To Whom It May Concern,

Firstly, sincerest thanks to the reviewers for all of their helpful suggestions– they've helped us to substantially improve our framework and its presentation both from an academic, as well as an industrial perspective (improvements thanks to reviewer feedback has been incorporated into the production version of our framework, due to be released shortly.)

This cover letter is broken into the following sections:

- **Editorial Feedback, Revisions** - Mapping/response to requested revisions that the Committee agreed must be addressed in the revised submission.
- **Review 1** - Mapping/response to reviewer 1's suggestions.
- **Review 2** - Mapping/response to reviewer 2's suggestions.
- **Review 3** - Mapping/response to reviewer 3's suggestions.

Author responses are in blue/bold.

Editorial Feedback, Revisions

The revisions that have to be made in order for the paper to be accepted:

1. need to show the serialization of graphs.

   The revised paper includes new experimental results (see Section 6.4) using a real-world benchmark with cyclic object graphs (pickling/unpickling of a subgraph of the real Wikipedia dataset is shown). In addition to the experimental results, the revised paper includes a discussion of the implementation of object graph support in scala/pickling. Section 4.6, Section 6.4 (pages 13-14, pages 15-16)

2. tone down the claims.

   The revised paper contains additional results where we compare scala/pickling with Java on a benchmark with cyclic object graphs (see Section 6.4). These results lead us to quantify our claims about performance more carefully and more precisely, in the introduction and elsewhere. Section 6.4 (pages 15-16)

3. provide more details for the sharing issue.

The new Section 5.6 includes a discussion of sharing, and shows how object identity tracking enables sharing in some cases. Section 5.6 (pages 13-14)

Review 1

Evaluation:

“ - I was a bit lukewarm about the (semi) formal presentation in Section's 4 and 5. I think I'd prefer to see some actual Scala code (for example for the IR combinators), or go fully formal. But I guess this presentation may have it's merits.”

The presentation in Section 4 has been reworked and has been split into two parts. The first part introduces OO pickler combinators informally using actual Scala code. The second part presents a new formalization using an operational semantics for a core object-oriented language. In contrast, to improve the presentation in Section 5 we have added actual Scala code from our real implementation for some of the IR types and combinators. Section 4, 5

Detailed Comments:

“- Section 3.2: When explaining the pickeable annotation, it could be helpful to illustrate the concept using some code. Perhaps show a class annotated with @picklable (say Firefighter) and then show an excerpt of the kind of code that gets generated.

“- Experimental evaluation: I wonder if something more could be said about kryo v2. It seems that the larger the number of elements is the better is the relative performance of kryo 2. In particular it beats Scala pickling for large number of elements. Is the reason for better perfomance the fact that a custom pickler was used for kryo?

> **Kryo uses an important optimization that has been viewed as somewhat biased in the open-source community– it allocates one enormous Java byte array for all serialized instances across the entire run of the entire application (resizing the array if necessary), and even across runs. This optimization often makes benchmarking against other frameworks unfair in that array allocations end up being measured rather than actual effort serializing. All other JVM approaches to serialization to the best of our knowledge allocate individual arrays per “pickle”. We believe this optimization is a key reason why Kryo appears to scale better. Furthermore, Kryo uses a slew of other runtime value-based optimizations, eg, representing bytes with fewer bits so as to avoid having to deserialize many zeroes needlessly. While these optimizations are certainly important, meticulously optimizing our generated code wasn’t within the (initial) scope of our work. Rather, our goal has been to show that a mixed compile-time/runtime approach is typically more performant, even without extra optimization. See sections 4-5 (pgs)**

“- Figure 2: Maybe a small legend in the figure would be useful. I guess the full circle means "heavy use", the empty circle means "not used", and the half-full circle mean "sparsely used".

> **Done. We also added a description of the legend at the beginning of section 7.3. See sections 4-5 (pgs) [DO!]**

Small Typos/Comments:

“-page 5, 2nd column: "return type Unit" should this be: "return type U"? The unpickle combinator returns a value of type U, I believe.”

> **Yes, you’re absolutely correct. This was a typo. However, this formulation has now been replaced with the aforementioned operational semantics in section 4. See sections 4-5 (pgs)**

“- page 8, 2nd column: "The function concat takes two arguments that are each sequences" sounds a little strange, maybe: “The function concat takes two sequences as arguments"”

> **Done. See sections 4-5 (pgs)**

“- page 11, 2nd column: "Scala's macros** [28]"”

> **Done.**

Revisions:

“Perhaps add some more Scala code corresponding to some of the semi-formal definitions in Section 4 and 5.”

> **Done. We have revamped sections 4 & 5 to begin first with a Scala example-based approach, followed by the formal opsem presentation See sections 4-5 (pgs) [DO!]**

Review 2

Technical Review:

“The paper included a rudimentary evaluation section but the favourable comparison with Java serialization seems inappropriate since the pickling framework makes no attempt to preserve graph structure (sharing) and avoid loops.”
&
“In a purely a functional setting, preserving sharing (let alone discovering additional sharing by hash-consing) is just an important optimization, so it's perhaps reasonable to ignore it at first. But in an imperative, object oriented language, like Scala, object identity is observable. The paper doesn't mention how this is dealt with at all and making performance comparisons with Java's serialization, which, as far as I'm aware, does preserve identity and deals with cycles, seem out of place.”

