especially important for persistent data structures, which are commonly used in Scala. Section 4.6.2 explains how compile-time analysis is used to reduce the amount of runtime checking in cases where object graphs are statically known to be acyclic.

4.6.1 Object Tracking

During pickling, a pickler keeps track of all objects that are part of the (top-level) object to-be-pickled in a table. Whenever an object that's part of the object graph is pickled, a hash code based on the identity of the object is computed. The pickler then looks up whether that object has already been pickled, in which case the table contains a unique integer ID as the entry's value. If the table does not contain an entry for the object, a unique ID is generated and inserted, and the object is pickled as usual. Otherwise, instead of pickling the object again, a special `Ref` object containing the integer ID is written to the pickle.[6] During unpickling, the above process is reversed by maintaining a mapping[7] from integer IDs to unpickled heap objects.

This approach to dealing with object identities also enables sharing, an optimization which in some big data applications can improve system throughput by reducing pickle size. Scala's immutable collections hierarchy is one example of a set of data structures which are persistent, which means they make use of sharing. That is, object subgraphs which occur in multiple instances of a data structure can be shared which is more efficient than maintaining multiple copies of those subgraphs.

Scala/pickling's management of object identities benefits instances of such data structures as follows. First, it reduces the size of the computed pickle, since instead of pickling the same object instance many times, compact references (`Ref` objects) are pickled. Second, pickling time also has the potential to be reduced, since shared objects have to be pickled only once.

[5] With Kryo, some of this overhead can be avoided when using custom, handwritten serializers.

[6] Several strategies exist to avoid preventing pickled objects from being garbage collected. Currently, for each top-level object to-be-pickled, a new hash table is created.

[7] This can be made very efficient by using a map implementation which is more efficient for integer-valued keys, such as a resizable array.

4.6.2 Static Object Graph Analysis

When generating a pickler for a given type `T`, the IR is analyzed to determine whether the graph of objects of type `T` may contain cycles. Both `T` and the types of `T`'s fields are examined using a breadth-first traversal. Certain types are immediately excluded from the traversal, since they cannot be part of a cycle. Examples are primitive types, like `Double`, as well as certain immutable reference types that are final, like `String`. However, the static inspection of the IR additionally allows scala/pickling to traverse sealed class hierarchies.

For example, consider this small class hierarchy:

```
final class Position(p: Person, title: String)
sealed class Person(name: String, age: Int)
final class Firefighter(name: String, age: Int, salary: Int)
  extends Person(name, age)
final class Teacher(name: String, age: Int, subject: String)
  extends Person(name, age)
```

In this case, upon generating the pickler for class `Position`, it is detected that no cycles are possible in the object graphs of instances of type `Position`. While `Position`'s `p` field has a reference type, it cannot induce cycles, since `Person` is a sealed class that has only final subclasses; furthermore, `Person` and its subclasses have only fields of primitive type.

In addition to this analysis, our framework allows users to disable all identity tracking programmatically (by importing an implicit value), in case it is known that the graphs of (all) pickled objects are acyclic. While this switch can boost performance, it also disables opportunities for sharing (see above), and may thus lead to larger "pickles".

5. Implementation

The presented framework has been fully implemented in Scala. The object-oriented pickler combinators presented in Section 3, including their implicit selection and composition, can be implemented using stable versions of the standard, open-source Scala distribution. The extension of our basic model with automatic pickler generation has been implemented using the experimental macros feature introduced in Scala 2.10.0. Macros can be thought of as a more regularly structured, localized, and more stable alternative to compiler plugins. To simplify tree generation, our implementation leverages a quasiquoting library for Scala's macros [33].

6. Experimental Evaluation

In this section we present first results of an experimental evaluation of our pickling framework. Our goals are

1. to evaluate the performance of automatically-generated picklers, analyzing the memory usage compared to other serialization frameworks, and
2. to provide a survey of the properties of data types that are commonly used in distributed computing frameworks and applications.

