Debugging Reactive Programming with Reactive Inspector

Guido Salvaneschi, Mira Mezini
Technical University of Darmstadt, Germany
{salvaneschi,mezini}@cs.tu-darmstadt.de

ABSTRACT

Reactive programming provides dedicated language abstractions for reactive software, relieving developers from manually updating outputs when the inputs of a computation change. Unfortunately, complementing the new paradigm with proper tools that support coding activities is a vastly unexplored area.

We investigate a primary issue in the field: debugging programs in the reactive style. We propose RP Debugging, a methodology for effectively debugging reactive programs. These ideas are implemented in Reactive Inspector, a debugger for reactive programs integrated with the Eclipse Scala development environment.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive environments

Keywords

Functional-reactive Programming, Debugging

Introduction.

Reactive programming (RP) has been proposed as a viable alternative to the Observer design pattern in developing reactive applications such as graphic user interfaces, animations and event-based systems. The idea behind RP is to support language-level abstractions for signals – time-changing values that are automatically updated by the language runtime. In RP, programmers specify the functional dependency of a signal on other values in the application and changes are automatically propagated when it is required. This way, developers do not risk forgetting to update dependent values and benefit from a programming style that is easier to read – thanks to the declarative approach of RP – and supports composition – as signals can be composed to define other signals. It has been shown that software based on RP is more composable [10]. Our earlier studies also suggest that RP is easier to understand [12] than their counterpart based on the Observer design pattern.

RP witnessed a long incubation in experimental research projects, which clarified the semantics foundations [4] and investigated the issue of providing a sound [6] and efficient [5] implementation.

Since then, researchers proposed several RP implementations such as FrTime [1] (Scheme), Flapjax [10] (Javascript) Scala.react [8] and REScala [13, 3] (Scala), DREAM [9] (Java) – just to mention a few. Recently, concepts from RP have been adopted by a number of front-end Javascript libraries like AngularJS.js, Razor.js, React, by Web frameworks like Scala Lift, and by Microsoft's Rx.

RP abstractions are now well understood and properly supported in a variety of languages. Yet, programmers that want to embrace RP have to face a number of challenges due to the immaturity of the field. A primary issue concerns supporting RP in the entirety of the development process through a proper tool ecosystem. In particular, novice RP developers struggle for the lack of proper debuggers – essential instruments to fix errors and understand programs since the early age of computing.

Of course, modern IDEs provide support for debugging high-level languages, but, unfortunately, existing debuggers are hardly useful for RP applications. The issue is a *conceptual* one. Existing debuggers have been designed for the imperative programming model and they are unsuitable for the declarative and data flow-oriented model of RP. Concepts like stepping-over statements, breakpointing or inspecting memory changes assume an imperative model where statements execute one after the other and modify memory state. Designing a debugger for RP requires *a paradigm shift*.

We propose a novel debugging technique, RP Debugging, which addresses the urgent needs of developers when debugging applications in the RP style. The key contribution is to adopt the dependency graph among signals – the same model developers adopt to reason about reactive applications – as a primary runtime representation of the program during the debugging process. We provide Reactive Inspector, a reference implementation for RP Debugging, in the form of an Eclipse plugin.[1] A preliminary evaluation based on a controlled experiment with 18 subjects shows that RP Debugging outperforms traditional debugging.

This poster provides an overview of the content available in a full paper to appear in the ICSE 2016 conference [14].

Background.

For illustration, consider the code snippet in Figure 1. It defines two *vars* a and b (Lines 1-2), i.e., reactive values that, in contrast to signals, can be imperatively updated. Line 3 defines a signal s which depends on a and b according to the signal expression a()+b(). The () notation inside signal expressions establishes a dependency among reactive variables. Signals can depend on vars and on other signals, like in Line 4. When a var is updated (Line 8), the signals that depend on the var are automatically updated without programmer intervention (Lines 9-10). It is easy to see that the

---

[1] http://guidosalva.github.io/reactive-inspector/

```
1  val a = Var(1)
2  val b = Var(2)
3  val s = Signal{ a() + b() }
4  val t = Signal{ s() + 1 }
5  println(s.get()) // 3
6  println(t.get()) // 4
7
8  a()=4
9  println(s.get()) // 6
10 println(t.get()) // 7
```
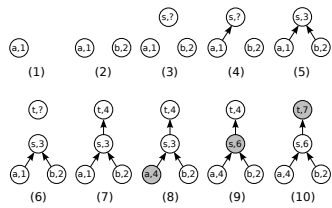
Figure 1: Signals and vars in RP and evolution of the dependency graph in the execution.

Traditional Debugging	RP Debugging
Stepping over statements	Stepping over the dep. graph
Breakpoint on line X	Breakpoint on node X
Inspect memory	Inspect values in the dep. graph
Navigate object references	Navigate signals in the graph
Per-function absolute performance	Per-node relative performance

Figure 2: Traditional debugging vs. RP Debugging.

relation between signals and vars can be described by a directed graph where edges model dataflow dependencies. Operationally, a change in a node of the graph triggers a reevaluation of the signal expressions in the dependent nodes. This is actually the implementation technique adopted by most RP frameworks. Figure 1 (right) shows the evolution of the dependency graph for the code presented before. Lines 1-4 in Figure 1 correspond to node creations (steps 1-2-3-6) and dependency construction (steps 4-5-7). A var update (Line 8) triggers a reevaluation in the graph (steps 8-10).