> **We’ve fixed this, and show a benchmark on an object graph created from a non-trivial partition of Wikipedia. See sections 7.. Importantly, due to our mixed compile-time/runtime approach, we’re able to statically detect that**

certain types (e.g. classes composed of primitives) cannot loop. Thus, in these cases, we're able to avoid the runtime performance hit caused by object identity book-keeping at runtime (that is, in these cases, we do not generate/ inline the code for object identity.) We've added a new section on object identity & sharing. See sections 4-5 (pgs)

Detailed Comments:

"page 3. You name subtype polymorphism and open class hierarchies as core components of object oriented programming but leave out object identity."

We've fixed this, object identity is now listed as core component of OO programming throughout the introduction. See sections 4-5 (pgs)

"You don't describe how you deal with Java's covariant arrays."

We have added a paragraph to Section 5.5 ("Generics and Arrays") which explains how Java's covariant arrays can be handled. We also point out that at the time of writing only support for primitive arrays has been implemented in scala/pickling. See section 5.5 (pgs)

"You use implicitly[T] in a number of places - without describing what it does. I assume it's just: def implicitly[T](implicit e: T): T = e?"

We now introduce the definition of implicitly[T] at the beginning of Section 4. See sections 4-5 (pgs)

"I don't fully understand why you don't support pickling of Scala functions - can you explain the issue? I imagine function values are represented as classes with a single method."

It's true that functions are represented as classes with a single method. However, even so, they can capture a complicated environment which might be difficult or impossible to pickle without additional compile-time support (e.g. a forced copy-on-capture semantics,). See sections 4-5 (pgs)

Revisions:

"Please address more prominently the issue with preserving object identity. Either describe how you do it, or if you don't (as I surmised from the paper) please clearly state why you have made this choice and why your comparison with Java serialization is fair."

We now preserve object identity and a form of sharing. The comparison with Java is fair because our mixed compile-time/runtime approach makes it possible to statically determine when it's not necessary to track object identity, making it possible only to pay for the object identity performance hit when there's a possibility of a given type containing cycles. We demonstrate the See sections 4-5 (pgs) [DO!]

Review 3

“Small comments for the technical contents”:

“1. Do you support modular unpickling? For instance, imagine we have a network packet whose first bits indicate the destination address. Instead of unpickling the whole packet, a receiver may want to find out whether the packet is directed to her, and only need to unpickle the packet partially. You support partial pickles, so you may very well already have this feature. (I'm not sure because you only explained PickleBuilder.)”

> **... See sections 4-5 (pgs) [DO!]**

“2. As a future work, it might be worth it to consider how much the DPickler vs. SPickler distinction can be inferred. I imagine in the most general case of open programming, full inference is not possible, but some may be viable (the trivial case is the final type picklers I guess).”

> **... See sections 4-5 (pgs) [DO!]**

“3. You described type variance in Sec 4.3. I don't think you made it clear which mechanism was eventually used. The F-bounded invariance? Or do you allow it all, i.e. whatever is a legal generic type use is also a legal pickler type use?”

> **The mechanism that we ended up using is now clearly stated (invariant picklers with type bounds) at the end of Sec. 4.3. See sections 4-5 (pgs) [DO!]**

Presentation:

“I. I think Sec 2 can be removed, with its content dispersed into respective sections. For instance, Sec 3 only needs some minimal knowledge of Sec 2 (implicit methods and annotations). As it stands now, Sec 2 is detailed (such as explaining @withNewToString) but Sec 3 is not (e.g. there is no example for @pickleable). Merging the two can reduce distractions and give you space to explain the key features better.”

> **... See sections 4-5 (pgs) [DO!]**

“II. I wonder if the definitions you introduced in Sec 4 can be formalized as opsem rules and definitions: (1) For Def 4.1, instead of defining picklers and unpicklers as Scala traits, just define them as values, e.g. <id; m; t> where id is the ID, m is a tag either as static or dynamic, and t is the type tag. With that, your dynTypeOf is just a tuple selector. (3) Def 4.3 can be defined as two opsem rules for "pickle" and "unpickle" expressions. What you have there is almost there. (4) There are some complexities when it gets to formalize partial pickles and the behaviors of static picklers.”

> **... See sections 4-5 (pgs) [DO!]**

“III. I wonder if some of the discussion in 5.3 can be formalized, especially 5.3.2. It now looks like a very algorithmic definition, with words like "emit" "generate." Some sort of rewriting rules may help here. Or if you really think the algorithmic description is a good idea, try to use the more rigorous pseudo-code form more commonly seen in program analysis papers.”

... See sections 4-5 (pgs) [DO!]

(Selected) Small Issues:
All other indicated small issues are typo fixes, that we have addressed.

“page 2, left column: "big data." This is a great theme. I wish you could bring this out more in the intro.”

... See sections 4-5 (pgs) [DO!]

“page 2: I think compositionality is a crucial feature of your framework, and it should be put into the principle list.”

... See sections 4-5 (pgs) [DO!]

“page 4 right column: "Note that in the above example, the unpickle method is parameterized on obj's precise type..." This should be mentioned right after the example. I have been wondering what "[Obj]" is for a while. Some of this problem can be fixed by introducing a simple abstract syntax for a core language.”

Done. See sections 4-5 (pgs) [DO!]

“page 9 Sec 5.3: This is in fact a very interesting algorithm. Unfortunately, it is a bit too late in the presentation. Consider moving it up a bit if possible.”

... See sections 4-5 (pgs) [DO!]