Cover Letter

This cover letter describes the changes made in paper #60 "Scrap your Boilerplate with Object Algebras" for OOPSLA '15 final submission, regarding the reviewers' comments in stage 1 revision.

Besides the specific issues listed below, all the minor ones (typos, grammar and figure positioning) have been fixed.

OOPSLA '15 Review #60A

The paper is well presented, except for the hyphenation, which is horrible in places.

We have addressed the issues with hyphenation now.

It might be interested to compare this with other Aspect-Oriented approaches to traversals as well as AOOD, e.g. Hannemann & Kiczales OOPSLA 2002 (I don't know if it is, because I'm on a plane without the Interwebz) or others from AOP-land.

This is done now, there is a paragraph called "Eliminating Boilerplate in Design Patterns" in related work.

p7 should `StatAlg` extend `ExpAlg`? if not, explain.

We added some text explaining the option not to have `StatAlg` extending `ExpAlg`.

p11 where did the QL programs come from?

The QL programs represent questionnaires describing binary search problem and were generated. We now included this in the text.

OOPSLA ’15 Review #60B

What is the code size difference between the Vanilla and Shy solutions? Please give a rough estimate in the author response (with/without boilerplate that could be generated).

We have added a subsection to the Case Study section detailing the difference in code size between vanilla and Shy solutions (Section 11.4).

What technique was used for generating the boilerplate interfaces? Is the annotation processor in javax used, or what?

We added a new (small) section (Section 10) about the implementation of Shy. This section explains what annotation processor has been used.

Why do you have so many outliers in the diagrams, in particular for fig 19. Did you not take averages? What are the confidence intervals? Did you measure start-up or warmed-up performance?

We didn’t measure warmup time: the benchmarks were run first to warm-up; then to measure warmup-up performance. The initial outliers were probably caused by garbage collection pauses. The current measurements were run with 4GB of memory to mitigate this effect.
This is the commandline used for running the benchmarks.

```
$ java -Xmx4096M -Xms512M  -server -d64 -cp bin:lib/Library.jar:lib/ant
lr-4.2.2-complete.jar:../naked-object-algebras/bin _ast.benchmark.Bench
mark
```

OOPSLA '15 Review #60C

Are top-down traversals supported? How are the desugaring transformations implemented?

We added a new section (Section 8) which discusses how desugaring transformations are implemented. In that section we also discuss some limitations, including mentioning that top-down traversals are not supported.

I got the impression that the paper claims compositionality. However, that may be a wrong impression. The extensibility that the paper does claim is indeed supported. Unfortunately it is asymmetric due to single inheritance in Java. Would Scala traits make the approach fully compositional?

Regarding extensibility, we can currently get quite far with Java 8 using interfaces with default methods. Using default methods allows us to get quite a few of of the benefits of multiple-inheritance and traits. Of course if we had traits we could do some things more elegantly. We hope that the discussion about "Independent Extensibility" in Section 9.2 illustrate the benefits of using interfaces with default methods.

There is no explanation of the implementation of the generator or its properties. Is it based on byte-code transformation? Does it actually exist or is it conceptual? I would have expected a bit more explanation about the interaction with the IDE; where does the generated code live?

Please see the new Section on implementation (Section 10), which provides answers to all of the questions above.

Origins of strategic programming in related work is mis-represented.

We have changed the formulation in the text to accurately reflect the origins of strategic programming.

p5: the code in Figure 4 is much shorter You write in multiple places that the code is so short, which seems to refer to its textual size. All the code fragments that are shown are pretty short. It would be more informative if the conciseness would be analytically characterized in terms of number of method definitions or perhaps more fine-grained, in terms of statements or method calls.

In the case-study we (Table 1) we measure the number of factory methods ("expression constructors") that had to be overridden relative to the total number of case.

p5/fig5: that rename is rather lame; why not a proper rename with a new name?

This has been done now: the renaming example in Section 2 is now a proper rename.

p8: Since Object Algebras are factories, the transformation is executed immediately during construction of tree structures. It would be useful at this point to show how such a transformation is invoked and how a transformation pipeline is constructed.

Section 6.1 now includes an example of how to invoke the SubsVar transformation. This transformation transforms into the FreeVars algebra.

p8: "To execute substitution, SubstVar should be subclassed with implementations for `x()`, `e()`, and `expAlg()`." That is a cumbersome way to construct closures; sounds rather boilerplaty; why not show such an invocation.

See client code in Section 6.1.

p9/fig13: That produces a fold/bottom-up transformation. How about top-down traversals? Later in the paper you mention a desugaring, but the code for the desugaring is never shown. Typically, desugarings are defined using top-down traversals, often pattern matching on combinations of constructor patterns (deep pattern matching). Is that supported in this approach?

Please see Section 8, where we now discuss how to do desugaring transformations and limitations of the approach.

For that matter, the transformation algebras represent type preserving transformations; how are translations (between types) expressed?

Object Algebra style translations are expressed by mapping factory calls of one algebra interface to a method calls on an algebra of a different type. In this sense, the queries discussed in the paper are an example of a transformation that maps a language (say, Expressions) to a monoid over some type (a different language). Another example of type translating transformation is discussed in the new section on Desugaring (Section 8). In this case a "larger language" is transformed in to a "smaller language"; the extra language constructs are desugared to combinations of constructs in the smaller language.
The current framework does not support arbitrary type transforming transformations, since it is not clear how to generate default implementations like in the case of the queries. For queries, the target "language" (monoids) is sufficiently regular to generate default implementations. So translations are possible, but they will not be structure-shy (every case needs to be provided by the programmer).

p10: The composition of these transformations is neat! Unfortunately, it seems to be assymetric, if I understand things well. One would really like to have independent implementations of an algebra (syntax) and implementation of a transformation and then compose thge implementations. As I understand the code, the implementation of `UniqueWithLambda<E>` inherits the implementation of `Unique<E>`, but cannot inherit the implementation of a `Unique` for `LamAlgTransform`, since that would require multiple inheritance. Is that a correct analysis? And then it seems an inherent problem for true compositionally in Java. Would Scala traits solve this issue?

Added a subsection about independent extensibility to show symmetric extension of the transformation. Java 8 interfaces with default methods provide almost the same expressivity as Scala traits.

p10: Im quite curious about the implementation of desugaring; why was it not included (in the Appendix)?

Added section on compositional desugarings (Section 8).

p11: Can the desugar transformations be combined into a single traversal? How about deforestation?

The transformations are always executed in a single bottom-up traversal. This is explained in the new section on desugaring. Non-desugared trees are never built, the desugarings are automatically deforested.

p12: "Strategic programming is an approach to data structure traversal, which originated in term rewriting [3, 24, 25]." This sentence is not historically accurate. Generic traversals in term rewriting were first introduced in [26] (Stratego), which extended the strategies of Elan [3] with 'structure-shy' generic one-level tree traversal operators (all, one, some), which could be combined with strategies in a wide range of generic traversals. [25] describes a core language for rewriting based on that approach. "Scrap your boilerplate" and earlier work of Laemmel et al. are implementations of Stratego generic traversals in Haskell and Java. The extension of ASF with traversals [24] came five years later and post-dates most of the work that is described in the next paragraphs. Dont cite is as the origin of strategic programming!

This has been fixed in the text.