Object-Oriented Pickler Combinators

and an Extensible Generation Framework

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

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

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

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

Abstract

Serialization or pickling, i.e., persisting runtime objects by converting them into a binary or text representation is ubiquitous in distributed programming. Pickler combinators are a popular approach from functional programming designed to alleviate 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 Java-based serialization frameworks tend to be tied to a specific pickle format, leaving programmers 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, designed to be the default serialization mechanism of the Scala programming language. Our framework is extensible; (1) using Scala's implicit parameters, users can add their own easily-swappable pickle format, (2) using the type class pattern, users can provide their own custom picklers to override the default behavior of the Scala pickling framework. In addition to extensibility and need for little to no boilerplate, the static generation of our OO picklers achieves a factor 6 speedup over Java Serialization, and performs on par or up to 3 times faster than popular "fast" Java serialization frameworks like Kryo.

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 interop 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 suprisingly 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. On the JVM, serialization has long been acknowledged as having a high overhead [7, 35], 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% [16, 24].

Due to the prohibitive cost of using Java Serialization in high-performance distributed applications, many frameworks for distributed computing, like Akka [32], Spark [36], SCADS [3], and others, provide support for higher-performance alternative frameworks such as Google's Protobuf [13], Apache Avro [1], or Kryo [19]. 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 a framework for generation either at runtime or at compile time. The introduced notion of object-oriented pickler combinators extends pickler combinators known from functional programming [15] with support for object-oriented concepts such as subtyping and mix-in composition. 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].

What's more, 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 [23] requires only little boilerplate, which mainstream Java developers have come to expect. Our framework aims to be useable 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, frameworks such as the latter 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 3rd 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.

1.1 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, 15] 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 [34] or C++ Concepts [25].

What's more, 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 [17]. 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 8.

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 and open class hierarchies.

- 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.

2. Background

The design and implementation of our pickling framework leverages several advanced features of the Scala programming language. This section introduces everything that is required to understand the rest of the paper. Apart from this section the reader is expected to be familiar with a typical statically-typed, class-based object-oriented programming language such as Java or C#.

2.1 Implicits

Implicit Parameters. In Scala, it is possible to select values automatically based on type. These capabilities are enabled when using the `implicit` keyword. For example, a method `log` with multiple parameter lists may annotate their last parameter list using the `implicit` keyword.[2]

```
def log(msg: String)(implicit o: PrintStream) =
  o.println(msg)
```

This means that in an invocation of `log`, 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.

```
implicit val out = System.out
log("Does not compute!")
```

In the above example, the implicit val `out` is in the implicit scope of the invocation of `log`; since it has the right type, it is automatically selected as an implicit argument.

Implicit Conversions. Implicit conversions can be thought of as methods which, like implicit parameters, can be implicitly selected (*i.e.,* invoked) based upon their type, and whether or not they are present in implicit scope. As with implicit parameters, implicit conversions also carry the `implicit` keyword before their declaration.

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

[2] Example taken from [8].

```
implicit def intWrapper(x: Int): Message =
  new Message {
    def message: String = "secret message!"
  }
```

In the example above, assuming there exists an abstract class `Message` with abstract method `message`, the implicit conversion `intWrapper` will be triggered when a method called `message` is called on an `Int`. That is, simply calling `39.message` will result in "secret message!" being returned. Since the implicit conversion has the effect of adding a "new" method to type `Int`, `message` is typically called an *extension method*. In our framework we use implicit conversions, for example, for adding a `pickle` method to arbitrary objects.

2.2 Reflection

Reflection is the ability of a program to inspect, and possibly even modify itself at runtime. Before Scala 2.10, Scala did not have any reflection capabilities of its own. Instead, one could use Java reflection which provided basic but limited runtime reflection capabilities. In Scala 2.10, a new reflection library was introduced not only to address the shortcomings of Java's runtime reflection on Scala-specific and generic types, but to also add a more powerful toolbox of general reflective capabilities to Scala. Along with full-featured runtime reflection for Scala types and generics, Scala 2.10 also ships with compile-time reflection capabilities, in the form of macros (covered in Section 2.3), as well as the ability to reify Scala expressions into abstract syntax trees.

