Functional Reactive Animation in SVG for the Web via Links

Chi-Feng Chou

Master of Science
School of Informatics
University of Edinburgh
2011

Abstract

This project aims to develop an animation framework for Links to help programmers model animation on the World Wide Web. We incorporate the idea of *Functional Reactive Animation* into the design of our framework. Our work strengthens the simplicity and expressiveness of Links by separating the model and the presentation. It enables programmers to be able to rapidly prototype animations.

Acknowledgements

Fisrtly, I would like to express my gratitude to my supervisor, Mr. Ian Stark, for his clear guidance throughout this project. My work benefited to a great extent from his kind and intelligent advice. I need also thank Mr. Samuel Lindley, for his invaluable insight that helped me to cope with some of the most difficult problems during the development.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Chi-Feng Chou*)

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Animation enriches the user experience on the Web by providing visually appealing and interactive contents. It is widely believed that animations often catch eyeballs better than static images. As the internet plays an ever more important role to convey ideas between business entities and individuals, there is a need of using more productive tools to create animating contents effectively. From a programmer's point of view, a *Domain Specific Language*(*DSL*) or a framework in the programming language would suffice. Links is a young functional programming language that has many features dedicated to Web applications. Having said that, it does not have a specific design to produce animations on the internet. It gives us an incentive to build a framework for Links to serve this purpose.

1.2 Introduction

An animation on the Web involves its static and dynamic presentations as well as the corresponding behaviors when the interaction between it and users is taken into account. The simplest form of it is a series of frames. A common example is GIF images. On the other hand, some animation formats like Flash do not only present the content but also engage user interaction. *Scalable Vector Graphics*(*SVG*)[7] is a XML based language used to describe images and animations. It has elements and attributes supporting both static and dynamic content and user interaction. SVG files are valid XML documents. When it comes to manipulate images in Links, programmers can do so easier to SVG files than other formats by using Links' XML DOM APIs.

Our work does not only beautify Links' syntax to control SVG. It aims at linking SVG and Links under the concept of *Functional Reactive Animation*. One can reckon an animation simply as a continuous, time-varying image. On computers, the presentation of animation is done by continuously drawing on the screen frame by frame. Therefore, a basic way to create animation is on per frame basis. With this insight, as to programming animations, programmers' job typically is to use low level graphic APIs to push the computer correctly displaying each frame step by step. However, this activity often entails too much detail irrelevant to the content of the animation. The problem of this approach is that, without a clear abstraction, developers often need to mix the complexity of the underlying computer system and the logic of animation in the code.

An abstraction that separates the model and presentation of the content thus is needed. *Functional Reactive Animation*(*FRA*)[1] allows programmers to simply declare what the content of the animation is and then let the computer worries about how to present it. The concept is different from developing an ordinary graphic library. Generally speaking, a library does not alter the programming paradigm and mostly emphasises only the encapsulation of the details of the underlying system. Nevertheless, FRA highlights the relation between values and time besides encapsulation. As a result, it constructs a programming paradigm dedicated to animation.

The value of FRA lies in the notations of of *behaviours* and *event streams* that make the presentation of an animation easy to reason about. Behaviors are the values that will be evaluated over continuous time. Changes of time will automatically propagate updates to dependent behaviours to maintain the consistency of the presentation. Event streams represent values that exhibit only in particular times. They are discrete over time regarding to their occurrences. An event in the stream may carry relevant data. For instance, a mouse moving event typically comes with the current position of the mouse. Depending on what kind of event it is and what information it carries, behaviours will be changed by it accordingly. By incorporating FRA into Links, developers can program the logic of animations in a declarative and compositional style and avoid being fuddled by too much detail.

1.3 Aims and Objectives

We set up the following objectives to evaluate the maturity of our project.

- A framework dedicated to creating animation will be implemented. Its APIs should conform to the semantics of Functional Reactive Animation.
- A set of SVG primitives can be created and composed in Links using our framework.
- A few animations will be demonstrated to show the ability to manipulate SVG primitives dynamically.
- We shall be able to show the ability of reacting to input from the user by a few examples.

1.4 Contribution

FRA is being actively researched and many significant works have been published. This project stands on the shoulders of existing works to make contributions:

- We explore the semantics of FRA and make sense of it in the context of Web applications.
- A functional reactive animation framework for Links is created. Four working applications are developed to evaluate our work.
- Given that Links is a strict language, a lazy list library is developed. Issues of inefficiency are tackled by imposing lazy evaluation on the essential data structures.

In the body of this thesis, we will introduce the underlying technologies essential to our framework, namely, FRA, Links and SVG. Then design of the data types, APIs and the internal of the framework will be examined under scrutiny. Next, a few examples using our framework will be demonstrated and explained. Before the conclusion, we will look at the completeness of our work and discuss about the future improvement and related works.

Chapter 2

Background

We will review the basic idea of FRA. An implementation in Haskell called *Fran* will be analysed to explain the notions of behaviours and event streams. Following up, we will discuss the purpose of Links and put an emphasis on the features that are relevant to our project. Finally, we will look at SVG and examine its strength in creating animation.

2.1 Review of Functional Reactive Animation

Functional Reactive Animation is best known from the paper by Conal Elliott and Paul Hudak[1]. An animation library for Haskell called *Fran* was implemented in the paper to demonstrate the idea. In Conal and Paul's paper, they use "Fran" referring to both Functional Reactive Animation and the library. In this paper, we will use "FRA" as the abbreviation for Functional Reactive Animation and use "Fran" exclusively for Conal and Paul's library.

FRA heats up following researches about *Functional Reactive Programming*(*FRP*). FRP refers a programming paradigm that view programming as building a continuous system that will react stimuli from both inside and outside. In the case of Functional Reactive Animation, the system being developed is an animation. FRA exploits the high level declarative programming paradigm of FRP in the domain of animation. The formal semantics of it is described in [1] and further explored in [11]. In some literature, it is referred as *Classic FRP*[15][21] to differentiate from many later works altering some of the semantics or using different representations to optimise the performance[4; 16].

To thoroughly embed the functional reactive programming paradigm into a pro-

gramming language, FRA is often implemented as a *Domain Specific Language*(*DSL*)[11] that behaves as built-in constructs of the "host" language. In Conal and Paul's paper, the "host" language for Fran is Haskell. Fran is then refined by Paul Hudak to implement a DSL called *FAL*[13], which is short for *Functional Animation Language*.

From a programmer's of view, a behaviour is an always updated value, an event stream is a source of events and an FRA program is just a set of mutually-recursive behaviours and events[11]. We will introduce the core concepts of FRA. Some code to assist explanation will be presented in Haskell owing to its concise and powerful syntax.

2.1.1 Behaviours

One of the most important concepts in FRA is behaviours, which are values evaluated over time. It is a polymorphic data type in Haskell. One can think it as the following definition:

```
type Behaviour a = Time -> a
```

The type signature reveals that given a polymorphic type variable `a`, `Behaviour a` can be viewed as a function accepts `Time` and then returns value of type `a`. One critical point which can be observed just from the type is that a behaviour is evaluated over time. In other words, `Time` is fed to the function at all times. Here is an example:

```
time :: Behaviour Real
time = id
```

`time` is of the type `Behaviour Real` and is defined as a identity function that returns whatever it receives. Because behaviours are fed with time continuously, so `time` simply means the time.

A slightly more complicated behaviour borrowed from [1]. Given a behaviour `pi` of the type `Behaviour Real` as well as the `time` behaviour we had in previous example, a higher level "wiggle" behaviour is defined:

```
pi :: Behaviour Real
wiggle :: Behaviour Real
wiggle = sin(pi * time)
```

`pi` is a so called *constant behaviour*, since it is always evaluated to the constant value π in the course of animation. `(*)` operator and `sin` are overloaded to functions that accept `Behaviour Real` and return `Behaviour Real`. `wiggle` is of the type `Behaviour Real` and defined based on the overloaded trigonometric function `sin` and only produces values ranging from -1 to 1 as time passes. This example shows FRA's ability of composing which allows programmers to write high-level behaviours by composing other ones. It also illustrates the power of *overloading*. Overloading in Haskell is done by the *type class* mechanism. In Fran, many built-in types have already been overloaded. It makes programs more readable[12].

2.1.2 Events

.

Besides Behaviour, another key concept in FRA is event streams(or just "events" in some literature). The notation of event streams stems from the need to response internal conditions and external events[12]. An event stream is a time-stamped and sorted sequence of values. It is convenient to express its type in Haskell as:

```
type Event a = [(Time, a)]
```

Like `Behaviour`, `Event` is a polymorphic data type that has a type variable `a`. An event may carry some information with respect to the event itself. For instance, an event of moving is expected to carry the mouse position where a user moves the mouse to. Its type signature is:

```
mMove :: Event(Int, Int)
```

For events do not have additional information, they would be of the type `Event ()`. `()`, read "unit", means that there is no information.

