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.)

Highlights of the revision:

- Significantly extended experimental evaluation including serialization of objects graphs
- Completely revised formalization of the most important properties based on an operational semantics
- Extended discussion of object identity tracking and sharing
- More Scala code, both in terms of examples and in terms of excerpts from the actual implementation

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 and extensive 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. See Section 4.6, Section 6.4 (pages 13-14, pages 16-17)

2. tone down the claims.

   We have removed or toned down several bold statements about performance dispersed through the paper. The revised paper also 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. See Section 6.4 (pages 16-17)

3. provide more details for the sharing issue.

The new Section 4.6 includes a discussion of sharing, and shows how object identity tracking enables sharing in some cases. Section 4.6 (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.

We agree. We've extended the explanation of the pickleable annotation in Section 2.2 (used to be Section 3.2) to show an annotated class as well as an excerpt of the generated code.

"- 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.

"- 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".

We've added an explanation of the different circles to the beginning of Section 6.6 in the revised paper. Section 6.6

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 which is fixed in the new Section 3.1.

"- 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"""

Yes, this is certainly much clearer. It has been corrected in the new Section 4.2.

"- 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."

We have added more Scala code both to the old Section 4 (now Section 5) and to the IR definitions in the old Section 5 (now Section 4). (pages 9-14)

Review 2

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. Section 1 (pages 1-3)

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

We have added a paragraph to Section 4.5 ("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. Section 4.5 (pages 12-13)

“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?”

Yes. We introduce implicitly now in Section 2.2. (page 4)

“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.”

Yes, in Scala function values are represented as classes with a single method. However, closures could capture not only user-visible values in the environment, but also references to enclosing objects which are implementation artifacts. Our plan is to first have a principled solution for this issue before integrating functions into our framework.

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.”

The issue of preserving object sharing is now discussed both in the text (Section 4.6) where we explain how we support it and it also features in a new benchmark. The new benchmark contains cyclic object graphs and we provide a comparison with Java. Since object identity is preserved the comparison with Java serialization is fair. Even though Java is faster for that benchmark, our framework can still leverage the absence of cyclic graphs for some of the other benchmarks. Sections 4.6 and 6.4. (pages 13-14 & pages 16-17)

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.)”

We believe our framework could support this use case. However we have not done a concrete experiment, yet, and so we left it out. We are very interested

in trying this use case and would like to address this in a revision of the paper if possible.

“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. 3.1.4. (pages 6-7)

“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.”

As suggested we have dispersed the content of Sec. 2 into the respective sections that follow. We removed what used to be Sec. 2. The old Sec. 3 (which is now Sec. 2) contains in addition an example for @pickleable. Section 2 (pages 3-5)

“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.”

We have reworked a large part of the old Section 4 to use an operational semantics as suggested. The new formal development is contained in Section 3.2 (pages 7-9).

“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.”

We have tried to use a more rigorous form for the discussion in 5.3, however neither rewriting rules nor the suggested pseudo-code turned out to be a good fit. The main reason is that the presented generation "algorithm" has already lost its recursive nature due to the fact that we can simply emit "implicitly[SPickler[FieldType]]" and Scala's implicit inference takes care of recursively expanding our macro. However, a formalization of (some suitable

subset of) Scala's implicits was unfortunately beyond the scope of the present paper. Section 4.3.2 (pages 11-12)