Simple as it is, this programming paradigm proved effective in managing the complexity of reactive applications in a number of fields, including Web applications [7], interactive GUIs [10], animations [2], wireless sensor networks [11] and robotics [6].

Designing RP Debugging.

When an application is debugged with RP Debugging, the user can visualize the dependency graph and use it as the basic model for reasoning about the application execution.

- At the definition site of the signals, the user can step through the construction of the graph, visualizing the creation of new nodes and of new dependencies among reactive values as soon as they are established.

- When the execution reaches the assignment of a var, the content in the nodes of the dependency graph starts changing. Similar to what developers would do with lines of code in imperative programming, they can step through the update of values in the dependency graph, and control the potentially changing shape of the graph to make sure it reflects their intentions.

- Programmers can set (conditional) breakpoints on the update of a node. The execution continues to traverse imperative and reactive code in the application until the node update is hit. At this point the reactive debugger stops and returns the control to the developer.

- Programmers can inspect the performance of an application on a *per node* basis (absolute performance). Also, in RP Debugging developers can observe the number of times a node outputs a different value as a percentage of reevaluation times (relative performance). This information is particularly useful to detect performance bugs related to erroneous graph configurations.

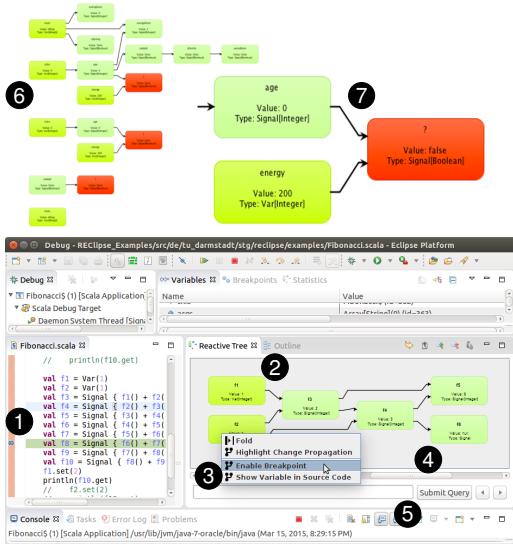Figure 2 shows how the main concepts of traditional debugging

Figure 3: The Reactive Inspector Eclipse plugin.

find a counterpart ($\Longrightarrow$ in the rest) in RP Debugging.

Stepping Users step over code to execute a statement at a time. As the execution is slowed down, the user can check the actual control flow of the application and stop the execution at interesting points. $\Longrightarrow$ Stepping over statements makes little sense for declarative languages. The user can step through the node update propagation in the dependency graph.

Breakpoints Stepping until a certain point in the execution may be tedious. Users can ask the debugger to stop the execution when an instruction in the flow is hit. $\Longrightarrow$ Users can stop the execution when a node in the dependency graph is evaluated and the result of its expression is updated.

Inspect memory During stepping, programmers can inspect the content of the memory, i.e., the active variables in the stack frame and the visible objects on the heap. $\Longrightarrow$ Programmers can inspect the current value of the reactive variables in the graph and inspect the *dependency relations among them*.

Navigate objects structure In OO debuggers, programmers can navigate object fields to inspect the objects structure. $\Longrightarrow$ Programmers can access vars and signals declared in the code to inspect the dependency graph they originate.

Performance In traditional debuggers, performance is analyzed on *per function* bases and it is *absolute* (time spent in each function). $\Longrightarrow$ Programmers analyze *per node* absolute performance and can inspect *relative* performance as fraction of node reevaluations that issued a new value.

Implementation.

Reactive Inspector, our reference implementation of RP Debugging, is an Eclipse plugin integrated with the debugger of the Scala Eclipse IDE [15]. Reactive Inspector supports the REScala [13] reactive language and is made of about 8000 LOC. In Reactive Inspector (Figure 3), when the user is debugging a reactive application (1), the dependency graph is displayed in the GUI (2). Users can set a breakpoint on a node (3). A sliding bar provides access to the previous states in the history of the graph (4) and an input field allows one to specify a query on previous state or as a conditional breakpoint (5). For illustration, we also show the case of multiple active dependency graphs (6) where colors indicate the performance of each node. In the enlarged detail (7) each node provides information about the signal name, type and current value.

1. REFERENCES

[1] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 411–422, New York, NY, USA, 2013. ACM.

[3] J. Drechsler, G. Salvaneschi, R. Mogk, and M. Mezini. Distributed REScala: An update algorithm for distributed reactive programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 361–376, New York, NY, USA, 2014. ACM.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[5] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[6] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[7] Lift reactive web site. http://scalareactive.org/.

[8] I. Maier, T. Rompf, and M. Odersky. Deprecating the Observer Pattern. Technical report, 2010.

[9] A. Margara and G. Salvaneschi. We have a DREAM: Distributed reactive programming with consistency guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 142–153, New York, NY, USA, 2014. ACM.

[10] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[11] R. Newton, G. Morrisett, and M. Welsh. The Regiment Macroprogramming System. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498, 2007.

[12] G. Salvaneschi, S. Amann, S. Proksch, and M. Mezini. An empirical study on program comprehension with reactive programming. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 564–575, New York, NY, USA, 2014. ACM.

[13] G. Salvaneschi, G. Hintz, and M. Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 25–36, New York, NY, USA, 2014. ACM.

[14] G. Salvaneschi and M. Mezini. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, New York, NY, USA, 2016. ACM.

[15] Scala Eclipse IDE. http://scala-ide.org/.