ScalaPy: Seamless Python Interoperability for Cross-Platform Scala Programs

Shadaj Laddad
University of California, Berkeley
USA
shadaj@berkeley.edu

Koushik Sen
University of California, Berkeley
USA
ksen@cs.berkeley.edu

Abstract

In recent years, Python has become the language of choice for data scientists with its many high-quality scientific libraries and Scala has become the go-to language for big data systems. In this paper, we bridge these languages with ScalaPy, a system for interoperability between Scala and Python. With ScalaPy, developers can use Python libraries in Scala by treating Python values as Scala objects and exposing Scala values to Python. ScalaPy supports both Scala on the JVM and Scala Native, enabling its usage from data experiments in interactive notebook environments to performance-critical production systems. In this paper, we explore the challenges involved with mixing the semantics and implementations of these two disparate languages.

CCS Concepts: • **Software and its engineering** → **Language features**.

Keywords: language interoperability, Scala, Python

ACM Reference Format:
Shadaj Laddad and Koushik Sen. 2020. ScalaPy: Seamless Python Interoperability for Cross-Platform Scala Programs. In *Proceedings of the 11th ACM SIGPLAN International Scala Symposium (SCALA '20), November 13, 2020, Virtual, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3426426.3428485

1 Introduction

Today, Python is the dominant language for data science with a plethora of machine learning and scientific computing libraries. Scala, on the other hand is the dominant language for big data processing and is widely used across the industry in production systems through platforms like Spark. With machine learning and big data analytics becoming critical components of modern products, developers often find themselves switching frequently between the two. After data scientists experiment with data models in Python, software developers must rewrite these models in Scala for production use. What if we could bridge the gap with a common language for both research and production?

ScalaPy is an open-source project (https://scalapy.dev) that brings the worlds of Python and Scala with a seamless interoperability layer that works on both the JVM and Scala Native [16]. By embedding a Python interpreter inside the Scala application, ScalaPy makes it easy for developers to bring Python libraries into existing production code without having to change their deployment systems. In addition, by offering the same end-user API regardless of target platform, ScalaPy is able to fit into all types of workflows from initial explorations inside Jupyter notebooks to production execution with low interoperability overhead in native code.

1.1 Motivating Example

Consider an application where we want to extract text from handwritten documents that users upload to our website (adapted from the TensorFlow beginner tutorial [6]).

Users start by uploading samples of their handwriting that our application will use to train a neural network. The resulting model will then evaluate new images to extract text data. TensorFlow is great for this application, but our server is written in Scala. While libraries exist that bind to TensorFlow's C API [13], they do not cover all the features available since a large portion of TensorFlow is implemented in Python. ScalaPy solves this problem in a practical way by making the Python interface accessible in Scala.

Figures 1, 2, and 3 show ScalaPy code that uses TensorFlow for this application.

```scala
// image, label tuples
val userSamples: Seq[(Seq[Int], Int)] = ...

val tf = py.module("tensorflow")
val trainData = userSamples
  .map(_._1.toPythonProxy).toPythonProxy
val trainLabels = userSamples
  .map(_._2).toPythonProxy
```

Figure 1. Loading TensorFlow and user data in ScalaPy

SCALA '20, November 13, 2020, Virtual, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8177-2/20/11.
https://doi.org/10.1145/3426426.3428485

We start by importing TensorFlow (line 4). Then, we load user data into Python by converting our Scala collections into Python sequences (lines 5-8). With our training data loaded, we are ready to train a machine learning model. In Figure 2, we define our model architecture by composing four layers using the TensorFlow Keras API (lines 1-10). We define a loss function for our categorical data (lines 12-15), configure our optimization strategy (lines 17-21), and train on the data (line 23).

```
1  val model = tf.keras.models.Sequential(Seq(
2    tf.keras.layers.Flatten(
3      input_shape = (28, 28)
4    ),
5    tf.keras.layers.Dense(128, "relu"),
6    tf.keras.layers.Dropout(0.2),
7    tf.keras.layers.Dense(10)
8  ).toPythonProxy)
9
10 val lossFn = tf.keras.losses.
11   SparseCategoricalCrossentropy(
12     from_logits = true
13   )
14
15 model.compile(
16   optimizer = "adam", loss = lossFn,
17   metrics = Seq("accuracy").toPythonProxy
18 )
19
20 model.fit(trainData, trainLabels, epochs = 5)
```

Figure 2. Creating a neural network in TensorFlow and training on user data

Finally, when a user submits new images for classification, we make predictions using our trained model. We do this by passing our input through the neural network (line 2), extracting probabilities and converting to a Scala collection (lines 3-4), and returning the index of the class with the highest probability (line 6).

```
1  def predict(input: Seq[Int]) = {
2    val pred = model(input.toPythonProxy).numpy
3    val probs = tf.nn.softmax(pred).numpy
4      .as[Seq[Double]]
5
6    probs.zipWithIndex.maxBy(_._1)._2
7  }
```

Figure 3. Making predictions on new user inputs with our model

Ignoring the toPythonProxy calls, using the TensorFlow Python API through ScalaPy feels no different than using a regular Scala library. With ScalaPy, Scala inherits a world of high-quality libraries from the Python ecosystem naturally.

1.2 Contributions

- We demonstrate how the Python interpreter can be embedded in Scala with simultaneous support for both the JVM and native platforms through Scala Native and explore the challenges of this embedding such as heap synchronization and thread safety (Section 2).
- We explain how primitives and complex constructs such as lambdas from Scala and Python can be converted to each other without requiring user intervention (Section 3).
- We describe a system of macro transformations that allow users to interact with Python APIs through type-safe definitions while still maintaining portability across the JVM and Scala Native (Section 4).
- We present multiple strategies for converting and sharing large collections of data between Scala and Python that allow users to tune the interoperability overhead for the intended usage (Section 5).

2 Embedding Python in Scala

ScalaPy binds to native APIs of the official CPython interpreter while exposing a high-level API for end users. Compared to interoperability projects such as Scala.js [2] that compile Scala to the target language, ScalaPy takes the approach of embedding an interpreter of the target language. This makes it easier for developers to bring Python into existing programs, but introduces a challenge since ScalaPy needs to support both the JVM and native platforms without introducing code duplication and risking differing behavior.