We have seen that behaviours can be composed to create a higher level behaviour. Similarly, there is a set of useful combinators that produces semantically richer event streams. Two common ones are *choice operators* and *predicates*[1; 13].

A choice operator is an operator that assembles multiple event streams in different ways such as performing logical "or" or "and" over them. For "or", the first occurrence in either streams would be taken, whereas for "and", the occurrence would count only if it appears in two streams at the same time.

Predicates, or boolean events[13], test a behaviour of boolean with a given time and give an event stream of occurrences that the behaviour evaluates to true.

2.1.3 Reactivity

Reactivity is how a behaviour changes in response to a Event. It is done by switch combinators and event handlers. A switch combinator would be in the form:

```
b `switch` es
```

Given that `b` is a behaviour and `es` is an event stream yields behaviours as its values(information), then its semantics can be described as initially behaving as `b`, then switches to the behaviour carried by the first event of interested in the event stream `es`, then to the next one, and so on[13].

`es` should contains events with a behaviour as information. If it does not, event handlers can be used to convert information of other kinds to a behaviour. Event handlers are higher order functions that accepts an event stream and an user supplied function. An event in the stream may have information, a handler would deal with the it for every event and return a new stream which can be the right operand of `switch`.

Event Mapping `(==>)` is an important handler that has this type:

```
(==>) :: Event a -> (a -> b) -> Event b
```

Given that a constant behaviour always being evaluated to the color of red:

```
red :: Behaviour Color
```

the expression:

```
mClick ==> (\_ -> red)
```

changes the type of an stream `mClick` from `Event ()` to `Event (Behaviour Color)`. Which means that it replaces the empty information `()` carried with an event with a color behaviour. In fact, another handler is arguably more suitable in this situation:

```
(-=>) :: Event a -> b -> Event b
```

It is a syntactic sugar of `(==>)` by which we can get rid of the lambda function. Henceforth we can simply write:

```
mClick -=> red
```

In fact, either `(==>)` or `(-=>)` can be seen as more specific versions of the generic `(+=>)` handler:

```
(+=>) :: Event a -> (Time -> a -> b) -> Event b
```

This generic handler passes the time stamp and information of an event to the user supplied function that will return a value of interest, with which the original information is replaced.

A event handler returns a transformed event stream. The switch combinator then use it to alter the behaviour. It has the type:

```
switcher :: Behaviour a -> Event (Behaviour b) -> Behavior b
```

The following expression defines `newcolor` as a `Behaviour Color` which exhibits red color until a mouse click, then turns into blue. It forms reactivity by simply composing behaviours and event streams.

```
newcolor = red `switcher` (mClick -=> blue)
```

Composing Behaviours or Events to build higher level behaviours forms the spine of a FRA program. In th end, there will be one top-level behaviour which embeds all dependencies of behaviours and events stream. As it is evaluated with the current time, the effect is similar to unfold and examine every behaviour and event stream which depends on one another.

2.2 Links

Links is a strict, typed functional programming language for the web[3]. The typical structure of a Web application often involves three tiers, which are:

- Scripts and web pages run on the client side in the browser.
  In order to provide consistent user experience, web pages had better follow a set of standards published and maintained by *W3C*. Nevertheless, there is no hard-and-fast rule about which programming languages should be used to write scripts. Nowadays, a large proportion of developers choose *JavaScript*.

- Programs run on the server side to serve requests from clients.
  A wide range of choices of programming languages are here. Some popular ones are *Perl, Ruby, Python, PHP*, just to name a few.

- A database accessible by *Structured Query Language(SQL)*.
  Programs in the second tier issue commands in SQL to the database. SQL is a standardised language and is supported by most of the database.

Traditionally, programs in different tiers are written or controlled in different languages. Links aims to change the situation by allowing developers to write Web applications in one language.

In practice, most server-side web programming languages has syntax that allows programmers to directly embed JavaScript for browser, which will then run on the client side. The compiler/interpreter will bypass the embedded JavaScript whenever seeing it. Links takes a different view on JavaScript. JavaScript is considered as the low-level "machine code" that a part of a Links program will be compiled to. Programmers can not and does not need to code JavaScript along with Links. As to the part of code runs on the server side, it will be compiled and executed by the Links compiler on the server.

On top of that, Links supports Language-integrated database queries. In most mainstream languages, SQL commands are represented as plain strings. The hosting language's major work is to concatenate those strings and send them to the database. The hosting language does not know about the semantics of the commands. From its viewpoint, commands are just strings. While in Links, commands are written in its built-in notation and translated into SQL[5]. The immediate advantage is that the queries can be syntax checked, type checked and even optimised by the compiler.

The features most relevant to our project will be briefly introduced below. More advanced features and examples can be found in [6] and [5].

2.2.1 Basic Syntax

As far as this project is concerned, we will introduce some of Links' basic syntax and some features essential to our framework.

2.2.1.1 Types

The most basic types in Links are `Int`, `Float` , `String` and `XML`. Links honours strong typing that does not allow implicit type coercion. Different sets of arithmetic operators are provided for `Int` and `Float`. Consequently, the following code is not legal:

```
var n = 1.0 + 2;
```

One have to choose operands to be either both `Int`:

```
var n = floatToInt(1.0) + 2;
```

or both `Float`

```
var n = 1.0 +. intToFloat(2);
```

In addition to primitive types, Links supports several forms of composite types. *Lists* and *tuples* and their operations are frequently used in programs. They both can be polymorphic, only that tuples can have elements of different types, whereas lists can not.

Moreover, two advanced composite types, *records* and *variants* are tremendously useful in our framework. Records are named tuples. Each field consists of a pair of label and value such as:

```
var attr = (positionX = 50.0, positionY = 100.0,
            text = "hello");
```

It can be updated by adding a different field:

```
var moreAttr = (stroke = "blue" | attr);
```

The value of an existing field can be overwritten:

```
var changedAttr = (attr with text = "goodbye");
```

Note that variables in Links are immutable and single-assigned. Hence updating `attr` in the example does not alter `attr` but create a copy with the updated value.

While records must have a determined kinds of fields, variants can uses explicit labels to distinguish different sets of possible value[6]:

```
typename RotateParam =
    [|Angle:Float
     |All:(angle:Float, about:(Float, Float))|];
```

In Links, `typename` is used for defining a type synonym. As such a value of `RotateParam` type in the example can be either just an angle or an angle and a center coordinate. Suppose we have a function that rotates an image according to `RotateParam`, which type signature is:

```
sig rotateImage : (Image, RotateParam) ~> Image
```

, then passing either the angle and the center, or just the angle to the function are legal:

```
var newImage1 = rotateImage(oldImage, Angle(45.0));
var newImage2 = rotateImage(oldImage, All(angle = 30.0, about =
    (50.0, i50.0)));
```

Like many functional programming languages, Links supports *pattern matching*. One can pattern match a value in a switch expression, variable definition, function application, etc. One advantage of using pattern matching is to de-construct a value of the composite type and bind its components to variables at the same time, as the implementation of `rotateImage` shows:

```
sig rotateImage : (Image, RotateParam) ~> Image
fun rotateImage(img, para:RotateParam) {
    switch (para) {
        case Angle(a) -> doRotate(img, a, 0, 0)
        case All(a, (x, y)) -> doRotate(img, a, x, y)
    }
}
```

Though it is not necessary to fully understand some of the most state-of-the-art designs in Links' type system, being able to read type signatures would facilitate users to better comprehend and make use our of framework. For instance,

```
sig filterE : ((Float, a) {}~> Bool, Event(a)) -> Event(a)
```

The first pair of parentheses of this function contains parameters needed. The return type follows the ordinary arrow. The first parameter is a predicate that has parameters of a `Float` and a polymorphic type `a` and then returns a value of boolean type. The pair of braces and the tilde arrow in the predicate are part of Links' type system that support effect typing[6]. A function hints that it makes recursive calls by replacing the ordinary arrow with the tilde arrow. Additionally, it annotates effects inside braces of a signature. In this case, `{}` stands for no effects at all.

2.2.1.2 Concurrency

Links implements *Actor Model* on top of JavaScript to achieve client-side concurrency[3]. Actor model is the concurrency model for some languages such as Erlang and Scala. In this model, each process is equipped with a "mailbox", the way to share data is by sending message to each other's mailbox. In Links, three primitives – `spawn, !, receive` – are implemented to handle concurrency.

1. `var pid = spawn{ expressions };`
   A process is created to evaluate `expressions`. A process id is returned and can be used to send message to the newly created process.

2. `pid ! msg;`
   `(!)` operator is the message sending primitive. Its left operand is the process id of the receiving process. `(!)` is asynchronous so that the sender does not wait for the receiver actually getting the message.