In the process, we are going to evaluate the performance of our framework alongside two popular and industrially-prominent serialization frameworks for the JVM, Java's native serialization, and Kryo.[8]

6.1 Experimental Setup

The following benchmarks were run on a MacBook Pro with a 2.6 GHz Intel Core i7 processor with 16 GB of memory running Mac OS X version 10.8.4 and Oracle's Java HotSpot(TM) 64-Bit Server VM version 1.6.0_51. In all cases we used the following configuration flags: `-XX:MaxPermSize=512m -XX:+CMSClassUnloadingEnabled -XX:ReservedCodeCacheSize=192m -XX:+UseConcMarkSweepGC -Xms512m -Xmx2g`. Each benchmark was run on a warmed-up JVM. The result shown is the median of 9 such "warm" runs.

6.2 Microbenchmark: Collections

In the first microbenchmark, we evaluate the performance of our framework when pickling standard collection types. We compare against three other serialization frameworks: Java's native serialization, Kryo, and a combinator library of naive handwritten pickler combinators. All benchmarks are compiled and run using a current milestone of Scala version 2.10.3.

The benchmark logic is very simple: an immutable collection of type `Vector[Int]` is created which is first pickled (or serialized) to a byte array, and then unpickled. While `List` is the prototypical collection type used in Scala, we ultimately chose `Vector` as Scala's standard `List` type could not be serialized out-of-the-box using Kryo,[9] because it is a recursive type in Scala. In order to use Scala's standard `List` type with Kryo, one must write a custom serializer, which would sidestep the objective of this benchmark, which is to compare the speed of *generated* picklers.

The results are shown in Figure 4 (a). As can be seen, Java is slower than the other frameworks. This is likely due to the expensive runtime cost of the JVM's calculation of the runtime transitive closure of the objects to be serialized. For 1,000,000 elements, Java finishes in 495ms while scala/pickling finishes in 74ms, or a factor 6.6 faster. As can be seen, the performance of our prototype is clearly faster than Kryo for small to moderate-sized collections; even though it remains faster throughout this benchmark, the gap between Kryo and scala/pickling shrinks for larger collections. For a `Vector[Int]` with 100,000 elements, Kryo v2 finishes in 36ms while scala/pickling finishes in 10ms–a factor of 3.6 in favor of scala/pickling. Conversely, for a Vector of 1,000,000 elements, Kryo finishes in 84ms whereas scala/pickling finishes in 74ms. This result clearly demonstrates the benefit of

[8] We select Kryo and Java because, like scala/pickling, they both are "automatic". That is, they require no schema or extra compilation phases, as is the case for other frameworks such as Apache Avro and Google's Protocol Buffers.

[9] We register each class with Kryo, an optional step that improves performance.