Using an embedded interpreter in a JVM-like host introduces challenges not seen in other interoperability work, such as the Lua-ML project [15] where the host's garbage collector can be repurposed for the embedded language. Embedding Python's runtime in Scala results in two independent heaps whose garbage collectors must stay synchronized since Scala code can reference Python values and vice-versa. On top of this, Python has a fundamentally different approach to threading compared to Scala, with a global interpreter lock which must be held when performing any operation. ScalaPy tackles these challenges to ensure that developers can be productive without worrying about interoperability details.

2.1 Cross-Platform Interpreter Bindings

Before we can offer a high-level API for our users, we need a low-level system to interact with the interpreter through Scala code. CPython, the standard implementation of Python, exposes all the features of the core language through a C API [14]. These APIs allow us to call Python functions, convert primitive C values into their Python equivalents, and adjust reference counts for heap synchronization.

ScalaPy supports both the JVM and Scala Native, but these platforms expose different APIs for calling into native code. If we were to rewrite our interpreter interface for each platform, we risk code duplication that can lead to diverging behavior. In ScalaPy, the majority of logic for interfacing with the CPython interpreter lies in shared code that is cross-compiled for the JVM and Scala Native, with just a thin binding abstraction that provides a common API over the equivalent C-level constructs in each platform.

In Scala Native, ScalaPy uses the built-in support for binding to C APIs through annotated Scala functions. Since Scala Native compiles directly to native code, we can access C-level constructs such as structs and pointers just like regular Scala objects. Holding a reference to Python values is as simple as storing a pointer to the value in a wrapping Scala object and maintaining reference counts appropriately. In our bindings for Scala Native in Figure 4, we also use zones to perform heap allocation, with an implicit Zone value introduced by the WithZone higher order function.

```
1  import scala.scalanative.native._
2  object Platform {
3    type Pointer = Ptr[Byte]
4    type PointerToPointer = Ptr[Ptr[Byte]]
5
6    def WithZone[T](f: Zone => T): T = Zone(fn)
7
8    def allocPtrToPtr(implicit zone: Zone) = {
9      alloc[Ptr[Byte]]
10   }
11   ...
12 }
```

Figure 4. Our binding abstraction for Scala Native

On the JVM, ScalaPy uses Java Native Access (JNA) [20] to bind to the Python interpreter. While the Java Native Interface (JNI) [8] offers higher performance by having the developer write custom bindings in C, JNA lets us access native constructs such as pointers directly from our Scala code which offers us a significant advantage since this mirrors the interface we have on the Scala Native side. Thus, ScalaPy on the JVM is built on bindings written with JNA, which we see in Figure 5. The performance loss due to JNA is less significant since most data science operations do not cross the interoperability bridge frequently. In any case, we recommend using Scala Native for applications requiring high-performance interoperability.

In our JVM bindings, we use the same Pointer type for all native pointers (lines 3-4) since JNA does not use generics to type the elements referenced by the pointer. For heap allocation, we use the Memory type (lines 6-8), which does not require an implicit zone. To maintain compatibility with the zone-based Scala Native API, we define WithZone to call the wrapped function with a dummy Unit value.

```
1  import com.sun.jna._
2  object Platform {
3    type Pointer = Pointer
4    type PointerToPointer = Pointer
5
6    def WithZone[T](f: Unit => T): T = fn(())
7
8    def allocPtrToPtr = {
9      new Memory(Native.POINTER_SIZE)
10   }
11   ...
12 }
```

Figure 5. Our binding abstraction for the JVM

Now, instead of directly using JNA/Scala Native APIs in the interpreter interface, we only use the APIs and type aliases exposed by the common Platform interface which results in identical behavior on both platforms.

2.2 Scala-Python Heap Synchronization

Both Scala and Python have their own systems for memory management. Scala uses the JVM/Scala Native's implementation of garbage collection and Python uses reference counting with cycle collection. When Scala code holds references to Python values, we need to ensure that these values have the appropriate reference counts so that the Python runtime does not prematurely collect them. We manage this through an intermediate Scala object, of type PyValue, that represents a Scala reference to a Python value.

Whenever we load a Python value out of the interpreter, we wrap the native pointer we receive in a PyValue object. Treating each PyValue instance as a single reference to the underlying Python value gives us a lightweight way to ensure garbage collection correctness. With this system, the only time that there will be no references to a Python value from Scala will be when all PyValues for it have been garbage collected.

With the CPython APIs for managing reference counts, we increment the reference count for the underlying value when we construct a PyValue and decrement the count when the host garbage collector finalizes it. With this strategy, there will never be extra references to a deleted Python value since any live PyValue objects correspond to an incremented reference count of the underlying object. In the following figure, we see how the intermediate PyValue objects keep Python values alive while they are referenced by user code.