TypeTags. One aspect of runtime reflection that was introduced in Scala 2.10 is the notion of `TypeTags`. As with other JVM languages, Scala's types are erased at compile time. `TypeTags` can be thought of as objects which carry along all type information available at compile time, to runtime. As we will see, `TypeTags` will prove to be invaluable in situations where precise type information would otherwise not be available at runtime.

Unified Runtime/Compile-time Reflection API. Another important aspect of Scala's reflection library is the one-to-one correspondence between Scala Reflection's compile-time (*i.e.,* macros) and runtime APIs. Each API is parameterized on a so-called `Universe`, an object which serves as the entry point to Scala reflection, and which provides all principal concepts used in reflection, such as `Types`, `Trees`, and `Annotations`. Depending on the task at hand, the choice between runtime and compile-time reflection is as easy as selecting either a compile-time or a runtime `Universe`. As we will see, this enables maximum code reuse in that a fallback runtime pickler generation mechanism can be achieved by simply reusing the code for static generation, and parameterizing it on a runtime `Universe`.

2.3 Macros

Scala reflection enables a form of metaprogramming which makes it possible for programs to modify themselves at compile-time. This compile-time reflection is realized in the form of hygenic macros [5], which *expand* at compile-time to manipulate abstract syntax trees (ASTs). In our framework, we make use of two principal types of macros.

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, the parameters of `assert_impl` are syntax trees, which the body of `assert_impl` will operate on, itself returning an AST of type `Expr[Unit]`. It is `assert_impl` which is expanded and evaluated at compile-time, its result is then inlined at the callsite of `assert` and the inlined result is type-checked It is also important to note that implicit defs as described earlier in Section 2.1 can be implemented as macros.

As we will see, these macros defs, coupled with implicits in Scala enable the boilerplate-free usage of the Scala pickling framework at the pickling use- site.

Macro Annotations. Unlike macro defs, macro annotations are capable of *adding members* to classes which carry their annotation.

```
@withNewToString
class D { ... }
```

The `withNewToString` annotation is defined using a standard class definition by extending a special `MacroAnnotation` marker trait, and by implementing a special `transform` method as a macro:

```
class withNewToString extends MacroAnnotation {
  def transform = macro transform_impl
  def transform_impl = { ... }
}
```

The `transform` macro implementation is passed the AST of the annotated class definition (the AST of "`class D { ... }`"), and returns a possibly changed AST as the new class definition (which could have added members, changed constructor parameters etc.)

3. Overview and Usage

3.1 Basic Usage

The Scala Pickling framework 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 pickl = obj.pickle
val obj2 = pickl.unpickle[Obj]
```

Here, the `import` statement imports the Scala Pickling framework, the method `pickle` triggers static pickler generation, and the method `unpickle` triggers static unpickler generation. Note that not every type has a `pickle` method; it is implemented as an extension method using implicits (Section 2.1), imported into scope as a member of the `scala.pickling` package.

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 5.3.1).

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, resulting in high performance (see Section 7).

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.

Note that in the above example, the `unpickle` method is parameterized on `obj`'s precise type `Obj`. Using the Scala Pickling framework, 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,

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

val ff = new Firefighter("Jim", 2005)
val pickl = ff.pickle[Person]
val ff2 = pickl.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 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.

The Scala Pickling framework 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:

```
val unpickld = pickl.unpickle[Any] match {
  case Firefighter(name, since) => ...
  case x: Int => ...
  ...
}
```

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.

3.2 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 higher performance, the Scala Pickling framework 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 which case, we call the method that returns the pickler specialized for that runtime class. If the class/trait has been annotated, the returned pickler has been generated statically (when the annotation, implemented using a macro, is expanded in each subclass, transitively).

This @`pickelable` 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.

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 4.

4. Model of Object-Oriented Picklers

In this section we provide a formal definition of object-oriented picklers. We introduce picklers as first-class objects, and provide their type definitions and contracts that valid implementations must guarantee. Subsequently, we demonstrate that the introduced picklers enable modular, object-oriented pickler combinators, *i.e.,* methods for composing more complex picklers from simpler primitive picklers.

Note that we are using a Scala-like program notation. However, the introduced concepts and definitions are realizable in most statically-typed OO languages with generics.

Definition 4.1. (First-Class Picklers and Unpicklers)

A pickler for some type T is an instance of one of two abstract class or interface types `DPickler`$[T]$ or `SPickler`$[T]$. Each have an abstract method `pickle` with a single parameter of type T and return type `Pickle`:

```
trait DPickler[T] {
  def pickle(obj: T): Pickle
}
```

(The type definition of `SPickler`$[T]$ differs only in the name of the type; below we define the difference in their contracts.)

Conversely, an unpickler for some type U is an instance of an abstract class or interface type `Unpickler`$[U]$ that has a single abstract method `unpickle` with a single parameter of type `Pickle` and return type `Unit`.

The pickle method takes an object to-be-pickled of static type T, pickles it by turning it into some external representation like a byte array, and returns an instance of type `Pickle` which wraps the external representation. 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].

Preliminary Definitions. To precisely specify the contracts of picklers and unpicklers, we require standard definitions of the dynamic type of an object `o`, written *dynTypeOf(*`o`*)*, and the erasure of a static type T, written *erasure(T)*.

Definition 4.2. (Structural Equality)
Two objects $\texttt{obj}_1$ and $\texttt{obj}_2$ are structurally equal, written

$\texttt{obj}_1 \equiv \texttt{obj}_2$

if and only if

- *dynTypeOf(*$\texttt{obj}_1$*)* `=:=` *dynTypeOf(*$\texttt{obj}_2$*)* `=:=` C for some class type C, and we have that for all $\texttt{fld} \in$ *fields*(C). $\texttt{obj}_1\texttt{.fld} \equiv \texttt{obj}_2\texttt{.fld}$, or
- *dynTypeOf(*$\texttt{obj}_1$*)* `=:=` T for some primitive type T, and $\texttt{obj}_1$ `==` $\texttt{obj}_2$.

The contracts of `DPickler`s, `SPickler`s, and `Unpickler`s are defined as follows.

Definition 4.3. (Object-Oriented Picklers)
Given the following typed objects,

  dp: `DPickler`[T]
  sp: `SPickler`[T]
 obj: T
  up: `Unpickler`[U] where $T <: U$
obj': U

where `obj` is the object to-be-pickled, and `obj'` is the unpickled object. Then,

`up.unpickle(dp.pickle(obj))` evaluates to `obj'`

`up.unpickle(sp.pickle(obj))` evaluates to `obj'` if *dynTypeOf*(`obj`) `=:=` *erasure*(`T`)

such that,

`obj': U`

`obj'` $\equiv$ `obj`

Note that `SPickler`'s `pickle` method has a precondition which requires the dynamic type of the object to-be-pickled to be equal to the erasure of its static type T. This means that an `SPickler`[T] is not guaranteed to pickle any object of a subtype of T. Because of this restriction we refer to instances of `SPickler` as *static picklers*. In contrast, a `DPickler`[T] pickles any object of type T. Therefore, we refer to instances of `DPickler` as *dynamic picklers*.

In the following section, we motivate the distinction between static and dynamic picklers.

4.1 Modular Pickler Combinators

Picklers as first-class objects make it possible to define pickler combinators, which in turn make it possible to build compound picklers from primitive picklers. However, only using the interface introduced above would not allow us to achieve this, since calling into any of the component picklers would produce a completed pickle; we would be left with the problem of composing completed pickles which is inefficient in general. Since one of our goals is high performance, we'll instead allow picklers to produce *partial pickles* which themselves are essentially builders [11] that multiple picklers can output to. The corresponding method has the following signature:

```
def pickle(obj: T, builder: PickleBuilder): Unit
```

`PickleBuilder` has methods to incrementally add elements of an object to-be-pickled to a pickle that is being constructed (see Section 5.3.1). When all elements have been added to a `PickleBuilder`, calling `result` returns the completed pickle.

For 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)
```

Modular pickler combinators would enable the composition of the desired pickler for type `Position` from picklers for types `String` and `Person`. 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.