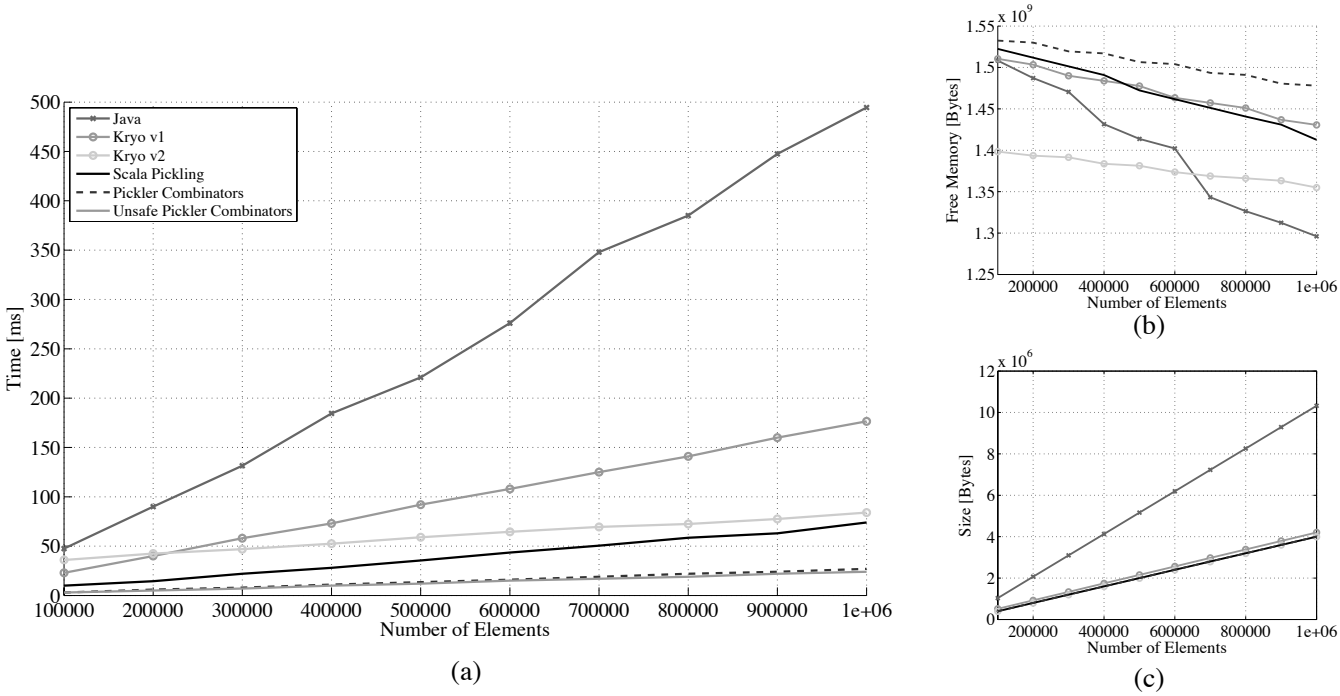

Figure 4: Results for pickling and unpickling an immutable `Vector[Int]` using different frameworks. Figure 4(a) shows the roundtrip pickle/unpickle time as the size of the `Vector` varies. Figure 4(b) shows the amount of free memory available during pickling/unpickling as the size of the `Vector` varies. Figure 4(c) shows the pickled size of `Vector`.

our hybrid compile-time/runtime approach: while scala/pickling has to incur the overhead of tracking object identity in the case of general object graphs, in this case, the compile-time pickler generation is able to detect that object identity does not have to be tracked for the pickled data types. Moreover, it is possible to provide a size hint to the pickle builder, enabling the use of a fixed-size array as the target for the pickled data. We have found that those two optimizations, which require the kind of static checking that scala/pickling is able to do, can lead to significant performance improvements. The performance of manually written pickler combinators, however, is still considerably better. This is likely due to the fact that pickler combinators require no runtime checks whatsoever– pickler combinators are defined per type, and manually composed, requiring no such check. In principle, it should be possible to generate code that is as fast as these pickler combinators in the case where static picklers can be generated.

Figure 4 (b) shows the corresponding memory usage; on the y-axis the value of `System.freeMemory` is shown. This plot reveals evidence of a key property of Kryo, namely (a) that its memory usage is quite high compared to other frameworks, and (b) that its serialization is stateful because of internal buffering. In fact, when preparing these benchmarks we had to manually adjust Kryo buffer sizes several times to avoid buffer overflows. It turns out the main reason for this is that Kryo reuses buffers whenever possible when serializing one object after the other. In many cases, the newly pickled object is simply appended at the current position in the existing buffer which results in unexpected buffer growth. Our framework does not do any buffering which makes its behavior very predictable, but does not necessarily maximize its performance.

Finally, Figure 4 (c) shows the relative sizes of the serialized data. For a `Vector[Int]` of 1,000,000 elements, Java required 10,322,966 bytes. As can be seen, all other frameworks perform on par with another, requiring about 40% of the size of Java's binary format. Or, in order of largest to smallest; Kryo v1 - 4,201,152 bytes; Kryo v2 - 4,088,570 bytes; scala/pickling 4,000,031 bytes; and Pickler Combinators 4,000,004 bytes.