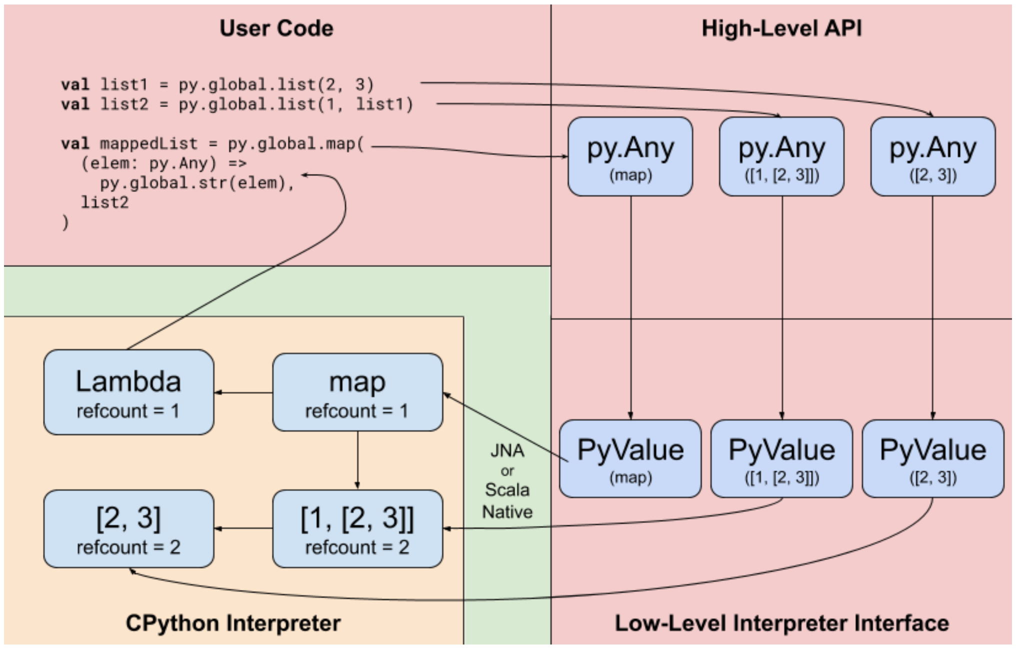

Figure 6. Diagram of references between Scala (red) and Python (yellow) in a simple ScalaPy program

As we see in Figure 6, there can also be references back from Python to Scala when lambdas are involved, which is much more complicated since Python does not have access to the host garbage collector. We will discuss how we handle heap synchronization in this reverse direction in Section 3.5.

2.3 Thread Safety and the Global Interpreter Lock

Unlike Scala, where threads are commonplace, Python does not support multithreading due to the Global Interpreter Lock (GIL) [4]. This lock must be held when performing any operations in the Python interpreter, which we must be especially aware of in the multithreaded world of Scala. Throughout our low-level Python interface, all calls to the native CPython API are wrapped in calls to `withGil`, a higher order function that lets us hold the GIL without duplicated code. As shown in Figure 7, this function acquires the GIL if necessary, calls the inner block, and releases the GIL.

```scala
def withGil[T](fn: => T): T = {
  val handle = PyGILState_Ensure()
  try { fn } finally {
    PyGILState_Release(handle)
  }
}

withGil { Py_IncRef(pyValue.underlying) }
```

Figure 7. Source of the `withGil` helper and a usage example

Unfortunately, the presence of the GIL means that Scala code interacting with Python from different threads will suffer from a performance penalty due to many threads trying to acquire a single lock. While subinterpreters [17] can avoid the GIL by creating instances of Python tied to specific threads, introducing these can cause inconsistent behavior when values are shared between threads which takes away from the referential transparency of using Python values like regular Scala objects. We recommend ScalaPy users to bring all Python interactions into a single thread with other threads used for processing data in Scala values. This can be achieved by using concurrency abstractions such as actors where many separate threads can communicate with each other while still processing events one-by-one locally.

3 High-Level Python Interface

We now introduce the high-level ScalaPy API, which exposes Python values as regular Scala objects with conversions between the equivalent data types in both languages.

3.1 Python API Entrypoints

Users have two main entrypoints to Python APIs: global scope and modules. ScalaPy exposes both of these as top-level APIs in the py package object: `py.global` and `py.module`.

Both global and module scopes are implemented using Scala's `Dynamic` type [3], which allows users to access attributes and functions in the scope with the same syntax as accessing members of Scala objects. For example, the Python code

```
myList = list()
print(myList) // []
```

can be equivalently written with ScalaPy as:

```
val myList = py.global.list()
println(myList) // []
```

Following the semantics of Scala's `Dynamic` type, the Scala compiler transforms any attribute access or method call under `py.global` or `py.module` to a call to `selectDynamic` or `applyDynamic`, respectively. These calls are invoked with the original element being accessed as a string (e.g. "list" in the above example), which allows us to load the function by its name and call it with `PyObject_Call`. To use modules, users can call `py.module` with the module name and use the returned object just like global scope. For example, we can call APIs from the NumPy library using the module API:

```
val np = py.module("numpy")
println(np.ones(3)) // array([1., 1., 1.])
```

3.2 Type Hierarchy and Correspondence

In ScalaPy, users interact with Python APIs through a hierarchy of types that correspond to core Python data types. This is modeled after Scala.js [1], where users interact with built-in JavaScript types through Scala facades. At the top of the ScalaPy hierarchy lies `py.Any`, shown in Figure 8, which is the base type for all Python values. Subtypes of `py.Any` include primitives such as `py.Number`, `py.Sequence`, and `py.Object`.

All the Python types in ScalaPy are traits so that users can extend them when working with other modules. Since these high-level wrappers must be able to access the underlying `PyValue` pointer, `py.Any` tracks this underlying value with the abstract member `underlying`.

```
trait Any {
  def underlying: PyValue
  override def toString: String =
    underlying.getStringified
  final def as[T: Reader]: T =
    implicitly[Reader[T]].read(underlying)
}
```

Figure 8. Definition of the `py.Any` type

With this set of core ScalaPy types, users can safely interact with Python APIs by converting Python values to instances of the corresponding Scala types through `Reader` instances, which we discuss next.

3.3 Python to Scala Conversions

ScalaPy supports conversion to Scala types with the `.as[T]` method through the `Reader[T]` typeclass. `Reader`s convert raw `PyValue` pointers into the target type `T`. For conversions into subtypes of `py.Any`, ScalaPy automatically synthesizes `Reader`s with an implicit macro [11]. For example, Figure 9 shows the `Reader` for `py.Sequence`, which constructs an instance with the given `PyValue`.

```
myPythonValue.as[py.Sequence](
  // automatically synthesized
  new Reader[py.Sequence] {
    def read(u: PyValue) = new py.Sequence {
      val underlying = u
    }
  }
)
```

Figure 9. A synthesized reader for the `py.Sequence` type

For Python primitives that directly correspond to Scala types such as `Int`/`Float`/`String`, we also support direct conversions that bypass the Python type hierarchy from the previous section. These conversions use native APIs exposed by the interpreter to convert the Python value into the appropriate C type. We then can directly return this C value since both JNA and Scala Native use Java's primitive types to represent C primitives. For example, the Python code

```
myList = [1, 2, 3]
print(len(myList))
```

can be written with ScalaPy as

```
val myList = py.global.list(1, 2, 3)
println(py.global.len(myList).as[Int])
```

