Reactive Programming for Stats Modelling and Data Analysis

Jesus Perez Colino

January, 2014

Table of contents

Introduction

- Data Analytics tools that depends on data-bases and statistical models or estimation algorithms, are increasingly more interactive, driven by all sorts of events originating from within the applications and their outside environments.
- Using *traditional programming solutions*, interactive applications are typically constructed around the notion of **asynchronous callbacks** (event handlers).
- Coordinating **callbacks** in increasing sophisticated environments can be difficult since numerous isolated code fragments can be manipulating the same data and their *order of execution* is *unpredictable*.

Introduction

- The **Reactive Programming** (RP) paradigm has been proposed as a solution for developing event-driven applications. It provides abstractions to express programs as reactions to external events and having the language automatically manage the flow of time, data and computation dependencies.
- The RP paradigm is based on the **synchronous dataflow programming** paradigm but with relaxed real-time constraints.
- The RP introduce the notion of **behaviours** for representing continuous time-varying values and **events** for representing discrete values
- The RP allows the structure of the data-flow to be dynamic (i.e., the structure of the data-flow can change over time at run-time) and support higher-order data-flow (i.e., the reactive primitives are first-class citizens).

Introduction

Objective of this Presentation

The main goal of this presentation is to introduce some of the features of **Reactive Programming for Statistical Modelling and Data Analysis**. Additionally, we will show some *examples* of basic callbacks and timers over simple statistical models.

Definition

Definition

Reactive programming is a programming paradigm that is built around the notion of **continuous time-varying values** and **propagation of change**.

- It allows developers to express programs in terms of *what to do*, and let the language automatically manage *when to do it*.
- For example, in *imperative programming*, $a := b + c$ would mean that a is being assigned the result of $b + c$ in the instant the expression is evaluated. Later, the values of b and c can be changed with no effect on the value of a.
- However, in *reactive programming*, values change over time and when they change all dependent computations are automatically re-executed. Therefore, the value of a would be automatically updated based on the new values.

Features

Following *E. Bainomugsha et al.* (2012):

1. Basic Abstractions
2. Evaluation model: Push vs. Pull
3. Glitch Avoidance
4. Lifting Operations: Explicit,Implicit and Manual Lifting
5. Multidirectional propagation

1. Basic Abstractions

- Reactive languages provide two kind of basic abstractions:
  1. **Behaviours**: representing continuous time-varying values
  2. **Events**: representing streams of timed values
- Depending of the host language, the implementation varies. In lazy-languages as *Haskell*, any continuous behaviour is computed lazily. *Fran* or *Yampa* offer dedicated operations working in continuously changing values like *integral* or *derivative*, with a strong importance in modelling or simulation.

2. Evaluation Model: Push vs. Pull

- In the basis of the reactive paradigm is the event, and the automatic propagation of the changes, without any intervention of the programmer
- **Evaluation Model**: How changes are propagated across a dependency graph of values and computations? Who initiates the propagation of the changes?
- Two possible designs of propagation:
  1. **Pull-based**: propagation is demand-driven, or asked when are needed (Fran or Yampa for Haskell)
  2. **Push-based**: propagation is data-driven (Flapjax, Scala.React or FrTime)

3. Glitch Avoidance

- **Glitches** (*Cooper and Krishnamurthi* (2006)) are update inconsistencies that may occur during the propagation of changes. When a computation is run before all its dependent expressions are evaluated, it may result in fresh values being combined with stale values, leading to a glitch.
- *An example*: `var x := 1; var y:= x * 1; var z:= x + y ;` updating `x` could make `z` wrong if `y` is not also up-to-date before the calculation of `z`
- The most important condition for the **Glitch Prevention** is that no model will be updated until everything on which it depends is also up-to-date.
- Also, an efficient reactive implementation should avoid unnecessary recompilations of values that do not change

4. Lifting Operations: Explicit,Implicit and Manual Lifting

Lifting is the conversion of an ordinary operator to a variant that can operate on behaviours.

1. **Implicit Lifting**: when an ordinary operator is applied on a behaviour, it is automatically lifted

$$f(b_1) \rightarrow f_{lifted}(b_1)$$

2. **Explicit Lifting**: when the language provides a set of combinators that can be used to "lift" ordinary operators to operate on behaviours

$$lift(f)(b_1) \rightarrow f_{lifted}(b_1)$$

3. **Manual Lifting**: when the programmer needs to manually obtain the current value of a time-varying value, which can be used with ordinary languages operators.

5. Multidirectional Propagation

- Another property of reactive programming languages is whether propagation of changes happens in one direction (*unidirectional*) or in either direction (*multidirectional*).
- With **multidirectionality**, changes in derived values are propagated back to the values from which they were derived.
- *For example*, writing an expression $F = (C * 1.8) + 32$ for converting temperature between Fahrenheit and Celsius, implies that whenever a new value of either F or C is available the other value is updated (see in examples later).

Solutions

Embedded Language	**Language Host**	**Features**
Fran	Haskell	Pull, Explicit, with G.Avoidance
NewFran	Haskell	Push, Explicit, with G.Avoidance
Yampa	Haskell	Pull, Explicit, with G.Avoidance
Scala.React	Scala	Push, Manual
Trellis	Python	Push, Manual, with G.Avoidance
Rx (Reactive Extensions)	.NET, Python, JS, C++, Ruby	Push, Manual

OpenSource Code in Github

Thank You!