4.2 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 [23, 24]. 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 5.4 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 [20] allows augmenting class definitions based on the presence of annotations.

- Static meta programming [5, 29] enables generation of picklers at compile time. In Section 5 we present an approach for generating object-oriented picklers from regular (class) type definitions.

4.3 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`.

4.4 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. Finally, we argue that object-oriented picklers should be invariant in their generic type parameter to allow for several scenarios of unanticipated evolution.

5. Generating Object-Oriented Picklers

An explicit goal of our framework is to require little to no boilerplate in client code, since practitioners are typically used 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.

What's more, 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 7 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.

5.1 Overview

Our framework generates type-specialized, object-oriented picklers using compile-time meta programming in the form of *macros*. Basically, 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 3). As a result, such an implicit `TypeTag`[2] 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. [3] This model of inheritance is central to the generation framework, and is formally defined in the following Section 5.2

5.2 Formal Model of Inheritance

The goal of this section is to formally define the representation, or IR, 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. 5.1).

Definition 5.1. (Elements of IR)

We define the syntax of values of the IR types.

$$\begin{aligned} F &::= \overline{(f_n, T)} \\ IR &::= (T, IR_{opt}, F) \\ IR_{opt} &::= \epsilon \mid IR \end{aligned}$$

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.

IR represents the pickling information for a class or some other object type. That is, an IR 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.

IR_{opt} is an optional IR; a missing IR is represented using ϵ.

Using the definitions of the elements of an IR from Definition 5.1, we go on to define a number of useful IR combinators, which form the basis of our model of inheritance.

Definition 5.2. (IR Combinators)

We begin by defining the types of our combinators before we define the combinators themselves.

Type Definitions

$$\begin{aligned} \mathit{concat} &: (F, F) \Rightarrow F \\ \mathit{extended} &: (IR, IR) \Rightarrow IR \\ \mathit{linearization} &: T \Rightarrow \overline{T} \\ \mathit{superIRs} &: T \Rightarrow \overline{IR} \\ \mathit{compose} &: IR \Rightarrow IR \\ \mathit{flatten} &: IR \Rightarrow IR \end{aligned}$$

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.[1]

Function Definitions

$$\mathit{concat}(\overline{f}, \overline{g}) = \overline{f}, \overline{g}$$

$$\mathit{extended}(C, D) = (T, C, \mathit{fields}(T)) \quad \text{where } D = (T, \_, \_) \ \wedge T <: C.1$$

$$\mathit{superIRs}(T) = [(S, \epsilon, \mathit{fields}(S)) \mid S \in \mathit{linearization}(T)]$$

$$\mathit{compose}(C) = \mathit{reduce}(\mathit{superIRs}(C.1), \mathit{extended})$$

$$\mathit{flatten}(C) = \begin{cases} (C.1, C.2, \mathit{concat}(C.3, \mathit{flatten(C.2).3})), \\ \quad \text{if } C.2 \neq \epsilon \\ C, \ \text{otherwise} \end{cases}$$

The function *concat* takes two arguments that are each sequences. 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.

Continued on next page...

[1] For example, in Scala the linearization is defined for classes mixing in multiple traits [21, 22]; in Java, the linearization function would simply return the chain of superclasses, not including the implemented interfaces.

[2] 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, see Section 2.

[3] 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. (IR Combinators) – Continued

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

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 $IR\ C$ for a type $C.1$ and returns a new IR for type $C.1$ which is the composition of the IRs of all supertypes of $C.1$. The resulting IR is a chain of super IRs according to the linearization order of $C.1$.

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

5.3 Pickler Generation Algorithm

The pickler generation is driven by the IR (see Section 5.2) 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, *erasure*$(T) =:= S$. In the second step, we explain how to extend the generation to dynamic picklers which do not require this assumption.

5.3.1 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 the Scala Pickling framework 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.

5.3.2 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:

[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.

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]`, 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 string-typed field reuses 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 5.1. (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.

5.3.3 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 $S1, \ldots, Sn$ 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 `HasPickler` interface trait. Consequently, a more optimal dispatch can be generated in this case:

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

5.4 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 3.2. 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 7 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 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.

