Functional Programming at PhET Interactive Simulations

Sam Reid
University of Colorado
reids@colorado.edu

Abstract

PhET Interactive Simulations at the University of Colorado creates free, open-source educational simulations with engaging and fun user interfaces. While our simulations are typically implemented using imperative programming, we recently experimented with developing a simulation using a functional programming style. This paper provides an overview of how functional programming was applied, the pros and cons and recommendations for future usage at PhET and similar projects.

Keywords Imperative Programming, Functional Programming, Java, Design Patterns, Programming Style

1. Introduction

PhET Interactive Simulations at the University of Colorado[PhET] creates free and publicly available open source software for science education. Simulations (sims) in physics, biology, chemistry, mathematics and other fields help students to visualize, interact with and experiment with different scientific phenomena. The simulations are primarily written in Java and Flash, but we have recently used Flex and Scala. Our sims have been translated into 58 languages and are launched over 2 million times per month.

In attempt to improve our development process, we recently investigated using a functional programming style for the development of a simulation about fractions. This simulation allows the user to create, match and interact with fractions. //TODO: figure

1.1 Promises of Functional Programming

Functional programming focuses on eliminating side effects, and using immutable data structures, higher order functions, recursion, etc. In "Functional Programming for Java Developers", Wampler points identifies several reasons to investigate functional programming as an alternative to imperative programming:

- Good for concurrency
- Good for data management
- Improved modularity
- Easier to provide correct code faster
- Simpler

PhET Interactive simulations currently run in a single thread and don't stand to benefit from safe multi-threading. All other promises of functional programming have the potential to improve our development productivity.

2. Varying Degrees of Functional Programming

Languages like Haskell enforce functional programming. Languages like Java and Scala allow the user to use functional programming and imperative programming interchangeably. This also allows developers a few modes of flexibility: using functional programming in different parts of an application, and using functional programming to different degrees within each usage site. At PhET, we applied functional programming to different parts of the Fractions Intro sim and with varying degrees. This section proceeds from small scale to large scale, primarily because the small scale examples of functional programming are more likely to be widely applicable.

2.1 Immutable Values

Prevent unwanted changes to state. Allows wrapping in a Property wrapper to control + observe changes. Examples: Integer, String, Vector2D, List. See Java book chapter "prefer immutable". Lombok provides an @Data annotation that was very helpful in creating immutable value classes (similar to Scala case classes).

2.2 Reinstantiate instead of maintain

By creating a new instance when necessary instead of maintaining it over time, it relieves the burden of having to test all possible combinations of ways of transitioning to each state. When can you reinstantiate instead of maintain? When transitions are irrelevant. For instance, when creating a level selection screen that only depends on the user's progress. There is no need to maintain this over time and provide setters for each of the parts that change. Instead, just provide a correct constructor and create a new one when the user's progress changes.

2.3 Functional Programming for Data Transformations

Functional programming collections libraries are often rich and provide concise and error-free way for providing data transformations. For instance, filter/map.

2.4 Fully immutable model

For this paradigm, the entire model is represented as a sequence of snapshots of immutable state. This is compatible with Piccolo2D and communication with Piccolo2D can be achieved with Property wrappers around the model instance. A significant disadvantage of this approach is that instance identifiers must accompany the model instances so the stateful piccolo2D instances can track them.

2.5 Fully immutable model + view

For this paradigm, the model is fully immutable and the view is transient and stateless instead of being maintained over time. This offers the most support in terms of minimizing the number of ways of attaining each state, and would provide a feasible foundation for sims that must implement sim sharing, since initial state and actions would be sufficient to recreate the entire sequence of states, and the views can be constructed from them. Also, no mutator methods on the view means only one thing to test, the construction of the view. Previously mentioned approaches require the constructor and setters to both work properly. A main disadvantage of this approach is that we have no suitable graphics + UI library for this, and we would have to recreate a great deal of finely crafted piccolo code. Just creating new piccolo trees at each time step is cost-effective,

but re-drawing the screen at each time step (the default for piccolo) is too costly for older machines. I experimented with trying to dynamically compute the dirty rectangle given two model states or two view states, but my implementation was too brittle and would have required too much maintenance.

3. Potential Maintenance Issues

Any novel or different technology introduces a new maintenance cost for the simple reason that it is different than the pre-existing technology. The main question is whether the new technology is worth the cost, and to mitigate this cost as much as possible by making it easy to maintain (with documentation, etc.)

I have already had to perform several maintenance tasks for Fraction Matcher. First, we wanted to add a back button and animation between level selection screen and game screen for consistency with the new Build a Fraction tab. Both of these additions were straightforward and did not introduce any new cost because this part of the view is standard Piccolo2D code. The new maintenance costs will arise if changes are required to the level generation or game model.

4. Lessons Learned

As with many paradigm shifts, some problems were made significantly easier at the cost of making other issues increasingly difficult. For example, in our application, testing the state transitions and graphical representation became very simple because they were always generated from the same model state. However, this came at the cost of increased complexity of the model.

Better separation of controller components.

Manually generated automatic regression testing:

5. Recommendation on where to apply Functional Programming in PhET Sims, with examples

There are several factors that would increase the value of ”Fully immutable model + view” functional programming for PhET.

- Resume work on simsharing (a process that broadcasts the sim state + usage to the teacher for review)
- Desire for more automated regression testing
- Using languages or APIs that provide better support for functional programming, such as Scala or Lombok-PG
- After discovering or creating graphics/UI API that provides better functional programming support. This may be the case with OpenGL with re-renders the scene at each time step.
- Sims with Record/playback
- Sims with Save/load

In the absence of these factors, it is difficult to justify using the fully immutable model or fully immutable model + view. But PhET stands to benefit significantly from the prior 3 levels of functional programming: immutable values, reinstantiate instead of maintain and functional programming for data transformations.

6. Conclusions

Acknowledgments

Thanks to NSF and the Hewlett Foundation for supporting the PhET Interactive Simulations project and to the other PhET developers John Blanco, Chris Malley and Jon Olson.

References

[PhET] http://phet.colorado.edu