6.3 Wikipedia: Cyclic Object Graphs

In the second benchmark, we evaluate the performance of our framework when pickling object graphs with cycles. Using real data from the Wikipedia project, the benchmark builds a graph where nodes are Wikipedia articles and edges are references between articles. In this benchmark we compare against Java's native serialization and Kryo. Our objective was to measure the full round-trip time (pickling and unpickling) for all frameworks. However, Kryo consistently crashed in the unpickling phase despite several work-around attempts. Thus, we include the results of two experiments: (1) "pickle only", and (2) "pickle and unpickle". The results

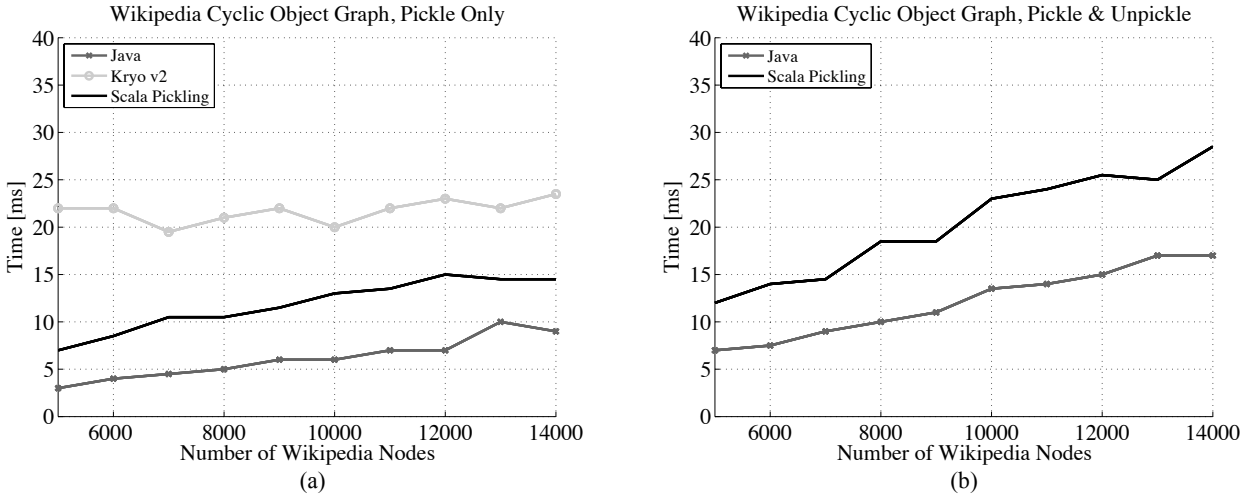

Figure 5: Results for pickling/unpickling a partition of Wikipedia, represented as a graph with many cycles. Figure 6(a) shows a "pickling" benchmark across scala/pickling, Kryo, and Java. In Figure 6(b), results for a roundtrip pickling/unpickling is shown. Here, Kryo is removed because it crashes during unpickling.