3. `receive { ... }`
   The receiving process can wait and listen to incoming messages by using *receive*. It often has the following syntax:

   ```
   receive {
       case pattern1 -> expression1
       case pattern2 -> expression2
       ...
   }
   ```

   An incoming messages is pattern matched against different clauses, the matched one will be executed. If a process just wants to take whatever the next message in its mailbox, then it can simply do:

   ```
   var newMsg = recv();
   ```

4. `var pid = spawnWait{ expressions };`
   `spawn` does not guarantee any order of execution between the calling and the new processes. However, `spawnWait` allows programmer to force the calling process to wait until the newly created process terminates. Synchronous messaging can be implemented indirectly by `spawnWait` a temporary process sending a message asynchronously and waiting until it terminates.

As we will see in section 3.4.2, concurrency plays an crucial role in our internal event handling structure.

2.2.1.3 Xml Manipulation

Links' ability of XML Manipulation is the fundamental technique used in our project to manipulate SVG. It provides built in XML and DOM APIs. Programmers can use them to change the XML document structure, style and content. Event handlers installed to a node in DOM by Links' *l-event attributes* can capture events associated with the node.

Links has convenient syntax for constructing XML data. Programmers can easily embed XML values within the program by *XML Quasi-quotes*[6]. A section of XML code enclosed by XML tags is a quasi. A section of Links code enclosed by a pair of braces insides a quasi can escape from the quasi to be evaluated as Links code rather than XML values. Then the result of evaluation will be embedded back to the quasi to construct a XML value.

2.3 SVG

Scalable Vector Graphics is a XML based language standardized by W3C and is already supported in most of the mainstream browsers. Unlike pixel-based image formats that define every pixel in a picture, it is vector-based. SVG images only consist of geometrical primitives like lines, curves, points and polygons which are defined by mathematical equations rather than pixels. Therefore, a SVG image can be scaled to fit all sizes of web pages neatly.

Besides, SVG is a markup language which allows users to describe an image in a declarative way. Instead of giving a series of drawing instructions for the primitives in an image, one specifies the attributes and the relationship between them and then leaves the browsers to handle the presentation of the whole image. For instance, to draw a circle filled in red, we can just specify the coordinates of the center, the radius and the style of the circle:

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
    <circle cx="60px" cy="100px" r="50px"
    style="fill:white; stroke:red; stroke-width:4"/>
</svg>
```

SVG inherits advantages from XML. Firstly, it is searchable. The data is stored in plain text. Therefore, the information inside is transparent to search engines and is comprehensible for human beings. Secondly, SVG is programmable. It is in essence a legal XML document and thus can be controlled by programming with DOM APIs supported by browsers. It allows us to create animations by programming the DOM structure of SVG in Links.

2.3.1 SVG Builtin Animation Elements

The declarative model of SVG does not only reside in static graphics but also extend to animation. SVG 1.1 supports the ability to change vector graphics over time[8]. This ability incorporates SMIL Animation specification[9], with some constraints and extensions. SMIL, short for Synchronized Multimedia Integration Language(SMIL), is an animation framework that defines a set of XML animation elements which can be integrated to a XML document. In a broad sense, SVG is a host language in terms of SMIL Animation. To animate in SVG, one first chooses either target element/elements or attribute/attributes, and then applies the animation elements and their corresponding attributes to them. There are five animation elements for different functionalities, namely *animate*, *set*, *animateMotion*, *animateColor* and *animateTransform*.

Listing 2.1 demonstrates the usage of *animate* and *animateMotion* to move a rectangle and change its color at the same time.

Listing 2.1:

```
<rect x="0" y="0" width="50" height="30" fill="green">
    <animate attributeName="fill"
        from="red" to="yellow"
        begin="1s" dur="8s" repeatCount="5"/>
    <animateTransform attributeName="transform"
        attributeType="XML"
        type="translate" from="0, 0" to="100, 0" dur="5s"
        additive="sum" fill="freeze"/>
    <animateTransform attributeName="transform"
        attributeType="XML"
        type="scale" from="1" to="3" dur="10s"
        additive="sum" fill="freeze"/>
</rect>
```

`animate` is a commonly used animation element. It specifies gradual transitions of certain types of attributes in the target element. For instance, Listing 2.1 demonstrates applying `animate` to the `fill` attribute of a green rectangle, in an attempt to change the color from red to yellow in seven seconds starting from the first second after the document is loaded. After that, the color changing effect will repeat for five times as specified by `repeatCount`, each time takes seven seconds as well.

Two `animateTransform` elements are apply to the same rectangle. One is for `translate` and the other is for `scale`. For the first five seconds of this animation, the coordinate system is being translated over time, which causes the rectangle position gradually moving rightwards. Moreover, the coordinate system is being scaled over the first ten seconds, which has the effect that the size of the rectangle grows. It is worth noting that the two transformations during the first five seconds are overlapped, hence the effects are "multiplied" in the sense that a series of transformations is in fact matrix multiplication to form one transformation matrix producing the overall effect. Because of the relationship between transformations and matrices, a different order of a series of transformations will result in a different multiplied matrix which produces a different transformation.

SVG animation has many outstanding features. First, like static SVG, the syntax is declarative. Secondly, its declarative syntax does not only offer a means to model animation over time, but also over events. All of the animation elements above can be triggered by internal events(such as the end of a revolving circle animation), and external events(such as a mouse click). With a language such as JavaScript that implements SVG standard DOM event API, events sent to SVG can be programmed to support user interaction.

2.4 Comparison Between SVG and FRA in Links

SVG is very powerful. With the ability to incorporate another programming language like JavaScript to handle user input events, It can create several kinds of animation with user interaction. In this way, however, we mix two languages but do not get doubled benefit. We take the advantage of SVG's declarative syntax to model animation, but only partly. We need also rely on JavaScript to process the logic of responding to external or internal events and organising SVG primitives. JavaScript is a more imperative programming language thus the entire program becomes less declarative: what to be shown and how to show are blended together.

The situation becomes even worse if one wants to control SVG in Links by Links' built-in XML DOM APIs. In Links, a piece of SVG code is of string type or XML type which can be directly embedded in the Links code(section 2.2.1.3). Nevertheless, Links has no way to know the semantics of the embedded XML it carries. For instance, if the "cx" attribute of a "circle" element is miswritten as a "x", the code will still be happily compiled, just the browser will not display it. Anything wrong happening in one language is not detected by another.

The problem here is similar to database programming in most mainstream languages where Links has its way to address the hazard of it(section 2.2). In the case of marrying SVG and Links, we are looking for a way without implementing a new language feature in Links.

As a typical solution in Computer Science, adding a layer of abstraction like a library to hide the details can help get rid of the hassles of SVG syntax. Even so, to alleviate other problems, we need to go further. Our framework for Functional Reactive Animation in Links would fit in this context better in that programs can be written in declarative style in one language.

Chapter 3

Design and Implementation

The key designs of our framework will be revealed in the section. We will first look at behaviours. Next, SVG primitives as behaviours will be discussed. Followed by the skeleton of framework and internal event structure, the implementation of Event streams and reactivity will be explained in detail.

3.1 Design of Behaviours

We start by looking at the type of behaviours.

```
typename Time = Float;
typename Beh(a) = (Time) {}⇝ a;
```

The first line suggests that Time is the synonym of Float. Like Classic FRP(section 2.1.1), our behaviour type `Beh(a)` is the synonym of a function type that takes Time and returns the value of a polymorphic type `a`.

Some of Links' built-in functions can be treated as behaviours naturally as they have compatible representation. For instance,

```
sig sin : (Float) -> Float
sig cos : (Float) -> Float
```

can be seen as:

```
sig sin : Beh(Float)
sig cos : Beh(Float)
```

In the jargon of FRP, the operation that converts a function over values to an analogous function over behaviours is called *lifting*[1]. Haskell saves programmers from using lifting operators ubiquitously in the program by *type class* mechanism, which is Haskell's way to achieve *ad-hoc polymorphism*[17]. For instance, the type `Behaviour Int` is declared to be an instance of type class *Num*. The programmers can use `Behaviour Int` in wherever the *Num* type fits in, such as `(+)` and `(*)` operators. This ability offers powerful expressive syntax to programmers. On the other hand, currently neither type class nor any ad-hoc polymorphism has been supported in Links, thus we have no way to declare `Beh(Int)` as an instance of type class *Num*. Thus as to lifting a value of any type to a behaviour, an unary lifting operator `const` is defined:

```
sig const : (a) -> Beh(a)}
fun const(v) {
    fun (t:Float) { v }
}
```

such that

```
var one = const(1.0);
var path = const("/image/button.jpg");
```

would just give us a behaviour of the type `Beh(Float)` and `Beh(String)`.

For functions with the higher arity, our approach to lift is to define different versions of function that can be applied to arguments of different types such as:

```
sig iAddB : (Beh(Int), Beh(Int)) -> Beh(Int)
sig fAddB : (Beh(Float), Beh(Float)) -> Beh(Float)
```

The two functions above are lifted versions of the `(+)` operator. They add up two `Beh(Int)` and two `Beh(Float)` respectively.

Inevitably, this approach slightly hinders the readability of a program, but we try to reduce the unease to the minimum by choosing meaningful names for them. Also it is encouraged to use of the infix form of binary functions to make an expression more straightforward:

```
var pos = sin `iAddB` const(1.0);
```

There is a set of functions assisting manipulation of behaviours. For the behaviour of a binary tuple, taking the behaviour of its first or second component can be done by:

```
sig fstB : (Beh((a, b))) -> Beh(a)
sig sndB : (Beh((a, b))) -> Beh(b)
```

`fstB` takes a behaviour of a pair, then only returns the behaviour of the first component. `sndB` is similar but interested the second component.

Programmers often need to deal with conversions from composite types to behaviours and vice versa. Following functions convert a pair of behaviours to and fro a behaviour of a pair:

```
sig toPairB : (Beh(a), Beh(b)) -> Beh((a, b))
sig toBPair : (Beh((a, b))) -> (Beh(a), Beh(b))
```

`toPairB` binds two behaviours, so that they will be evaluated based on the same time. To the contrary, `toBPair` splits a behaviour of a pair to two behaviours. A complete list of arithmetic and other APIs can be found in Appendix A.

3.1.1 Time Transformation

By default, a behaviour is evaluated regarding to the current time. Suppose we want to make a behaviour be evaluated to the value as at the half time the behaviour would originally being evaluated to, we can achieve it by transforming the time frame to two times longer. It is called *Time Transformation*. The transformed time frame therefore is called *local time frame*. We have transformation utilities:

```
sig slowerB : (Beh(a), Beh(Float)) -> Beh(a)
sig fasterB : (Beh(a), Beh(Float)) -> Beh(a)
```

`slowerB` accepts the two behaviours. The first one is a behaviour which value varies over time, another is a `Beh(Float)` indicating how many times slower the variation will become, i.e. how many times larger one would like the local time frame to be. In a similar sense, `fasterB` is the dual of `slowerB`.

Another transformation does not alter the size of each time frame, but shift it. It makes a behaviour be evaluated with some time earlier.

```
sig delayB : (Beh(a), Beh(Float)) -> Beh(a)
```

3.2 SVG Shapes

Our framework supports most of SVG basic shapes. For each kind of shape, we can apply common attributes to produce animation. For example, rectangle has this type:

```
sig rect : () -> (Attrs) -> (Time) {}⇝ Xml
```

Observing that what is after the last straight arrow is the type synonym of `Beh(Xml)`, we can conclude that a shape function would return a value varying over attributes `(Attrs)` and time. We will call it a `shape behaviour` to distinguish from ordinary behaviours.

To make the type signature more concise, we define a synonym for shape behaviours as:

```
typename SBeh = (Attrs) {}⇝ Beh(Xml);
```

The signature of shape function `rect` becomes to:

```
sig rect : () -> SBeh
```

`Attrs` is a record type include the common attributes over shapes:

Listing 3.1: Attrs

```
1  typename Attrs = (posX:Beh(Float), posY:Beh(Float),
2                    height:Beh(Float), width:Beh(Float),
3                    fill:Beh(String), hrefImg:Beh (String),
4                    stroke:Beh(String), strokeWidth:Beh(Float),
5                    transform:[TForm],
6                    points:Beh(Points),
7                    text:Beh(String),
8                    ffamily:Beh(String),
9                    fsize:Beh(Float),
10                   fweight:Beh(FontWeight),
11                   fstyle:Beh(FontStyle));
```

Notice that each field is of a behaviour type, so that the change of time causes the change of attributes and finally propagates to the shape behaviour.

As shown in the implementation of `rect` in Listing 3.2, a shape function typically takes some attributes of its interest from `Attrs`, then synthesises a XML node.

Listing 3.2: rect

```
sig rect : () -> SBeh
fun rect() {
    fun (attr:Attrs) {
        fun (t:Float) {
            var x = intToString(floatToInt(attr.posX(t)));
            var y = intToString(floatToInt(attr.posY(t)));
            var w = floatToInt(attr.width(t));
            var h = floatToInt(attr.height(t));
            var s = floatToInt(attr.strokeWidth(t));

            var trans = multiTFormString(attr.transform)(t);

            <rect
            transform="{trans}"
            x="{x}"
            y="{y}"
            width="{intToString(w)}"
            height="{intToString(h)}"
            style="fill:{attr.fill(t)};stroke:{attr.stroke(t)};
                   stroke-width:{intToString(s)}" />
        }
    }
}
```

We may use `over` to express the order such that one shape is on top of another if they come across each other:

```
var c = rect() `over` ellipse();
```

`over` puts one XML node before another to produce a XML forest during the evaluation.

Listing 3.3: over

```
sig over : (SBeh, SBeh) -> SBeh
fun over(elm1B, elm2B) {
    fun (attr:Attrs) {
        fun (t:Float) {
            <#>
            {elm2B(attr)(t)}
```

```
7               {elm1B(attr)(t)}
8               </#>
9           }
10      }
11  }
```

3.2.1 Manipulating SVG Shapes

For animation, the most used pair type is a pair of Float's, where they can represent coordinates or pairs of width and height:

```
typename FPair = (Float, Float);
typename Points = [FPair];
```

Having SVG shapes at hand, we offer utilities to manipulate them. All of the functions for SVG shapes manipulation accept a shape behaviour as the first argument. As it is suggested before, adequately using infix form of functions with binary arity can significantly help readability.

One basic function is `at`, which declares the position of a given shape:

```
sig at : (SBeh, Beh(FPair)) -> SBeh
```

The expression:

```
var staticRect = rect() `at` const((100.0, 50.0));
```

evaluated to a rectangle that is always positioned at `(100.0, 50.0)`. The result is a statically positioned shape behaviour.

There are times, the coordinate we have is not one behaviour of a pair value but two behaviours. Then `toPairB`(section 3.1) is needed to convert them to a single behaviour based on the same time. As the following code which creates a rectangle moving to and fro horizontally shows:

```
fun toAndFroRect(x:Float, y:Float) {
    var behX = sin `fMulB` const(x) `fAddB` const(x);
    rect() `at` toPairB(behX, const(y))
}
```

Likewise, we have `sizeof`, which accept the same pattern of arguments only that its second argument means the width and height behaviour.

A set of functions named with the prefix of "with" are defined for modifying some common attributes, to name a few:

```
sig withText : (SBeh, Beh(String)) -> SBeh
sig withPoints : (SBeh, Beh(Points)) -> SBeh
sig withColor : (SBeh, Beh(String)) -> SBeh
```

`withText` sets the text of a text shape, `withPoints` sets the path for a polyline shape, and `withColor` sets the filling color of shapes of many kinds.

3.2.2 Consecutive Transformations

Those shape manipulation functions mostly do not accumulate. If they are called upon the same shape multiple times, the effect would not be carried out one after another. In contrast, only the first issued call would take effect. That is, if `sizeof` is called upon the same shape twice, only the first call holds.

Our framework offers programmers expressive syntax to organise the dependencies between behaviours. It is the norm to combine behaviours into an expression as a higher level behaviour by combinators. Then the expression is bound to a named variable to which can be referred in following expressions. The last expression is produced finally as the top-level behaviour. For example, given the code:

```
var position = const((200.0, 200.0));
var radius = const(50.0);
var sideLen = radius `fMulB` const(2.0);

var square = rect() `at` position
                   `sizeof` toPairB(sideLen, sideLen)
                   `withColor` const("gray");

var circle = ellipse() `at` position
                       `sizeof` toPairB(sideLen, sideLen)
                       `withColor` const("white")
                       `withStrokeWidth` const(5.0);

var circle_on_square = circle `over` square;
```

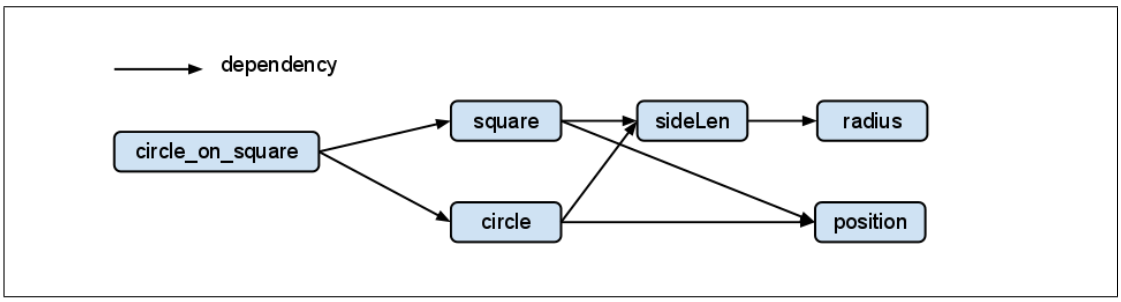

Figure 3.1: Dependencies of expressions

The dependency of expressions forms a *directed acyclic graph* as show in Figure 3.1. The direction of arrows means not only the dependency, but also the order of evaluation. As the result, if an attribute is set up, then any latter write of the same attribute can not overwrite its value.

This does not bring much trouble in practice. Most of time, programmers only want to express what value a behaviour has, rather than how to obtain its value. This conforms to the declarative nature of SVG and captures the idea of FRP.

However, in the case of geometrical transformation, accumulation of effect is needed as they have to be able to be chained. In SVG, the *transform* serves the purpose of the transformation of coordinate system. It supports *translate*, *scale*, *rotate*, *skewX* and *skewY*. They can be chained one after another to produce a series of transformation at once.

In our framework, a set of corresponding functions are provided:

```
sig translate : (SBeh, Beh(FPair)) -> SBeh
sig scale: (SBeh, Beh(FPair)) -> SBeh
sig rotateAbout : (SBeh, Beh(Float), Beh(FPair)) -> SBeh
sig rotate : (SBeh, Beh(Float)) -> SBeh
sig skewX : (SBeh, Beh(Float)) -> SBeh
sig skewY : (SBeh, Beh(Float)) -> SBeh
```

To chain multiple transformations, a list `transform:[TForm]` is kept in `Attrs`(section 3.2) so that whenever a transformation is issued, the related arguments are attached to the list. During the evaluation, the whole list is converted a string in SVG syntax.

3.3 Design of Events and Reactivity

Reactivity plays an important role in an user interface based on animation. Having said that, our preliminary design suffered from the serious problem of time leaks. Re-

sponses to user inputs become much slower as the time goes. It makes the idea of FRA in Web applications unfeasible. We have put an effort to solve issues of this kind in the course of development. In this section, our implementation of the lazy list as the major data structure will be introduced. Then event streams based on the lazy list will be explained. Finally, we will look at the intrinsic design of reactivity.

3.3.1 Lazy List

Unlike Haskell, which adopts ”lazy evaluation” by default, Links is a language that would evaluate values strictly. The disadvantage can be seen from the following scenario: if we want to map all integers in a list to characters and then take only the first character of the list, the map operation will be carried out over the whole list regardless the fact that we only care about the first element. To avoid such inefficiency when processing a potentially very large list, a generic lazy list library is developed in the framework. A lazy list is a list that only presents the values of those elements being accessed. A polymorphic lazy list is defined as:

```
typename LLst(a) = mu x . (() {}~> [|Nil|Cons:(a,x)|]);
```

`mu x . (...)` is the Links' syntax for recursive type definition. It means that `x` represents `(...)` and it can be used inside `(...)` to refer to `(...)` itself recursively.

Most common lazy list operations are supported with almost the same meaning as their strict, built-in list counterpart, just to name a few:

```
sig lmap: ((a) {}~> b, LLst(a)) {}~> LLst(b)
sig lfilter: ((a) {}~> Bool, LLst(a)) {}~> LLst(a)
sig llen: (LLst(a)) {}~> Int
sig lfoldl: ((b, a) {}~> b, b, LLst(a)) {}~> b
```

It is worth noticing that some of the operations are not lazy at all such as `llen` and `lfoldl`. They are not lazy because they can not process without traversing the whole list.

3.3.2 Events

Events, or event streams are the sources of of events. They are discrete steams of values. In Classic FRP(section 2.1), they are conceptually presented as an infinite

list of pairs of time stamp and value. Although infinite lists essentially rely on the language's support for lazy evaluation, we can simulate lazy evaluation of lists in Links with the lazy list library. In addition, there can be a very large number of events in a stream, but generally only those events before a particular time are of interest. So we modified the semantics of event streams from a source of events to a source of events that occurred before a given time. As a result, an event stream is defined as a function that takes an argument of `Time` and returns a lazy list of time-value pairs:

```
typename LEvent(a) = Time {}~> LLst((Float, a));
```

which can be seen as a behaviour:

```
typename LEvent(a) = Beh(LLst((Float, a)));
```

An event comes with a time stamp and a value which is the information with respect to that event. One adjustment to the semantics of Classic FRP that dramatically boosts the efficiency is the choice of making the list in descending order rather than in ascending one according time stamps. This is because most of time, only the most recent events are of concern. It helps that the newer event is always "cons" to the head of the event stream.

All kinds of events are captured by the framework and stored together in a record `ELLst`.

```
typename ELLst = (mmEvts:LLst((Float, FPair)),
                  mdEvts:LLst((Float, ())),
                  muEvts:LLst((Float, ())));
```

The record contains three kinds of events currently captured by our framework, the information they carried are listed bellow:

- Moving the mouse:
  `LLst(Float, FPair)`, under the label of `mmEvts`, which means x-y coordinate when the mouse is moved.
- Pressing the mouse:
  `LLst((Float, ()))`, under the label of `mdEvts`, which has the occurrences of pressing mouse.

- Releasing the mouse:
  `LLst((Float, ()))`, under the label of `muEvts`, which has the occurrences of pressing mouse

The design of separated event lists rather than an universal one mixing all three kinds also concerns efficiency. Considering the amount of moving events greatly outnumbers that of the others, finding the most recent pressing or releasing events in an universal list would take considerable time even in a lazy list.

Listing 3.4:

```
1  fun compose(user) {
2      var mmE = mouseMoveLE(user);
3      var mmB = mouseMoveLB(mmE);
4      var circle = ellipse() `at` mmB
5                             `sizeof` const((50.0, 50.0))
6                             `withStrokeWidth` const(2.0);
7
8      topSVG(svg_child_id, circle,
9          const(800.0), const(600.0))
10 }
```

Listing 3.4 creates an animation in which a circle is chasing the position of the cursor. As illustrated in this program, the recored of event lists is supplied by the framework to the drawing function *compose*(section 3.4.2), which is where programmers embody the model of the animation. The record has event lists of all kinds, Nevertheless, it is not what programmers want. Most of time, a dedicated event stream is the most useful. There are functions that give event streams of interest by looking into the record:

```
sig mouseMoveLE: (Beh(ELLst)) {}⇝ LEvent(FPair)
sig mouseDownLE: (Beh(ELLst)) {}⇝ LEvent(())
sig mouseUpLE: (Beh(ELLst)) {}⇝ LEvent(())
```

`mouseMoveLE` is seen by programmers as the source of mouse moving events. `mouseDownLE` and `mouseUpLE` represent the source of mouse pressing events and mouse releasing events, respectively. If clicks are needed to be captured, using releasing events would suffice as a click just means a releasing event after a pressing one.

Since a cursor is always exhibited on the display and mouse moving events come in continuously. It is nature to view the mouse position as a continuous values, a behaviour. `mouseMoveLB` is defined for this purpose:

```
sig mouseMoveLB: (LEvent(FPair)) -> Beh (FPair)
```

On the first line of Listing 3.4, an event streams of mouse movement `mmE` is initialised. It is used to create the mouse moving behaviour `mmB`. The circle's position is set to `mmB`(line 4). Consequently, the circle would alway be located at the most recent position of the mouse.

3.3.3 Reactivity

Reactivity is achieved by switch combinators that glue behaviours and event streams. As described in 2.1.3, switch combinators have the semantics that exhibiting old behaviour until seeing the first event in an event stream, then switching a new behaviour the event yields, and then doing the same to the rest in the stream.

The primitive switch combinator `lswitcher` is:

Listing 3.5: lswitcher

```
1  sig lswitcher : (Beh(a), LEvent(Beh(a))) -> Beh(a)
2  fun lswitcher(b, evt) {
3      fun (t:Float) {
4          fun f ((te, _)) {
5              te > t
6          }
7          var llst = ldropWhile(f, evt(t));
8          switch (llst()) {
9              case Nil ->
10                 b(t)
11             case Cons((_, x), _) ->
12                 x(t)
13         }
14     }
15 }
```

`ldropWhile`(line 7) discards the events occurred later than the time concerned. From the perspective of our implementation, the semantics is simplified to either that the old

behaviour continues if there is no such event in the event stream, or the new behaviour yielded by the last event in the stream.

Listing 3.6 is drawn to explain the usage of `lswitcher` and event handlers. It displays a circle whose color will change from red to green for good after a mouse click.

Listing 3.6: lswitcher example

```
1 fun compose(user) {
2     var muE = mouseUpLE(user);
3
4     var f = fun (_) { const("green") };
5     var clr = const("red") `lswitcher` mapLE(f, muE);
6
7     var circle = ellipse() `at` const((100.0, 100.0))
8                            `sizeof` const((50.0, 50.0))
9                            `withStrokeWidth` const(2.0)
10                           `withColor` clr;
11
12    topSVG(svg_child_id,
13        circle,
14        const(800.0), const(600.0))
15 }
```

`mapLE` on line 5 is an event handler responding to events. As explained in section 2.1.3, different handlers are interested in different part of information carried in an event. The most general handler, like `(+=>)` in Fran, is `handleLE`. It handles per occurrence of the event by replacing the information with the return value of a user-supplied function. The user-supplied function receives time stamp and information carried by an event and response accordingly.

Listing 3.7: handle

```
1 sig handleLE : ((Float, a){}⇝ b, LEvent(a)) -> LEvent(b)
2 fun handleLE(f, evt) {
3     fun (t:Float) {
4         var f2 = fun ((te, a)) {
5                      (te, f(te, a))
6                  };
7         lmap(f2, evt(t))
8     }
9 }
```

`mapLE` is a rather less general but commonly used handler. It is similar to `handleLE` except that the user-supplied function does not care about the time stamp of the event.