Beyond these primitives, ScalaPy also includes one additional special type for developers to create accurate type definitions for their libraries: the union type. While union types will be supported in Scala 3 [10], they are not yet available in the Scala 2.x series. So, we must provide special support for this type in a manner similar to Scala.js. When a union type is taken as a parameter, users can supply arguments of either type, with implicit conversions available that wrap the value into the union type. For example, in Figure 10 we define a method taking either a tuple or sequence and call it with either input.

```
val myTuple: py.Tuple = ...
val mySequence: py.Sequence = ...
def myFunc(in: py.|[py.Tuple, py.Sequence]) =
  py.global.len(in)

myFunc(myTuple) // compiles!
myFunc(myList) // compiles!
myFunc(123) // does not compile
```

Figure 10. An example of using union types in ScalaPy

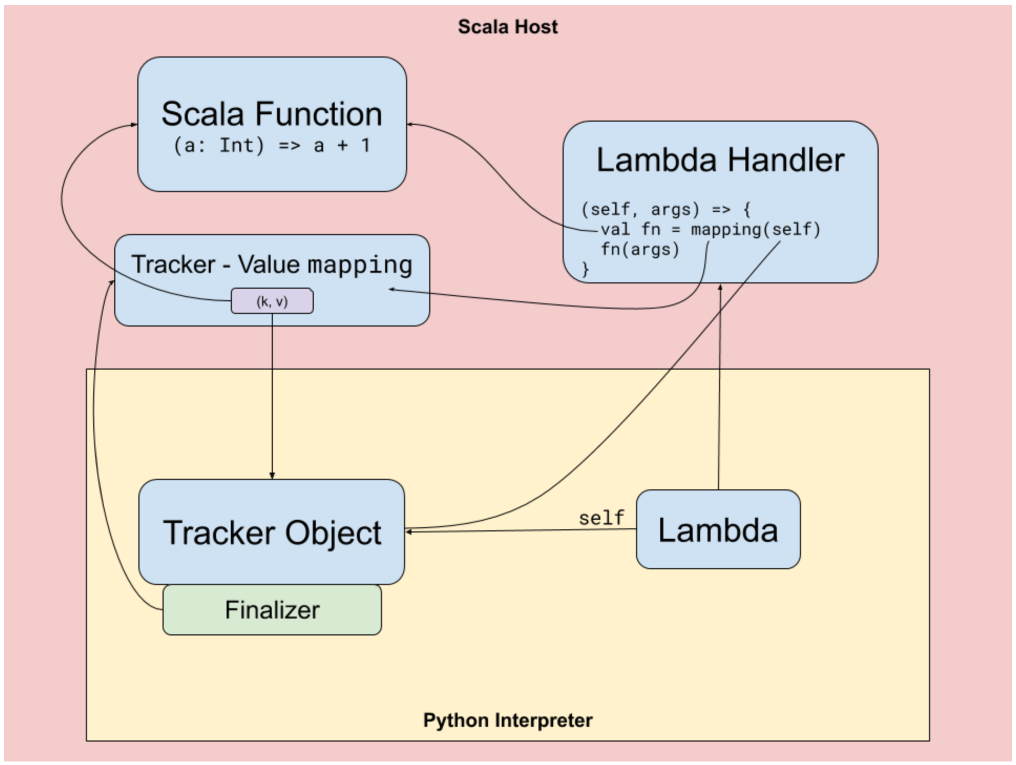

Figure 11. Diagram of the tracker object system for converting Scala functions to Python lambdas

3.4 Scala to Python Conversions

ScalaPy also supports conversions from Scala to Python values. With our embedding approach, Scala primitives must be converted, not just casted, into Python primitives. This is a critical difference between ScalaPy and other interoperability projects such as Scala.js and Scala Native which compile Scala code to the target language and therefore can encode Scala values using the target's primitives. We define the mapping between Scala and Python primitives in Figure 12.

Scala Type	**Corresponding CPython Type**
`Boolean`	`PyBool`
`Byte` / `Short` / `Int`	`PyInt`
`Long`	`PyLong`
`Float` / `Double`	`PyFloat`
`String`	`PyString`
`Unit`	`Py_None` (singleton)

Figure 12. Mapping from core Scala types to their CPython equivalents

In order to have Python API calls look just like regular Scala calls, these primitive conversions are defined as implicit conversions from the primitive type to subtypes of `py.Any`. In user code, these conversions will be automatically invoked when a Scala-to-Python conversion is necessary to satisfy type requirements. These automatic conversions can be performed safely from Scala types to Python types. Conversions in the reverse direction must done explicitly by the user as described in Section 3.3 because the types of Python values are only known at runtime so we cannot guarantee safety at compile-time.

With just these conversions, we can support simple Python API calls that look just like regular Scala calls. For example, we can construct a Python list of integers and have the individual Scala integers converted implicitly into Python numbers:

```scala
val myList = py.global.list(1, 2, 3)
println(myList)
// [1, 2, 3]
```

3.5 Function Conversions

In many Python libraries, developers can pass in callbacks to listen to logging events or provide custom transformations. So, with ScalaPy, we need to be able to support such libraries by allowing developers to pass in Scala functions where Python lambdas are expected.

For most of our primitive type conversions, there is a simple CPython API for converting the C equivalent to Python by copying the data involved. For functions, however, when passing function pointers into Python we lose all closure state associated with the function which makes a direct conversion impossible. What we need is a way to associate instances of generated Python lambdas with the original Scala function they are wrapping. We discuss how ScalaPy generates these associations in the following section.

3.5.1 Tracker Objects. Holding references to Scala values from Python is implemented with "tracker objects". A key challenge of holding such references is handling garbage collection, since the JVM/Scala Native host cannot see any references originating in the Python interpreter. Tracker objects let us keep both garbage collectors in sync by notifying the host of any garbage collection activity in Python that affects references to Scala values. Thus, we can safely allow Python to reference Scala values without introducing memory leaks or dangling references.