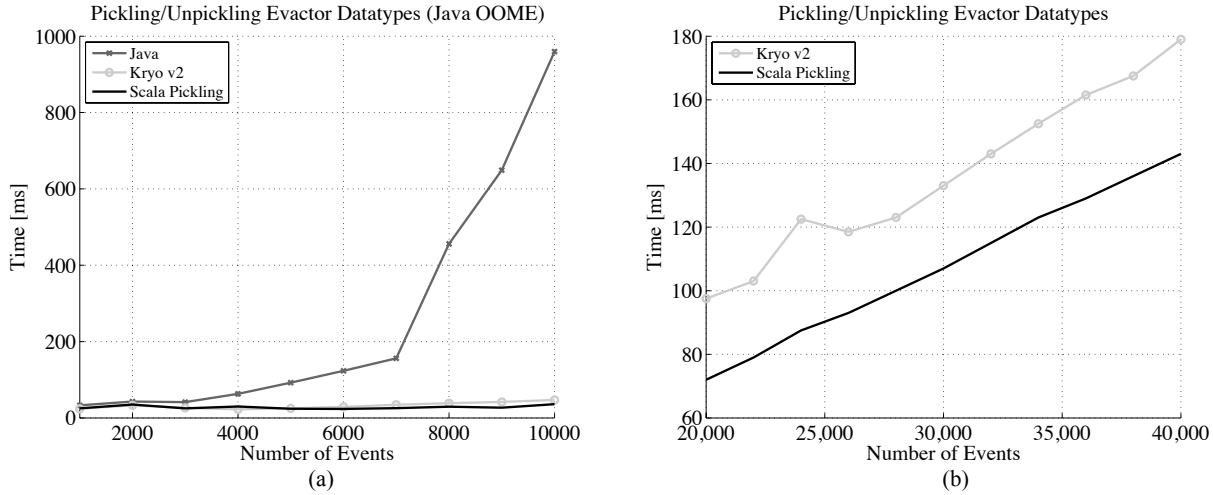

Figure 6: Results for pickling/unpickling evactor datatypes (numerous tiny messages represented as case classes containing primitive fields.) Figure 6(a) shows a benchmark which pickles/unpickles up to 10,000 evactor messages. Java runs out of memory at this point. Figure 6(b) removes Java and scales up the benchmark to more evactor events.

show that Java's native serialization performs particularly well in this benchmark. In the "pickle only" benchmark of Figure 5 between 12000 and 14000 nodes, Java takes only between 7ms and 10ms, whereas scala/pickling takes around 15ms. Kryo performs significantly worse, with a time between 22ms and 24ms. In the "pickle and unpickle" benchmark of Figure 5, the gap between Java and scala/pickling is similar to the "pickle only" case: Java takes between 15ms and 18ms, whereas scala/pickling takes between 25ms and 28ms.

6.4 Microbenchmark: Evactor

The Evactor benchmark evaluates the performance of pickling a large number of small objects (in this case, events exchanged by actors). The benchmark creates a large number of events using the datatypes of the Evactor complex event processor (see Section 6.4); all created events are inserted into a collection and then pickled, and finally unpickled. As the results in Figure 6 show, Java serialization struggles with extreme memory consumption and crashes with an out-of-memory error when a collection with more than 10000

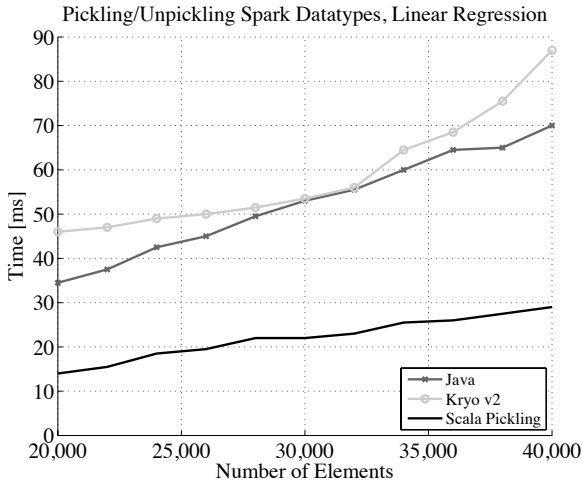

Figure 7: Results for pickling/unpickling data points from an implementation of linear regression using Spark.

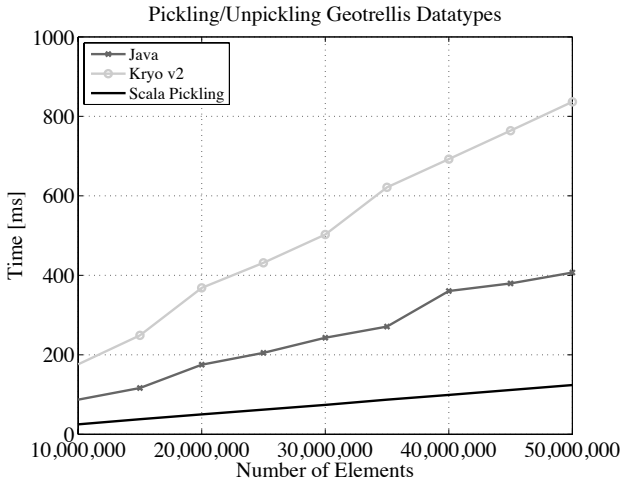

Figure 8: Results for pickling/unpickling geotrellis datatypes (case classes and large primitive arrays).

events is pickled. Both Kryo and scala/pickling handle this very high number of events without issue. To compare Kryo and scala/pickling more closely we did another experiment with an even higher number of events, this time leaving out Java. The results are shown on the right-hand side of Figure 6. At 40000 events, Kryo finishes after about 180ms, whereas scala/pickling finishes after about 144ms–a performance gain of about 25%.