```
sig mapLE : ((a){}⇝ b, LEvent(a)) -> LEvent(b)
```

For situations where the information of an event does not matter, `justLE` helps. It replaces the information with whatever supplied by the user.

```
sig justLE : (LEvent(a), b) -> LEvent(b)
```

In the Listing 3.6, the new color of circle does not depend on the information carried in a mouse releasing event so that line 4 and line 5 can be simplified by `justLE`:

```
var clr = const("red") `lswitcher`
              justLE(muE, const("green"));
```

In addition to these handlers that have equivalent counterparts in Fran, `filterLE` is developed:

```
sig filterLE : ((Float, a){}⇝ Bool, LEvent(a)) -> LEvent(a)
```

It acts as a filter on an event stream and preserves only those events that passes the user-supplied predicate. The user-supplied predicate decides whether to accept an event by the time stamp and information of it.

Events of mouse pressing and releasing do not carry information along with them because they do not have to. We can take a snapshot of the behaviour that has the value interested by the `lsnapshot` operator:

```
sig lsnapshot : (LEvent(a), Beh(b)) ⇝ LEvent((a, b))
sig lsnapshot2 : (LEvent(a), Beh(b)) ⇝ LEvent(b)
```

Since a behaviour evaluates to a value over continuous time. There is always a value when some event happens during that time. In the case of getting position when pressing the mouse, we can take the snapshot of the mouse moving behaviour:

```
var pressPosE = mdE `lsnapshot` mmB;
```

Then we get an event stream contains information of both the pressing event and the position of the mouse at that time. Pressing event does not have anything we are interested, so we can just use `lsnapshot2` to only obtain the position instead. It is common that programmers want a value switching to whatever an event carries. Such operator is called `lstepper` and is a variant of `lswitcher`:

Listing 3.8: lstepper

```
1 sig lstepper : (a, LEvent(a)) -> Beh(a)
2 fun lstepper(a, evt) {
3     lswitcher(const(a), mapLE(const, evt))
4 }
```

The definition of `mouseMoveLB` mentioned in section 3.3.2 illustrates the usage of `lstepper`:

Listing 3.9: mouseMoveLB

```
1 sig mouseMoveLB: (LEvent(FPair)) -> Beh (FPair)
2 fun mouseMoveLB(mmE) {
3     (0.0, 0.0) `lstepper` mmE
4 }
```

The value of mouse position behaviour is initialised at `(0.0, 0.0)`, once the cursor moves inside the display area where moving event handler is installed(section 3.4.1). The mouse position behaviour's value immediately switches to the value carried by the most recent mouse moving event.

3.4 Skeleton of Framework

Our goal is to providing means for developers to program animations easily. Unfortunately, to achieve this goal, we have to at this stage enforce certain layout of the source code. The skeleton is show in Listing 3.10. every function inside will be discussed. Surely the liberty of programming style is sacrificed slightly, however, we will be rewarded with the expressiveness of the model of animation.

Listing 3.10: Layout of code

```
1 fun container() client {
2     ......
3 }
4
5 fun evtMgr(evts:ELLst) client {
6     ......
7 }
```

```
8
9  fun drawInit(user, scene, dura) client {
10     ......
11 }
12
13 fun compose(user) client {
14     ......
15 }
```

3.4.1 Customising Template

The animation program will have to generate a legal HTML document. We provides a basic template called `container` that must be incorporated into the program.

Listing 3.11: Container

```
1  fun container() client {
2      var mouseMgr = spawn { evtMgr((mmEvts = lnil(),
3                                     mdEvts = lnil(),
4                                     muEvts = lnil())) };
5      var user = createE(mouseMgr);
6
7      <#>
8      <div id="svgbasics" >
9      <svg xmlns="http://www.w3.org/2000/svg" version="1.1"
10     xmlns:xlink="http://www.w3.org/1999/xlink"
11     width="800" height="600"
12     viewBox="0 0 800 600"
13
14     l:onmousedown="{
15         mouseMgr ! MDown(intToFloat(getTime(event)), ())}"
16     l:onmouseup="{
17         mouseMgr ! MUp(intToFloat(getTime(event)), ())}"
18     l:onmousemove="{
19         mouseMgr ! MMove(intToFloat(getTime(event)),
20                          (intToFloat(getPageX(event)),
21                           intToFloat(getPageY(event))))}" >
22
23     <g id="{svg_parent_id}"> </g>
24     </svg>
25     </div>
```

```
26      <button id="press1" type="button"
27      l:onclick="{
28          ignore(spawn { drawInit(user, compose, 30000) }) }">
29          draw image</button>
30      </#>
31  }
```

In Listing 3.11, `container` contains of two parts. The first half consists of the work of initialisation. Firstly, a permanent process(`evtMgr`) is spawned. Then the driver for querying events is set up.

The other half is the XML value that will be embedded in the web page. By virtue of XML quasi(section 2.2.1.3), Links code can be conveniently embedded inside XML value. A permanent SVG node is here for having event handlers that receive user input events and send them to the event managing process(`evtMgr`). This is done by using Links' *l-event attributes*(section 2.2.1.3) to install event handlers along with other attributes of the svg element. Currently, there are three kinds of user input events being handled, namely `onmousemove`, `onmousedown` and `onmouseup`. Noticing that because the message passing operator (`!`) inside handlers runs asynchronously(section 2.2.1), the web page could be always responsive.

Besides the permanent SVG, a button is embedded as well(line 26 to line 29). In response to pressing the button, a drawing process will be spawned to evaluate `drawInit` . The button is optional, but programmers should find an adequate way to trigger the drawing process.

3.4.2 Processes

As shown in Figure 3.2, our framework mainly relies on three processes to cooperate with each other – the main process, the event manager and the drawing process.

The main process is in charge of calling the template(`container`) as described in section 3.4.1 to perform framework initialisation, including the spawning event manager and event handlers that accept and forward the user input events to the event manager.

The event manager(`evtMgr`), is defined in Listing 3.12. Its argument `evts` is a `ELLst` record(section 3.3.2) that has all events coming from the main process. `evtMgr` also responses to queries for events collected. A query comes in with a time stamp. Event manager then responses with all the events coming in earlier then the time stamp.

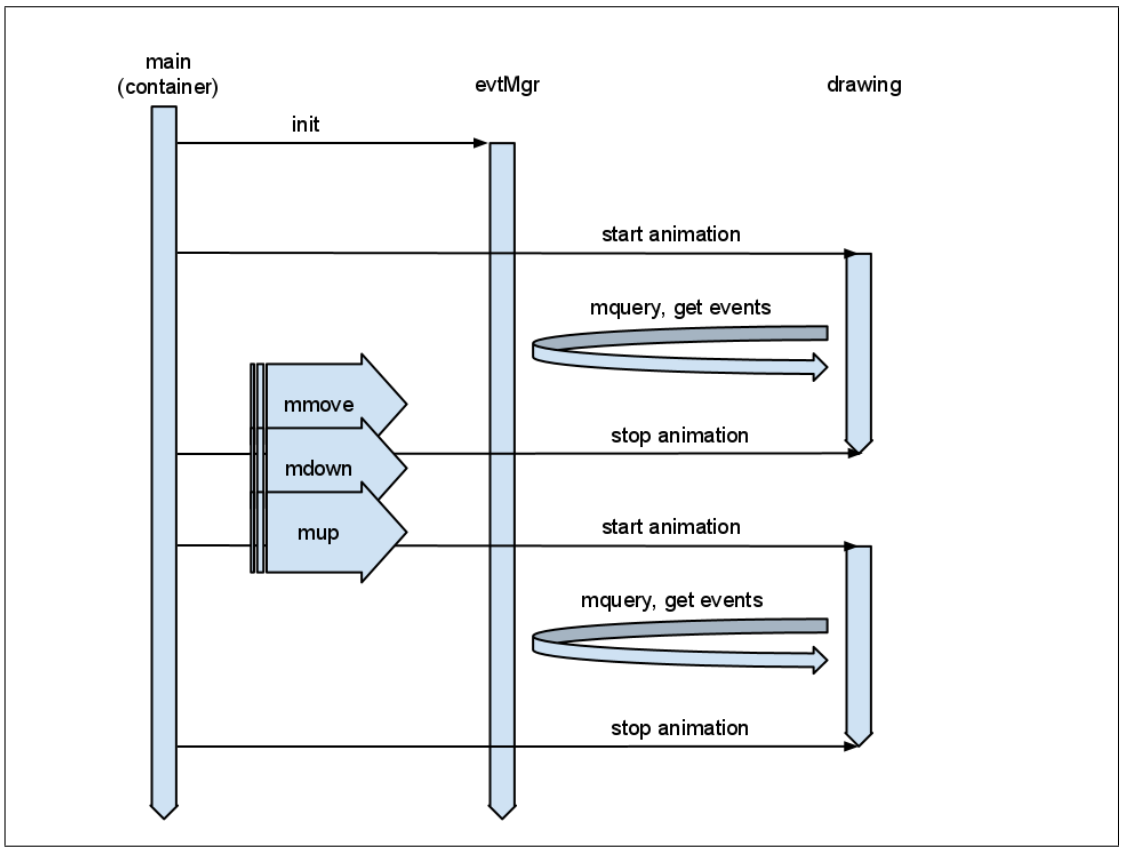

Figure 3.2: Processes and Messaging

Owing to the design of separated, descending ordered lazy lists discussed in section 3.3.2, `evtMgr` reduces the cost of operations on `ELLst` to the minimum. `evtMgr` responses queries for events(`MQuery`) with events in separated lists before the given time. This operation in practical only looks up recent events which were attached to the front of the lazy list when user events `MMove, MDown and Mup` came in.

Listing 3.12: Event manager