As shown in Figure 11, tracker objects and Scala values have a one-to-one mapping. This bijection (`mapping` in the figure) is implemented with two hash maps – one from the Scala object to the tracker object referencing it and one in the reverse direction. When the Python runtime collects the tracker object and executes its finalizer, ScalaPy can safely release the reference to the Scala value since there can only be one Python object associated with it at a time. When handling function calls, tracker objects enable us to use the same base implementation for different lambdas by storing a tracker for the Scala function in the `self` property of the Python lambda. Then, when handling a call in our base lambda handler, we unwrap the associated tracker into the original function and execute it.

To create tracker objects, we define a Python class using Python's dynamic types API [18]. Then, when a user executes a conversion that requires a tracker, ScalaPy creates an instance of the tracker class and updates the mapping. Immediately after creating the tracker, ScalaPy associates a finalizer [12] with it that will remove the tracker and its associated value from our maps when it is run. With this system, as long as the tracker object is being referenced in Python, the Scala value it tracks will stay alive in our maps, but as soon as Python cleans out the tracker, the Scala value is removed from our maps so can be freed if no Scala references exist.

3.5.2 Generating Python Lambdas. With tracker objects, converting Scala functions to Python lambdas becomes easier since we can reference Scala closures through trackers, but we still need to handle details such as argument packing. On initialization, we create a generic function pointer which will be invoked when handling calls to all converted lambdas. We use Python's support for associating lambdas with a `self` object to associate individual lambda instances with trackers for the underlying Scala functions. When handling a lambda call, we unwrap the tracker into the referenced Scala function, invoke this function normally, and return the resulting `PyValue`. This process is summarized in the following snippet.

```scala
(self: PyValue, args: PyValue) => {
  val fn = unwrapTracker(self)
    .asInstanceOf[PyValue => PyValue]
  fn(args).underlying
}
```

Figure 13. Handler for proxied calls to Scala functions

When wrapping a specific function, we do not directly generate a tracker for the function to be converted. Instead, ScalaPy generates an intermediate function to be tracked which takes only a single `PyValue`. This is necessary since when lambdas are called with multiple arguments, the arguments are packed together into a tuple that we must manually unpack before calling the Scala function. For example, we generate the following intermediate function when converting a function taking two parameters.

```scala
def fn2(f: (PyValue, PyValue) => PyValue) = {
  val handler = (args: PyValue) => f(
    PyTuple_GetItem(args, 0),
    PyTuple_GetItem(args, 1)
  )
  ...
}
```

Figure 14. Intermediate function to convert a `Function2`

Using the mechanisms described above, ScalaPy is able to support conversions of any Scala function into a Python lambda. This makes interactions with data processing libraries even more natural since Scala developers can add callbacks to listen to events just like they would with Scala libraries.

4 Dynamic and Static Typed Python Interfaces

Once a user has loaded a Python value into Scala code, they have two main options for interacting with the object: dynamically through the `py.Dynamic` type or statically by creating Scala types that describe the attributes and methods available on the value. While the dynamic interface makes it easy to get started with a new library without any extra effort, the option to introduce static types enables Scala code to interact with Python values with confidence that there will not be runtime errors due to missing methods or mismatched argument types.

4.1 Dynamic Interface with `py.Dynamic`

Dynamic calls in ScalaPy are handled with the `py.Dynamic` type, a subtype of `py.Any` which indicates that the user has explicitly opted into the dynamic interface. The `py.Dynamic` type supports attribute loads and stores, method calls, and even some concepts with no syntax equivalents in Scala such as indexing into arrays and dictionaries.

Dynamic calls in ScalaPy directly return `py.Dynamic` instead of just `py.Any`, which enables chained dynamic calls. Similarly, both global and module calls through `py.global` and `py.module` return `py.Dynamic` values. On the user side, this results in a natural interface when getting started with ScalaPy since they can use familiar APIs without any extra steps to involve static type definitions. For example, a user can create a list, append elements, and get the sum of them all with code that looks like it could have been for a regular Scala library.

```scala
val myList = py.global.list(1, 2, 3)
// myList: py.Dynamic

myList.append(4)
myList.append(5)
println(py.global.sum(myList)) // 15
```

Figure 15. An example of dynamic calls on Python objects in ScalaPy

Some syntax features in Python do not have equivalents in Scala, such as the use of square brackets for array and dictionary access. To perform this indexing operation on a Python value in ScalaPy, users can call the `.arrayAccess` method. For example, the Python code

```python
print(myList[0]) // 1
```

can be converted to Scala as

```scala
println(myList.arrayAccess(0)) // 1
```

In some situations, users may have difficulty translating Python code into Scala. ScalaPy allows developers to write snippets of Python code in Scala that mix in Scala values using the `py""` string interpolator. For example, we can perform the sum operation in Figure 15 with a Python snippet:

```scala
println(py"sum($myList)") // 15
// expanded to
println(StringContext("sum(", ")").py(myList))
```

Under the hood, this generates a new scope containing all the values being interpolated under generated variable names and then executes the snippet with the interpolated sections replaced with references to the generated variables. Since each generated variable is only used once within the temporary scope, it can only be used to load values since setting the variable to a new value will have no effect on the program execution. With this option, developers can quickly integrate Python code examples they find and translate components into strongly typed code over time with the APIs we describe in the next section.

4.2 Static Interfaces with `@py.native`

While dynamic interfaces are excellent for experimentation with new APIs, using them in production code can be dangerous since doing so loses the benefit of Scala's strong type system. In addition, dynamic interfaces do not play well with IDE features such as code completion since there is little knowledge these tools can gather from `py.Dynamic`. To enable developers to use Python while maintaining strong type safety, ScalaPy includes a type definition system that can wrap Python values in statically typed interfaces.

Python value types are defined as regular traits with a few annotations that let ScalaPy generate the appropriate runtime support code. All type definition traits must be annotated with `@py.native`, which enables macro transformations that replace the bodies of the trait's members with the appropriate API forwarders. Every method definition in the trait must have a `py.native` body, which acts as a placeholder that will be replaced by a forwarder. For example, in Figure 16, we create a type definition for the `random.Random` type.

```scala
@py.native trait Random extends py.Object {
  def uniform(a: Double, b: Double): Double =
    py.native
}

val r = py.module("random").as[Random]

println(r.uniform(0, 1)) // 0.25347253974
```

Figure 16. Creating a static type definition for `Random`

The overall definition style is designed to mirror Scala.js, which is already proven to be a success in the real-world. Scala.js specially handles type definitions when emitting JavaScript, so it does not require any macros to be involved. ScalaPy, however, depends on the Scala compiler's existing backends for emitting Java byte code and Scala Native IR. This means that we can only perform transformations until the typechecking phase, which leaves us with macros as the only choice.