6.5 Microbenchmark: Spark

Spark is a popular distributed in-memory collections abstraction for interactively manipulating big data. The Spark benchmark compares performance of scala/pickling, Java, and Kryo when pickling data types from Spark's implementation of linear regression.

Over the course of the benchmark, frameworks pickle and unpickle an `ArrayBuffer` of data points that each consist of a double and an accompanying `spark.util.Vector`, which is a specialized wrapper over an array of 10 `Doubles`. Here we use a mutable buffer as a container for data elements instead of more typical lists and vectors from Scala's standard library, because that's the data structure of choice for Spark to internally partition and represent its data.

The results are shown in Figure 7, with Java and Kryo running in comparable time and scala/pickling consistently outperforming both of them. For example, for a dataset of 40000 points, it takes Java 68ms and Kryo 86ms to perform a pickling/unpickling roundtrip, whereas scala/pickling completes in 28ms, a speedup of about 2.4x compared to Java and about 3.0x compared to Kryo.

6.6 Microbenchmark: GeoTrellis

GeoTrellis [4] is a geographic data processing engine for high performance applications used by the US federal government among others.

In this benchmark one of the main message classes used in GeoTrellis is pickled. The class is a simple case class containing a primitive array of integers (expected to be large). Figure 8 shows the time it takes to pickle and unpickle an instance of this case class varying the size of the contained array.

The plot shows that Java serialization performs, compared to Kryo, surprisingly well in this benchmark, *e.g.,* a roundtrip for 50000000 elements takes Java 406ms, whereas Kryo is more than two times slower at 836ms. It is likely that modern JVMs support arrays of primitive types well, which is the dominating factor in this case. Scala/pickling is still significantly faster with 124ms, since the static type of the array is final, so that efficient array-pickling code can be generated at compile time.

6.7 Data Types in Distributed Frameworks and Applications

Figure 9 shows a summary of the most important data types used in popular distributed computing frameworks like Spark [42] and Storm [20]. The fully shaded circles in the table representing "heavy use" means either (a) a feature is used frequently in application-level data types or (b) a feature is used frequently in data types that the framework registers with its underlying serialization system. Half-shaded circles in the table representing "light use" mean a feature is used only infrequently in the data types used in applications or registered by frameworks. We categorize the data types shown in this table into two groups.

In the first group at the top are distributed *applications* using data types suitable for distributed event processing and message passing. We consider two representative open-source applications: GeoTrellis and Evactor. Both applications use Akka [37], an event-driven middleware for distributed message passing. However, the properties of the

	primitives/ primitive arrays	value-like types	collections	case classes	type descriptor	generics	subtyping polymorphism
GeoTrellis (Akka)	●	○	○	●	○	○	○
Evactor (Akka)	●	◐	◐	●	○	○	◐
Spark	●	●	●	◐	○	○	○
Storm	○	●	●	N/A	○	○	◐
Twitter Chill	○	◐	●	◐	◐	◐	◐

Legend: ●: Heavy Use ◐: Light Use ○: No Use

Figure 9: Scala types used in industrial distributed frameworks and applications.

exchanged messages are markedly different. Messages in GeoTrellis typically contain large amounts of geographic raster data, stored in arrays of primitives. Messages in Evactor represent individual events which typically contain only a few values of primitive types. Both applications make use of Scala's case classes which are most commonly used as message types in actor-based applications.