```
1  fun evtMgr(evts:ELLst) client {
2      receive {
3          case MQuery((t, proc)) ->
4                var f = fun ((t2, _)) {
5                            t2 > t
6                       };
7                var mmEvts2 = ldropWhile(f, evts.mmEvts);
8                var mdEvts2 = ldropWhile(f, evts.mdEvts);
9                var muEvts2 = ldropWhile(f, evts.muEvts);
10               proc ! (mmEvts = mmEvts2, mdEvts = mdEvts2, muEvts
                     = muEvts2);
```

```
11                evtMgr(evts)
12          case MMove(new) -> # (Float, (Float, Float))
13              evtMgr((evts with mmEvts = lcons(new, evts.mmEvts)))
14          case MDown(new) -> # (Float, ())
15              evtMgr((evts with mdEvts = lcons(new, evts.mdEvts)))
16          case MUp(new) -> # (Float, ())
17              evtMgr((evts with muEvts = lcons(new, evts.muEvts)))
18      }
19  }
```

`container` decides when and how to invoke the drawing process evaluating `drawInit` , which is given in Listing 3.13. Line 5 is the core expression. With the arguments supplied by `container`(line 28 in Listing 3.11), the core expression actually first calls `createE` with the current time, which subsequently queries events from `evtMgr` in a synchronous manner(section 2.2.1.2). Then it supplies the result to the modeling function `compose`.

compose is where programmers use the constructs of behaviours and events to describe the model of animation. It returns the top-level behaviour that encodes all dependencies of expressions inside itself. The framework at this point evaluates the result with the current time to get the XML value `svgXml`, which will be used to update the DOM of the document.

`drawInit` finally calls `draw` which performs almost the same operation. `draw` then calls itself recursively until either the drawing process terminates or the animation expires. In other words, the way we produce the animating effect is continuously evaluating the modeling function and updating the DOM structure of the web page with the result.

Listing 3.13: Drawing process

```
1 fun drawInit(user, scene, dura) client {
2     ......
3     var now = clientTime();
4     var nowf = intToFloat(now);
5     var svgXml = scene(user(nowf))(nowf);
6     replaceNode(svgXml, getNodeById(svg_child_id));
7     ......
8     draw(user, scene, now + dura)
9 }
```

```
10
11 fun createE(mgr) (t:Float) {
12     spawnWait {
13         mgr ! MQuery(t, self());
14         var evts = recv ();
15         fun (t2:Float) {
16             evts
17         }
18     }
19 }
20
21 fun compose(user) {
22     ......
23     topSVG(svg_child_id, ......)
24 }
```

Chapter 4

Examples

Four applications will be presented to support our argument that programmers can create animation in the framework with expressions in succinct syntax. All of them only differ in their modeling functions `compose`.

4.1 Karate Kicks

The Karate Kicks application displays two figures on the screen. The bigger one moves steadily and diagonally. The smaller figure orbits the bigger one. This simple program demonstrates the power of composing behaviours one on top of the other.

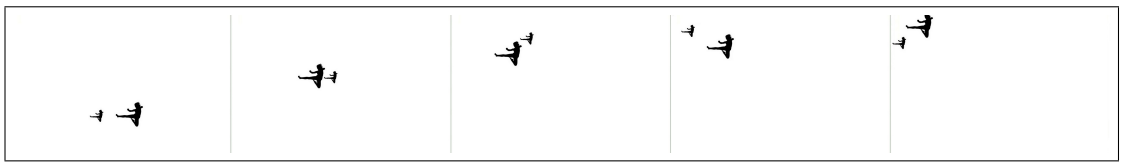

Figure 4.1: Continuous screenshots of Karate Kicks

Listing 4.1: Karate Kicks

```
1 fun compose(user) {
2     # [-1, 1]
3     var wiggle = sin;
4     var waggle = cos;
5
6     # [0, 2]
7     var wiggle2 = const(1.0) `fAddB` wiggle;
8     var waggle2 = const(1.0) `fAddB` waggle;
```

```
9
10      var fig = image() `withImage` const("images/karate.png");
11
12      # ----------------------------------
13      # modeling the bigger figure
14      # ----------------------------------
15      var x = slowerB(const(200.0) `fMulB` wiggle2,
16                      const(5000.0));
17      var y = x;
18      var bigKick = fig `at` toPairB(x, y)
19                        `sizeof` const((100.0, 100.0));
20
21      # ----------------------------------
22      # modeling the smaller figure
23      # ----------------------------------
24      var x2 = x `fAddB` slowerB(wiggle `fMulB` const(100.0),
25                                 const(300.0));
26      var y2 = y `fAddB` slowerB(waggle `fMulB` const(100.0),
27                                 const(300.0));
28      var smallKick = fig `at` toPairB(x2, y2)
29                          `sizeof` const((50.0, 50.0));
30
31      topSVG(svg_child_id,
32          smallKick `over` bigKick,
33          const(800.0), const(600.0))
34  }
```

A few useful primitives are defined at the beginning. `wiggle` and `waggle` are aliases of the built-in trigonometric functions `sin` and `cos`, respectively. As described in section 3.1, they can be treated as behaviours naturally as they have compatible representation. They are evaluated to a value wiggling between -1 and 1 over time. `wiggle2` and `waggle2` adjust the range to 0 to 2.

On line 10, a constant behaviour of the image path is created. Following up are expressions modelling the position and the size of the bigger figure. Firstly, the behaviour of the x coordinate of the figure is defined in terms of `wiggle2` to get a value wiggling between 0 and 400. The use of `slowerB` transforms the time frame so that the value changes slowly. The behaviour of y coordinate is defined to be `x`. The size of the bigger figure does not change over time, thus a constant behaviour suffices. Then the behaviour of the bigger figure `bigKick` is defined with respect to the behaviours of

position and size. As a result, th bigger figure that moves steadily from the top-left corner to the bottom-right is created.

Next, the behaviour of the smaller figure is to be defined. The behaviour of its x coordinate `x2` depends on that of the bigger figure. No matter what `x` becomes, `x2` is shifted away a varying distance from it. The distance is wiggling between the value from -100 to 100. This reveals the power of composing simple behaviours to form a complex one. In the end, the behaviour of smaller figure `smallKick` is created in the same fashion as the bigger one.

4.2 Chasing Cursor

Chasing Cursor is an application that has a circle that chases after the mouse.

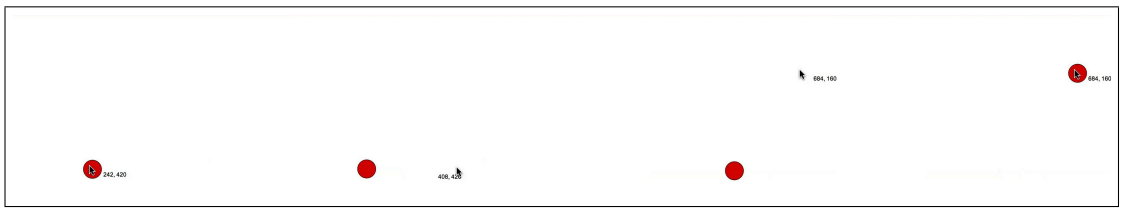

Figure 4.2: Continuous screenshots of Chasing Cursor

The mouse moving behaviour `mmB` is first created and initialised by mouse moving event stream `mmE`. It will be set to the position of the circle. To make the visual effect more significant, we slightly delay the time which the moving behaviour evaluates over by `delayB`. So the position of the circle is always lagging behind the cursor unless the mouse stops moving.

An text revealing current mouse position is displayed on the lower right side of the cursor. `fstB` and `sndB` are used to accessed the x and y coordinate behaviour of the moving behaviour respectively. The coordinate behaviours is converted and concatenated to a string behaviour `coord` by the inline function `appendStringB`. A position behaviour `coordPos` is obtained by adjusting the mouse moving behaviour to the lower right slightly. `coord` is next attached to a shape behaviour `text` which is positioned at `coordPos`.

Listing 4.2: Chasing Cursor