The `py.native` macro is expanded in trait members with a dynamic call to a method with the same name as the member. The result of the call is converted to the expected result type with the `.as` method.

```scala
def uniform(a: Double, b: Double): Double =
  as[py.Dynamic].uniform(a, b).as[Double]
```

Figure 17. Expanded implementation of `Random.uniform` after typechecking

Then, when calling the `.as` method to wrap an existing Python value in a static facade, the generated reader constructs the trait with the abstract `underlying` value filled in with the value being wrapped.

```scala
val r = py.module("random").as[Random](
  new Reader[Random] {
    def read(v: PyValue) = new Random {
      val underlying = v
    }
  }
)
println(r.uniform(0, 1)) // 0.25347253974
```

Figure 18. Generated instance of the `Reader` typeclass when casting into the type definition

With these transformations, developers can write Scala code that interacts with Python APIs while being confident that type errors will be caught at compile-time.

5 Scala-Python Data Sharing

While the Python C API works well for conversions between primitive types, converting larger data structures such as sequences requires a significant amount of logic since these have completely different representations in Scala and Python. In addition, for such data structures, conversions affect not only the top level value, but also nested values which must all be converted to their equivalents in the target language. In this section, we focus on sequence conversions, but ScalaPy supports additional built-in data structures such as dictionaries and tuples as well.

5.1 Loading Python Sequences in Scala

When loading sequences from Python, we emit a Scala sequence wrapper that forwards all operations to the underlying Python value. This conversion is introduced as a `Reader[Seq[T]]` which can be synthesized as long as a `Reader[T]` is available for converting individual elements. Emitting only a proxy instead of a full copy of the data matches the intended semantics of the `.as` method, where we want our converted value to always match the underlying Python value even if other code modifies it. For example, we can observe changes to an underlying collection

```scala
val myPythonList = py.global.list(1, 2, 3)
val asScalaList = myPythonList.as[Seq[Int]]

println(asScalaList) // Seq(1, 2, 3)

myPythonList.append(4)
println(asScalaList) // Seq(1, 2, 3, 4)
```

Figure 19. Using proxies that reflect underlying changes

In some data-heavy applications, developers may write code that frequently reads data from loaded collections but does not need to observe underlying changes after the conversion. Proxy collections can introduce a significant performance penalty since conversions of individual elements will be executed on every access. This penalty is significantly reduced in Scala Native, where calls into native functions have effectively zero overhead, but can have a major slowdown on data-heavy code running on the JVM. In this case, users can easily convert their proxy into a realized Scala collection through one of the built-in collection operations such as `toVector`.

5.2 Sending Scala Sequences to Python

In the reverse direction, when sending Scala collections to Python, ScalaPy offers both proxy and copy conversion options as separate APIs. Unlike when pulling data from Python, pushing data as a copy is significantly faster since having Python grab a copy of each element of a proxy involves function pointer calls into the host language which can incur a serious performance penalty. For this reason, developers must decide in their Scala code whether to send data to Python as a proxy or a copy.

With our conversion from Scala functions to Python lambdas described in Section 3.5, generating Python proxies over Scala collections becomes relatively easy since we can implement this conversion in terms of the high-level APIs. Users can request conversion with `.toPythonProxy`, which takes in a `T => py.Any` implicit parameter for converting elements. We generate lambdas to get elements at a given index and for the length of the sequence. Then, we construct a sequence proxy that will invoke these lambdas when Python code reads elements. We can see the semantics of proxies in action in lines 10-14 of Figure 20.

For copying, we have `.toPythonCopy` which takes the same implicit parameter but now uses low-level APIs from the interpreter interface to insert elements one-by-one into a newly created Python list. As a result of using the low-level interface, we have to do a bit of extra work to pass off ownership of the new elements to the list. In lines 16-20 of Figure 20, we see that all elements of the sequence are extracted during the `toPythonCopy` call instead of lazily at the final `println`.

```scala
val mySeq = new Seq[Int] {
  def apply(index: Int) = {
    println(s"requested $index")
    index + 1
  }
  def length = 2
  def iterator = ...
}

val pythonProxy = mySeq.toPythonProxy
println(pythonProxy)
// requested index 0
// requested index 1
// [1, 2]

val pythonCopy = mySeq.toPythonCopy
// requested index 0
// requested index 1
println(pythonCopy)
// [1, 2]
```

Figure 20. Passing a sequence into Python through proxies and copies

With both these options, developers can make the appropriate decisions throughout their code base to send data as proxies that have no memory overhead but are slower to get individual elements or as copies which increase memory use but are fast in data-heavy applications. A major advantage of Scala Native as the target platform here is that using a proxy is a competitive option even in data-heavy code since compilation to native code eliminates the overhead of crossing between Scala and native functions so the overhead of proxies is much lower.

6 Evaluation

Our primary claim in this paper is that with ScalaPy, developers can use Python libraries with ease in Scala applications for various data science workflows. We demonstrate this capability through benchmarks that test interoperability performance and multiple projects developed both by the authors and members of the open-source community.

6.1 Interoperability Benchmarks

ScalaPy enables developers to use Python libraries while maintaining high performance. We implement two benchmarks to evaluate the performance aspects most relevant to data science workflows. The first measures the performance of transferring collections between Scala and Python, which occurs frequently when using Python libraries to analyze Scala data. The second compares the performance of a complete application written with ScalaPy against a pure Python equivalent.

6.1.1 Data Transfer Benchmarks.

We perform two data transfer benchmarks: one for sending a collection to Python and one for reading values back. We use both proxy and copy strategies for collection conversion to demonstrate the tradeoff between the two. We also benchmark on both the JVM and Scala Native to demonstrate the differences in native binding performance. We ran our benchmarks on OpenJDK 13 with a 2.9 GHz Intel i7-7820HQ and 16 Gb RAM.

