This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

On Serializing and Deserializing FRP-style Interactive Programs

Yoshiki Ohshima

February 15, 2013

1 Introduction

Our group has been developing a GUI framework called KSWorld [7] that draws upon Functional Reactive Programming (FRP) [3]. The core idea is to describe the application logic in terms of the graph of data dependency in a time-aware manner. As in other GUI frameworks, there is a type of basic graphical object, which is called the "`Box`". A `Box` has fields that represent its characteristics such as its location, its graphical appearance (called a shape) etc., and also its dynamic actions. What separates the KSWorld from more conventional GUI framework is that these fields are nodes, or *streams*, in the dependency graph.

The KSWorld is designed to support end-user authoring in the spirit of Etoys [1]. The user can add and remove `Box`es and streams into his project at any time while the system/application is running. When the user is satisfied, a set of `Box`es that the user is interested in will be serialized to be stored or sent over the network, and deserialized at the other end of the pipe. The deserialized project may be merged into the environment that may be quite different from where it was created.

Upon serializing a project made in this manner, which elements in the project to export and how to import them is an interesting question. (But, here is the short answer: "behaviors" should be saved with their current values while the "events" should not be saved with their current values. The behaviors should be re-evaluated upon loading.)

In the following sections, we explain the idea more deeply.

2 Events and Behaviors

In FRP, there are two types of time-varying variables. One is called the "behavior", which represents a continuous value over time whose time domain stretches from $-\infty$ to ∞. In other words, a behavior has *always* a value at any given time, including at the "current time", which denotes the logical time where the system is. (The value of the behavior at the current time is "current value".) Another type is called the "event", which represents a discrete sequence of values

VPRI Memo M-2013-001

that spread over time, but until the first value in the event arrives, the "current value" is undefined. (The terms "behaviors" and "events" can be confusing because these have other meanings.)

As often suggested in discussions [4] [5], the distinction is rather subtle in the implementation; a behavior can be implemented as an event with the latest value cached as the current value. They are connected in the dependency graph, and a change in one triggers the changes in its dependents. The events and behaviors in the KSWorld is implemented thusly; there is a single kind of type called `EventStream`.

3 What to Save?

As described above, the fields of a `Box` are all EventStreams. Upon saving a `Box` into the external form, some values, such as the location of the `Box`, the shape of the `Box`, etc. should be saved so that they can be reconstructed when loaded. On the other hand, events do not retain their current values, as the value of the event is useful only at the time when the event occurs.

4 Examples

Let us take an example. Imagine that there is a `Box` who is reacting to click events, and moving to right by 20 pixels for each click:

```
this.myMover = $$(@click.doE(() -> this.translateBy(P(20, 0))))
```

There are three streams involved; `click`, that is an event and receives a new value when the box is clicked. The `@myMover` stream describes a reaction to `click`; it uses the side-effecting method `translateBy`. In the `translateBy` method, there is a stream that represents the transformation (called `@transformation`) of the box from its container. The double-dollar quote ("$$") serves as a macro and the expression enclosed in it is expanded to an expression that creates an `EventStream`.

(As described in [7], one could write the same effect without using the side-effecting method but more directly, even the presence of self-referencing stream such as in this case. Explaining this is out of scope of this memo.)

The `Box` *always* has a valid value in `transformation`. When this `Box` is exported and later imported, the position of the `Box` should be as it was, and the next click should move the box by 20 pixels from that location.

But what if a behavior is dependent on another behavior? Let us imagine that a `Box` is monitoring the color of the extent of top-level `Box` (also known as the "window") and showing its textual representation:

```
this.extentPrinter =
  $$(this.textContents(@__topContainer__.extent.asString()))
```