5.5 Generics

Subclassing and generics: The combination of subclassing and generics poses a similar problem to that introduced above in Section 5.4. 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.

6. Implementation

The presented framework has been fully implemented in Scala.[5] The object-oriented pickler combinators presented in Section 4, including their implicit selection and composition, can be implemented using stable versions of the standard, open- source Scala distribution. The component that extends our basic model with automatic pickler generation has been implemented using the experimental macros feature introduced in Scala 2.10.0 (January 2013). Macros are being used as a more regularly structured and more stable alternative to compiler plugins. To simplify tree generation, our implementation leverages a quasiquoting library for Scala's macros. [28].

7. 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 compared to manually-written picklers, analyzing the memory usage compared to other serialization frameworks,
2. to provide a survey of the properties of data types that are commonly used in distributed computing frameworks and applications, and

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.[6]

7.1 Experimental Setup

The following benchmarks were run on a MacBook Pro with a 2.7 GHz Intel Core i7 processor with 16 GB of memory running Mac OS X version 10.8.2 and Oracle's Java HotSpot(TM) 64-Bit Server VM version 1.6.0_43. In all cases we used the following configuration flags: `-Xms1536M -Xmx4096M -Xss2M -XX:MaxPermSize=512M -XX:+UseParallelGC`. Each benchmark was run on a warmed-up JVM. The result shown is the median of 9 such "warm" runs.

7.2 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 Scala version 2.9.3, except for the present framework which depends on an experimental macro system, as well as a couple of minor bug fixes that exist only in a current development version of Scala.

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 protypical collection type used in Scala, we ultimately chose Vector as Scala's standard List type could not be serialized out-of-the-box[7] using Kryo, because it is a recursive type in Scala. In order to use Scala's standard List type

[5] A preview of the pickling framework is made available by the authors upon request.

[6] We select Kryo and Java because, like the Scala Pickling framework, 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 Protobuf.