In our first benchmark, we generate a Scala vector of doubles and measure the time to convert it into a Python sequence. We see the results of this benchmark in Figure 21.

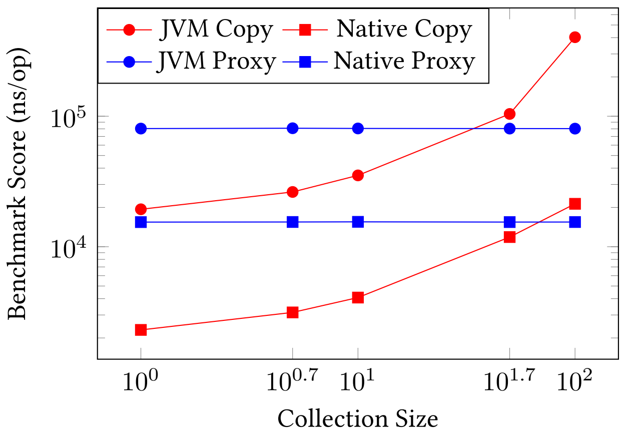

Figure 21. Scala to Python conversion benchmarks (lower is better)

In our second benchmark, we focus on read performance. We initialize a Python sequence by the same process as the first benchmark. Then, we convert the sequence back to Scala and measure the time to sum its elements. This captures both the overhead of loading values from Python as well as of proxy reads. As a baseline, we sum the original sequence without any conversions. We see the results in Figure 22.

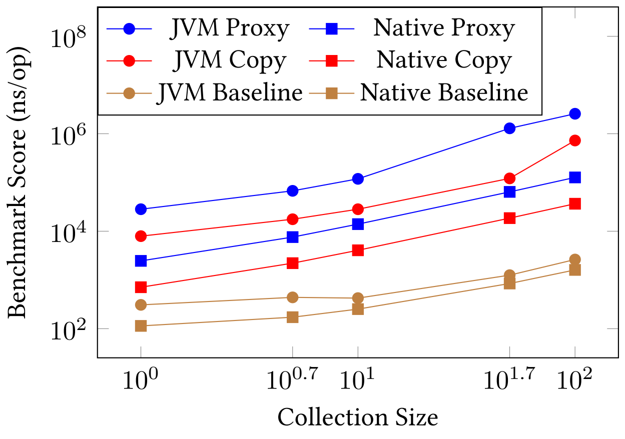

Figure 22. Reading benchmark results (lower is better)

Across both of these benchmarks, we see a clear performance advantage of using Scala Native to bind to the Python APIs, with an almost 10x performance gain. They also demonstrate the tradeoff between proxies and copies, since proxies have a constant conversion overhead time compared to a per-element overhead for copying the data. On the other hand, proxies have a larger overhead compared to copies when reading back values. For applications where converted collections will be accessed at only a few indices, proxies have an advantage since even large collections only have constant overhead for conversion and individual reads. For data science applications where all the collection data will be used, copies offer performance gains since Python code can read values with minimal overhead.

6.1.2 End-to-End Application Benchmark. In addition to low-level collection benchmarks, we compare the performance of ScalaPy to hand-written Python for an end-to-end data science application. In our benchmark, we define a TensorFlow graph for least-squares linear regression and perform 50 iterations of gradient-descent. In this benchmark, ScalaPy on the JVM averaged 0.2868 sec/op, ScalaPy on Scala Native averaged 0.2873 sec/op, and hand-written Python averaged 0.2868 sec/op. These numbers are almost identical with benchmark variability likely making up the difference. This demonstrates how in data science applications, where the majority of the time is spent in specialized native libraries for data operations, the interoperability overhead of ScalaPy has almost no consequence on the final application performance.

6.2 Production Applications

In addition to performance benchmarks, ScalaPy has been proven through various projects that cover a wide range of interoperability needs.

6.2.1 Machine Learning Research. ScalaPy was originally created to enable Scala developers to access TensorFlow from their programs. To that end, we have developed official bindings to TensorFlow [1] and NumPy [2] for ScalaPy that have been used in various complex projects. One project involved deep reinforcement learning to control a simulated robot in an unknown terrain. ScalaPy made it possible to simulate the environment with high performance in Scala while sending reward updates to TensorFlow for learning. In another project, we used ScalaPy to train a convolutional neural network to mimic human responses to complex environments in real-time. Here, Scala was used to capture streaming images and pre-process them with OpenCV and ScalaPy was used to relay human actions to TensorFlow for training. With ScalaPy's concise static type definition API, all the types required for these projects to safely interact with NumPy and TensorFlow took fewer than 400 lines of code.

6.2.2 Jupyter Notebook Compatibility. One of the reasons why ScalaPy supports the JVM is to handle notebook environments, where snippets of code must be compiled and executed at runtime through the Scala interpreter, which only works on the JVM. We have developed multiple notebook examples [3] that port existing tutorial notebooks developed for TensorFlow to ScalaPy. With the ScalaPy dynamically-typed interface, porting these notebooks was as simple as copying the original Python source and replacing incompatible syntax the Scala equivalent and introducing the occasional `toPythonProxy` call for a sequence conversion. These notebooks look almost identical to their Python equivalents with no major changes in the layout of the code since ScalaPy offers equivalent Scala APIs for all Python features.

6.2.3 Type-Safe Tensor Operations. One project [4] building on top of ScalaPy developed by a Scala community member offers a stress-test of our static typing system. This project offers a type-safe interface to TensorFlow that goes beyond just checking object types to also check the shape of tensors involved. Built on top of Dotty, this system tracks the dimensions of tensors through literal types and uses new features of the typechecker to check tensor operations at compile time. We see this in action in the following snippet

```scala
val tensor = tf.zeros(10 #: 20 #: 30 #: SNil)
// ...: Tensor[Float, 10 #: 20 #: 30 #: SNil]

tf.reduce_mean(tensor, axis = 0 :: SNil)
// ...: Tensor[Float, 20 #: 30 #: SNil]

tf.reduce_mean(tensor, axis = 1 :: 2 :: SNil)
// ...: Tensor[Float, 10 #: SNil]
```

With this setup, even dimension errors can be caught at compile time, which shows how ScalaPy enables new ways to tackle active research areas like tensor operation checking.