The second group in the bottom half of Figure 9 consists of distributed computing frameworks. What this table suggests is that the majority of distributed computing frameworks and applications requires pickling collections of various types. Interestingly, application-level data types tend to use arrays with primitive element type; a sign that there is a great need to provide easier ways to process "big data" efficiently. From the table it is also clear that case classes tend to be primarily of interest to application code whereas frameworks like Spark tend to prefer the use of simple collections of primitive type internally. What's more, the demand for pickling generics seems to be lower than the need to support subtyping polymorphism (our framework supports both, though). At least in one case (Twitter's Chill [26]) a framework explicitly serializes manifests, type descriptors for Scala types, which are superceded by type tags. The shaded area (which groups "heavily-used" features across applications/frameworks) shows that collections are often used in distributed code, in particular with primitive element types. This motivates the choice of our collections micro benchmark.

7. Other Related Work

Pickling in programming languages has a long history dating back to CLU [15] and Modula-3 [6]. The most closely-related contemporary work is in two areas. First, pickling in object-oriented languages, for example, in Java (see the Java Object Serialization Specification [25]), in .NET, and in Python [38]; second, work on pickler combinators in functional languages which we have already discussed in the introduction. The main difference of our framework compared to pickling, or serialization, in widespread OO languages is that our approach does not require special support by the underlying runtime. In fact, the core concepts of object-oriented picklers as presented in this paper can be realized in most OO languages with generics.

While work on pickling is typically focused on finding optimally compact representations for data [39], not all work has focused only on distribution and persistence of ground values. Pickling has also been used to distribute and persist code to implement module systems [30, 32]. Similar to our approach, but in a non-OO context, AliceML's HOT pickles [31] are universal in the sense that any value can be pickled. While HOT pickles are deeply integrated into language and runtime, scala/pickling exists as a macro-based library, enabling further extensibility, *e.g.,* user-defined pickle formats can be interchanged.

There is a body of work on maximizing sharing of runtime data structures [2, 10, 36] which we believe could be applied to the pickler combinators presented in Section 3; however, a complete solution is beyond the scope of the present paper.

8. Conclusion and Future Work

We have introduced a model of pickler combinators which supports core concepts of object-oriented programming including subtyping polymorphism with open class hierarchies. Furthermore, we have shown how this model can be augmented by a composable mechanism for static pickler generation which is effective in reducing boilerplate and in ensuring efficient pickling. Thanks to a design akin to an object-oriented variation of type classes known from functional programming, the presented framework enables retrofitting existing types and third-party libraries with pickling support. Experiments suggest that static generation of pickler combinators can outperform state-of-the-art serialization frameworks and significantly reduce memory usage.

In future work we plan to further optimize the pickler generation and to extend the framework with support for closures.

Acknowledgments

We would like to thank the anonymous OOPSLA 2013 referees for their thorough reviews and helpful suggestions which greatly improved the quality of the paper. We are grateful to the artifact evaluation committee and the anonymous arti-

fact referees for their detailed reviews of scala/pickling. We would particularly like to thank Matei Zaharia for several helpful conversations which inspired this vein of work. Finally, we would like to thank Denys Shabalin for his work on quasiquotes for Scala which has helped simplify the code base of scala/pickling considerably.

References

[1] Apache. Avro®. http://avro.apache.org. Accessed: 2013-08-11.

[2] A. W. Appel and M. J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, Computer Science Department, 1993.

[3] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-independent storage for social computing applications. In *CIDR*, 2009.

[4] Azavea. GeoTrellis. http://www.azavea.com/products/geotrellis/, 2010. Accessed: 2013-08-11.

[5] E. Burmako and M. Odersky. Scala macros, a technical report. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.

[6] L. Cardelli, J. E. Donahue, M. J. Jordan, B. Kalsow, and G. Nelson. The modula-3 type system. In *POPL*, pages 202–212, 1989.

[7] B. Carpenter, G. Fox, S. H. Ko, and S. Lim. Object serialization for marshalling data in a Java interface to MPI. In *Java Grande*, pages 66–71, 1999.

[8] B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type classes as objects and implicits. In *OOPSLA*, pages 341–360, 2010.

[9] G. Dubochet. *Embedded Domain-Specific Languages using Libraries and Dynamic Metaprogramming*. PhD thesis, EPFL, Switzerland, 2011.