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

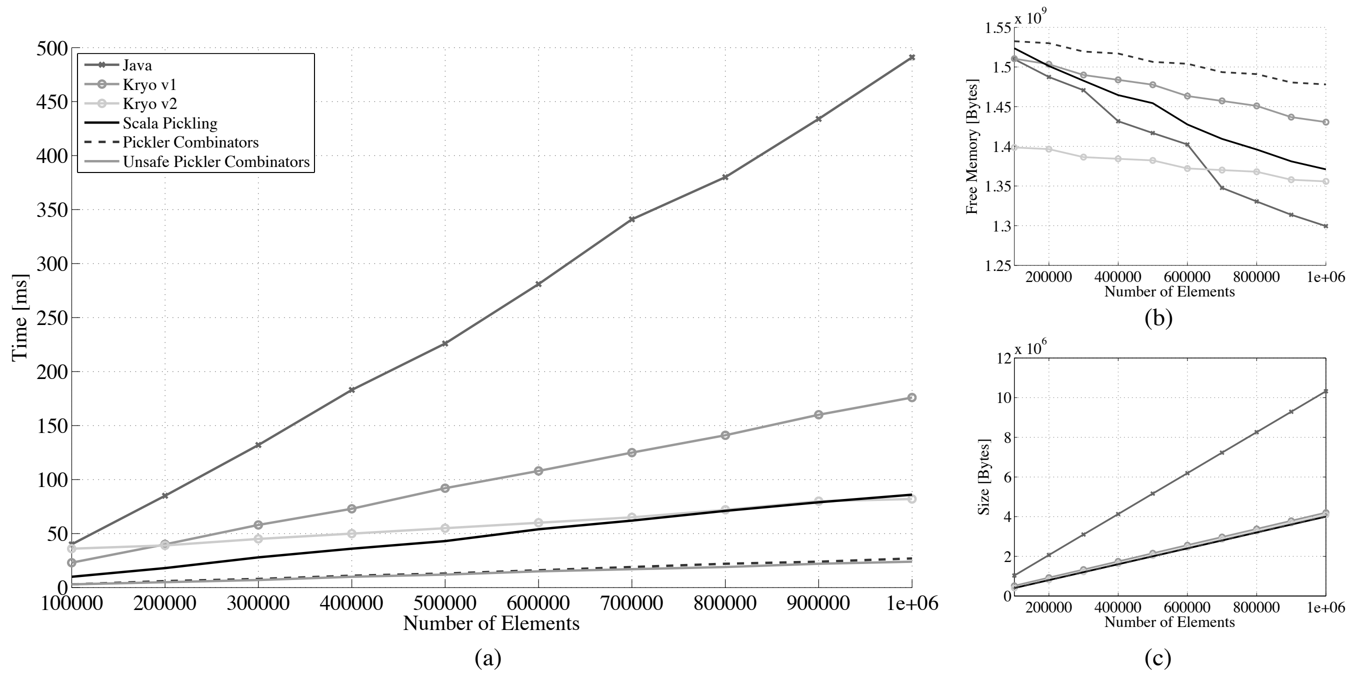

Figure 1. Results for pickling and unpickling an immutable `Vector[Int]` using different frameworks.

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 1 (a). As can be seen, Java is markedly slower than any other framework. 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 491ms while Scala Pickling finishes in 86ms, or a factor 5.7 faster. As can be seen, the performance of our prototype is faster than Kryo for small to moderate-sized collections, and is on par with Kryo or even slightly slower for large collections. For a Vector with 100,000 elements, Kryo 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 82ms whereas Scala Pickling finishes in 86ms. 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– picklers 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[8].

Figure 1 (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 append 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 1 (c) shows the relative sizes of the serialized data. For a Vector 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.

7.3 Data Types in Distributed Frameworks and Applications

Figure 2 shows a summary of the most important data types used in popular distributed computing frameworks like Spark [36] and Storm [18]. We categorize the data types 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-

[8] As a framework intended for eventual production use, we are actively seeking to bring performance closer to that of fully-static hand-composed pickler combinators. Towards that aim, we maintain a full benchmark suite at `http://lampwww.epfl.ch/~hmiller/pickling`

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

Figure 2. Scala types used in industrial distributed frameworks and applications.

source applications: GeoTrellis [4] is a geographic data processing engine for high performance applications used by the US federal government among others. Evactor is a complex event processor based on actors. Both applications use Akka [32], an event-driven middleware for distributed message passing. However, the properties of the 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 2 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 [31]) a framework explicitly serializes Manifests, type descriptors for Scala types, which are superceded by TypeTags (see Section 2.2) The shaded area (which is "heavily-used") shows that collections are often used in distributed code, in particular with primitive element types. This motivates the choice of our collections micro benchmark.

8. Other Related Work

Pickling in programming languages has a long history dating back to CLU [14] 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 [23]), in .NET, and in Python [33]; 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 wide-spread 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.

Pickling has been used not only for distribution and persistence of ground values, but also of code to implement module systems [26, 27]. There is a body of work on maximizing sharing of runtime data structures [2, 10, 30] which we believe could be applied to the pickler combinators presented in Section 4; however, a complete solution is beyond the scope of the present paper.

9. 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.

References

[1] Apache. Apache Avro. `http://avro.apache.org`, 2009.

[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/geotrellis/`, 2010.

[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. ACM Press, 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] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[13] Google. Protobuf. `https://code.google.com/p/protobuf`, 2008.

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

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

[16] 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.

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

[18] Nathan Marz and James Xu and Jason Jackson et al. Storm. `http://storm-project.net/`, 2012.

[19] Nathan Sweet. Kryo. `https://code.google.com/p/kryo`.

[20] 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.

[21] M. Odersky. Scala language specification. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`, 2011.

[22] M. Odersky and M. Zenger. Scalable component abstractions. *ACM SIGPLAN Notices*, 40(10):41–57, Oct. 2005.

[23] Oracle, Inc. Java Object Serialization Specification. `http://docs.oracle.com/javase/7/docs/platform/\\ serialization/spec/serialTOC.html`, 2011.

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

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

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

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

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

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

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

[31] Twitter. Chill. `https://github.com/twitter/chill`, 2012.

[32] Typesafe, Inc. Akka. `http://akka.io`, 2009.

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

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

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

[36] 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.