All these projects together exercise many different features of ScalaPy, such as function conversions to pass callbacks into machine learning models, type definitions to interact with these libraries while maintaining safety, and both proxies and copies for sharing large datasets. The projects demonstrate the ability of ScalaPy to handle these varied workflows with a natural system for users to interact with Python APIs.

7 Related Work

Interoperability with Python is a popular area as many languages look to adopt the large ecosystem the language has developed around scientific computing.

PythonKit [19], originally developed as part of the Swift for TensorFlow project [5], enables developers to use Python

[1]https://github.com/shadaj/scalapy-tensorflow
[2]https://github.com/shadaj/scalapy-numpy
[3]https://gist.github.com/shadaj/29d77180aeefc41a749273026f7d1fd9
[4]https://github.com/MaximeKjaer/tf-dotty

APIs from Swift. This is most similar to ScalaPy in its goals since Swift shares many features with Scala with a similar focus on offering safety and performance while being concise. However, this project only offers dynamically typed APIs which limits its use case to prototype projects and research where strong typing is not as important.

Jep [7] is a library for interoperability between Java and Python that embeds Python through JNI. Since Scala can run on the JVM, Scala programs can use Jep to access Python libraries. However, Jep fails to preserve semantics when using Python values because it uses built-in Java structures to wrap Python values rather than a separate type hierarchy. When a suitable Java equivalent for a Python value does not exist, Jep often falls back to unusable representations such as a stringified version of the object. In addition, Jep cannot be used in Scala Native since it is written in Java, which rules out its use on resource-constrained platforms.

Polynote [9] is a notebook environment that focuses on supporting multiple languages simultaneously with data conversions between the languages it supports, which include Python and Scala. As a notebook environment, Polynote cannot be used for production systems and performance is not its primary concern. To that end, Polynote's support for conversions between Scala and Python is internally implemented using Jep, so it inherits all the limitations we previously discussed. In future work, it would be interesting to explore replacing Jep with ScalaPy.

8 Conclusion

Python and Scala represent very different programming paradigms. Python is a dynamically-typed language that sacrifices performance in exchange for conciseness and ease of use. Scala, on the other hand, is statically-typed and is used in many large-scale, high-performance applications. With ScalaPy, we have brought these two worlds together with a seamless interoperability layer. With a cross-platform interpreter embedding, we give developers the flexibility to integrate Python into existing JVM applications or compile directly to native code for maximum performance. With heap-synchronization and thread safety, developers can treat Python values as having the same semantics as regular Scala objects. Through automatic conversions between Scala and Python types, developers can mix Scala and Python values naturally. Just like Python, developers can quickly get started with new APIs through our dynamically-typed interfaces, but can gradually adopt static types to ensure safety and maintainability.

Acknowledgments

We would like to thank Maxime Kjaer, Bruno Fernandes, Andrew Valencik, and Mark Melton, who made various open-source contributions to ScalaPy that shaped its development.

References

[1] Sébastien Doeraene. 2013. Scala.js: Type-Directed Interoperability with Dynamically Typed Languages. (2013), 10. http://infoscience.epfl.ch/record/190834

[2] Sébastien Doeraene, Tobias Schlatter, and Nicolas Stucki. 2016. Semantics-Driven Interoperability between Scala.js and JavaScript. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. Association for Computing Machinery, New York, NY, USA, 85–94. https://doi.org/10.1145/2998392.2998404

[3] EPFL and Lightbend. 2019. *Scala Standard Library 2.13.1 - scala.Dynamic*. https://www.scala-lang.org/api/2.13.1/scala/Dynamic.html

[4] Python Software Foundation. 2020. *Thread State and the Global Interpreter Lock*. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock

[5] Google. 2020. *Swift for TensorFlow*. https://www.tensorflow.org/swift

[6] Google. 2020. *TensorFlow 2 Quickstart for Beginners*. https://www.tensorflow.org/tutorials/quickstart/beginner

[7] Mike Johnson et al. 2020. *Jep - Java Embedded Python*. https://github.com/ninia/jep

[8] Sheng Liang. 1999. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional.

[9] Netflix. 2019. *polynote/polynote - A better notebook for Scala (and more)*. https://github.com/polynote/polynote

[10] Martin Odersky et al. 2020. *Dotty Compiler: A Next Generation Compiler for Scala*. EPFL. https://dotty.epfl.ch

[11] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. 2010. Type Classes as Objects and Implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 341–360. https://doi.org/10.1145/1869459.1869489

[12] Antoine Pitrou. 2013. *PEP 442 – Safe object finalization*. Python Software Foundation. https://www.python.org/dev/peps/pep-0442/

[13] Emmanouil Platanios. 2020. *TensorFlow Scala*. https://github.com/eaplatanios/tensorflow_scala

[14] Python Software Foundation. 2020. *Python/C API Reference Manual*. https://docs.python.org/3/c-api/index.html

[15] Norman Ramsey. 2003. Embedding an Interpreted Language Using Higher-Order Functions and Types. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME '03)*. Association for Computing Machinery, New York, NY, USA, 6–14. https://doi.org/10.1145/858570.858571

[16] Denys Shabalin and Martin Odersky. 2018. Interflow: Interprocedural Flow-Sensitive Type Inference and Method Duplication. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala (Scala 2018)*. Association for Computing Machinery, New York, NY, USA, 61–71. https://doi.org/10.1145/3241653.3241660

[17] Eric Snow. 2018. *PEP 554 – Multiple Interpreters in the Stdlib*. Python Software Foundation. https://www.python.org/dev/peps/pep-0554/

[18] Talin. 2007. *PEP 3115 – Metaclasses in Python 3000*. Python Software Foundation. https://www.python.org/dev/peps/pep-3115/

[19] Pedro José Pereira Vieito. 2020. *pvieito/PythonKit - Swift framework to interact with Python*. https://github.com/pvieito/PythonKit

[20] Timothy Wall et al. 2020. *java-native-access/jna: Java Native Access*. https://github.com/java-native-access/jna