Introducing Lightweight Reactive Values to Java

Tetsuo Kamina
Ritsumeikan University, Japan
kamina@acm.org

Abstract

This paper introduces SignalJ, a lightweight extension of Java with reactive values. A reactive value is a value that can depend on other reactive values, and it is implicitly updated when the depended reactive values are updated. Each reactive value is typed with a *signal type*, which ensures that the dependent reactive values are functional. SignalJ also provides *handlers* of reactive values that are called whenever the monitored reactive value is updated. With these features, SignalJ declaratively specifies dataflows within an application in a functional manner, which enables effective implementation of reactive software. The syntax of SignalJ is almost identical to that of Java 8 except that it introduces a new modifier, `signal`, to represent signal types.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Reactive programming; Java; Control systems

1. Introduction

The current trend of seamless connections between computing systems and their surrounding environment, such as cyber physical systems and the Internet of Things, is making reactive software increasingly important. To effectively implement such reactive systems, it is known that, rather than relying on callbacks, which make control flows tangled and thus hard to understand, it is preferable to directly specify reactions to events, or declaratively specify the dataflows from the environment to the reactions. First class events at the language level have been proposed [1, 3], as well as dataflow language flavors in modern programming languages [2]. Both approaches are now integrated in the REScala programming language [4], which supports both functional reactive values as well as imperative events.

This paper proposes SignalJ[1], a lightweight language extension of Java with reactive values where events are implicit. To represent reactive values, we introduce *signal types*, which ensure that a change in reactive value is implicitly propagated to other reactive values that consume the changed value. Thus, the dataflow is declaratively represented. Events in SignalJ are any changes in reactive values, and their *handlers* implement reactions to such value changes. In this paper, we show intuitive explanations of SignalJ using simple examples. Some details for complicated cases (such as interprocedural dataflows and concurrency) are not described in this paper. In particular, we show how the propagation from the sensor values to the actuation of motors is declaratively specified using the example of a PID control system.

2. A Brief Introduction to SignalJ

Reactive values. Reactive values are used to represent functional dependencies between values in a declarative way; i.e., a reactive value is a primitive value or a value that depends on other reactive values. A reactive value is stored in a variable, which we call a *signal*, declared with the `signal` modifier.

```
1 signal int a = 5;
2 signal int b = a + 3;
3 a++;
4 System.out.println(b);  // 9
5 System.out.println(b.last()); // 8
```

A signal initialized with a primitive value (e.g., `a` in the above code fragment) is a source of the dataflow without further dependencies, and its value can be updated at runtime. A signal that depends on other signals (e.g., `b` in the above code fragment) represents the functional dependency between reactive values, and thus it is considered *final*. The change in the source signal is implicitly propagated to other signals that depend on that source. Thus, initially the value of `b` is 8, but after that the value of `a` is updated by `a++`, the value of `b` becomes 9.

Reactive values in SignalJ provide some useful features. First, line 5 of the above code fragment indicates that we

[1] `https://github.com/tkamina/SignalJ`

SPLASH Companion'16, October 30 – November 4, 2016, Amsterdam, Netherlands

http://dx.doi.org/10.1145/2984043.2989215

```
class Control {
  final double P = ..., I = ..., D = ...;
  Motor motor;
  signal int position = 2000;
  signal int p = position - 2000;
  signal int derivative = p - p.last();
  signal int diff =
    (int)(p*P + p.sum()*I + derivative*D);
  Control(Motor motor) {
    this.motor = motor;
    diff.publish(motor::update);
  }
  void setPosition(int p) { position = p; }
}
class Motor {
  int left, right;
  void update(signal int diff) {
    left = diff < 0 ? MAX+diff : MAX;
    right = diff >= 0 ? MAX : MAX-diff;
  }
}
```

Figure 1. PID control in SignalJ

can obtain the value of the signal at the time of the last update. We call the `last` pseudo method on the signal for this purpose. Thus, line 5 results in 8, which is the value of `b` before execution of `a++`. We can also use the `sum` pseudo method to obtain the summation of the reactive value from its initialization.

Event handlers. We can implement the response to the change in reactive value by means of event handlers. An event handler is a lambda expression or a method reference that is passed to the `publish` pseudo method called on the signal. The following code fragment shows an example:

```
signal int a = 5;
a.publish(e -> System.out.println(e));
a++;  // display 6
```

The handler is called whenever the signal is updated. Thus, the lambda expression passed to the `publish` is called at the subsequent `a++`, and the value of `a`, which is now 6, is displayed. We note that an argument of `publish`, a subscriber, must be either a lambda expression or a method reference that receives the reactive value; i.e., the formal parameter type of the subscriber must be compatible with the reactive value, and this subscriber implicitly takes the reactive value as an argument when it is called.

PID control in SignalJ. One of the examples where reactive values and handlers are effective is a program that controls motors on the basis of sensor values, which are continuously changing. One well-known method to effectively adjust the power of motors in accordance with the sensor values is PID control. Figure 1 shows a simplified PID control in SignalJ. This program controls a two-wheeled robot where the motor of each wheel moves independently. The motor's power difference `diff` is calculated by considering three parameters, `P`, `I`, and `D`, of PID control in each step within the feedback loop, and this power difference determines the power of each motor in the next step of the feedback loop. Note that the *integral* value of PID is obtained by calling `sum` on `p` (the *proportional* value of PID), and the last proportional value (`p.last()`) is used to calculate the `derivative` value of PID.

The feedback loop is formed as follows:

```
Control ctl = new Control(motor);
while (true) {
  int pos = .. // computed from sensor values
  ctl.setPosition(pos); }
```

This loop continuously computes the position (`pos`) from the sensor values and updates the reactive value `position` in `Control`. Because it is a reactive value, its change is propagated to all other values that consume `position`, and thus the value of `diff` is updated. Because `update` in `Motor` subscribes to `diff`, this update triggers execution of `update`, which updates the power of each motor.

In SignalJ, this control of motors is implemented in a declarative way, in that we can explicitly specify the dataflow from the position calculated by the sensors to the power of the motors. This declarative implementation enables us to only specify the equations using the PID parameters and abstracts the details of how the motor powers are calculated in each step in the feedback loop.

3. Conclusions and Future Work

SignalJ, an extension of Java with reactive values, has been proposed. As shown by the simplified PID control system, it declaratively specifies the dataflow in a functional manner, and effectively represents reactions (such as actuating) with respect to environmental changes (obtained by sensor values). The remaining issues are (1) the formalization to precisely describe the semantics of SignalJ and (2) robust evaluation. We are planning to implement a number of reasonably large applications using SignalJ in a variety of domains, including control software, GUI applications, and feed readers, to further study the effectiveness of SignalJ.

References

[1] Patrick Eugster and K.R. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP'09*, volume 5653 of *LNCS*, pages 570–594, 2009.

[2] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *OOPSLA'09*, pages 1–20, 2009.

[3] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*, pages 155–179, 2008.

[4] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging between object-oriented and functional style in reactive applications. In *MODULARITY'14*, pages 25–36, 2014.