[10] M. Elsman. Type-specialized serialization with sharing. In *Trends in Functional Programming*, pages 47–62, 2005.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. 1993.

[12] J. Gil and I. Maman. Whiteoak: introducing structural typing into Java. In G. E. Harris, editor, *OOPSLA*, pages 73–90, 2008.

[13] Google. Protocol Buffers. https://code.google.com/p/protobuf/, 2008. Accessed: 2013-08-11.

[14] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In T. D'Hondt, editor, *ECOOP*, pages 354–378, 2010.

[15] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst*, 4(4): 527–551, 1982.

[16] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst*, 23(3):396–450, May 2001.

[17] A. Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6): 727–739, 2004.

[18] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An efficient implementation of Java's remote method invocation. In *PPOPP*, pages 173–182, Aug. 1999.

[19] J. P. Magalhães, A. Dijkstra, J. Jeuring, and A. Löh. A generic deriving mechanism for Haskell. In J. Gibbons, editor, *Haskell*, pages 37–48, 2010.

[20] Nathan Marz and James Xu and Jason Jackson et al. Storm. http://storm-project.net/, 2012. Accessed: 2013-08-11.

[21] Nathan Sweet et al. Kryo. https://code.google.com/p/kryo/. Accessed: 2013-08-11.

[22] K. Ng, M. Warren, P. Golde, and A. Hejlsberg. The Roslyn project: Exposing the C# and VB compiler's code analysis. http://msdn.microsoft.com/en-gb/hh500769, Sept. 2012. Accessed: 2013-08-11.

[23] M. Odersky. Scala Language Specification. http://www.scala-lang.org/files/archive/nightly/pdfs/ScalaReference.pdf, 2013. Accessed: 2013-08-11.

[24] M. Odersky and M. Zenger. Scalable component abstractions. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 41–57, 2005.

[25] Oracle, Inc. Java Object Serialization Specification. http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html, 2011. Accessed: 2013-08-11.

[26] Oscar Boykin and Mike Gagnon and Sam Ritchie. Twitter Chill. https://github.com/twitter/chill, 2012. Accessed: 2013-08-11.

[27] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency - Practice and Experience*, 12(7):495–518, 2000.

[28] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[29] G. D. Reis and B. Stroustrup. Specifying C++ concepts. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 295–308, 2006.

[30] A. Rossberg. *Typed open programming: a higher-order, typed approach to dynamic modularity and distribution*. PhD thesis, Saarland University, 2007.

[31] A. Rossberg, G. Tack, and L. Kornstaedt. Status report: HOT pickles, and how to serve them. In *ML*, pages 25–36, 2007.

[32] P. V. Roy. Announcing the mozart programming system. *SIGPLAN Notices*, 34(4):33–34, 1999.

[33] D. Shabalin, E. Burmako, and M. Odersky. Quasiquotes for Scala. Technical Report EPFL-REPORT-185242, EPFL, Switzerland, 2013.

[34] K. Skalski. Syntax-extending and type-reflecting macros in an object-oriented language. Master's thesis, University of Warsaw, Poland, 2005.

[35] R. Strnisa, P. Sewell, and M. J. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514, 2007.

[36] G. Tack, L. Kornstaedt, and G. Smolka. Generic pickling and minimization. *Electr. Notes Theor. Comput. Sci*, 148(2):79–103, 2006.

[37] Typesafe. Akka. http://akka.io/, 2009. Accessed: 2013-08-11.

[38] G. van Rossum. Python programming language. In *USENIX Annual Technical Conference*. USENIX, 2007.

[39] D. Vytiniotis and A. J. Kennedy. Functional pearl: every bit counts. *SIGPLAN Not.*, 45(9):15–26, Sept. 2010.

[40] S. Wehr and P. Thiemann. JavaGI: The interaction of type classes with interfaces and inheritance. *ACM Trans. Program. Lang. Syst*, 33(4):12, 2011.

[41] M. Welsh and D. E. Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency - Practice and Experience*, 12(7), 2000.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX, 2012.