```
1 fun compose(user) {
2     var mmE = mouseMoveLE(user);
```

```
3     var mmB = mouseMoveLB(mmE);
4
5     # ----------------------------------
6     # a circle chasing mouse
7     # ----------------------------------
8     var circle = ellipse() `at` delayB(mmB, const(300.0))
9                            `sizeof` const((50.0, 50.0))
10                           `withColor` const("red");
11
12    # ----------------------------------
13    # text revealing mouse position
14    # ----------------------------------
15    fun appendStringB(sB1, sB2) {
16        fun (t:Float) {
17            sB1(t) ^^ ", " ^^ sB2(t)
18        }
19    }
20    var coord = appendStringB(ftosB(fstB(mmB)),
21                              ftosB(sndB(mmB)));
22    var coordPos = toPairB(fstB(mmB) `fAddB` const(30.0),
23                           sndB(mmB) `fAddB` const(20.0));
24    var displayCoord = text() `at` coordPos
25                              `withText` coord
26                              `withStrokeWidth` const(0.0)
27                              `withFontSize` const(16.0)
28                              `withColor` const("black");
29
30    topSVG(svg_child_id,
31        displayCoord `over` circle,
32        const(800.0), const(600.0))
33 }
```

4.3 Pressing Button

In this example, a simple reactivity is presented. It replaces two button images with each other to mimic the visual effect of pressing button.

Two constant behaviour of image paths are created on line 8 and 9. Line 10 to 17 defines the behaviour before, while, and after pressing the button. Line 17 means that `btnFloat` is exhibited until seeing a mouse pressing event in the mouse down event

Figure 4.3: Screenshots of before and while pressing the button

stream `mdE`. Then the following variation is defined in the inline function `flip`, in which `btnPressed` is displayed until a subsequent mouse releasing event turning up in the mouse up event stream `muE`, then it switches back to `btnFloat`.

Listing 4.3: Pressing Button

```
fun compose(user) {
    var mdE = mouseDownLE(user);
    var muE = mouseUpLE(user);

    # ----------------------------------
    # images flipping to mimic the effect of pressing button
    # ----------------------------------
    var btnFloat = const("images/btn.png");
    var btnPressed = const("images/btn_pressed.png");
    fun flip (td, _) {
        fun afterDown(tu, _) {
            tu > td
        }
        btnPressed `lswitcher`
            justLE(filterLE(afterDown, muE), btnFloat)
    }
    var img = btnFloat `lswitcher` handleLE(flip, mdE);

    var button = image() `at` const((100.0, 100.0))
                         `sizeof` const((90.0, 40.0))
                         `withImage` img;

    topSVG(svg_child_id,
        button,
        const(800.0), const(600.0))
}
```

4.4 Magnifier

The last application demonstrates a magnifier on an image. Users can drag the small gray frame on the image. Meanwhile, what is inside the frame will be magnified and displayed next to the image. Besides operators for reactivity, it also illustrates the usage of transformation functions.

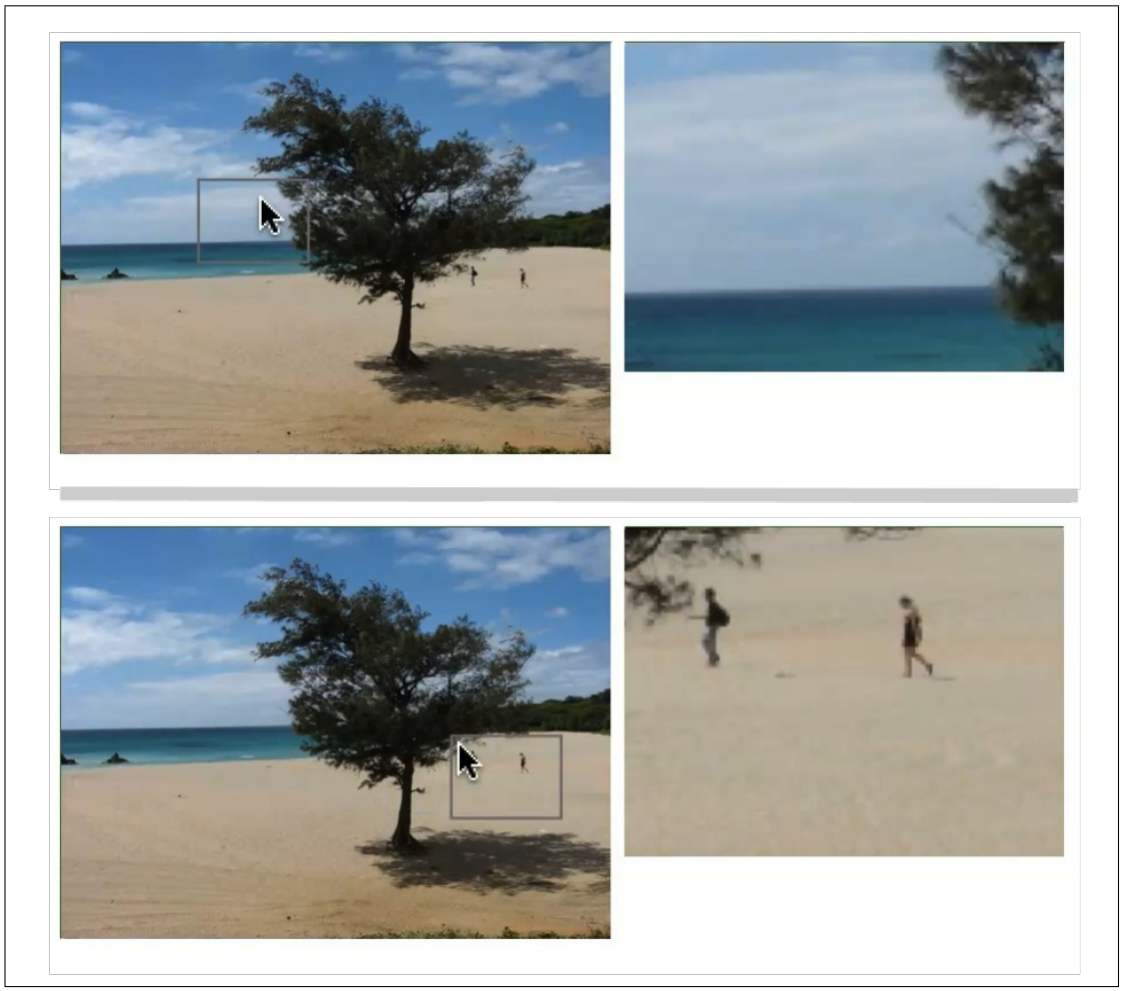

Figure 4.4: Screenshots of Magnifier

The behaviour `drag2mmB` is constantly evaluated to the position of the gray frame's top-left corner. Its value varies when the user drags the frame as it is defined between line 10 to 19. Lines 18 says that it stays at `(100.0, 100.0)` until seeing a mouse pressing event in the mouse down event stream `mdE` because the user presses the mouse.

Then the following variation of `drag2mmB` is defined in the inline function `drag`, in which the position of the frame coincides the position of the cursor as long as the user has not released the mouse. Once it sees an subsequent releasing event in the mouse

up stream `muE`, it immediately takes a snapshot of of the cursor's position(line 16) as the last location after dragging.

Next a static image is created. On top of it, the gray frame is created and located at the position behaviour `drag2mmB`.

Observing that the top-left corner of the frame can be seen as the origin of a new coordinate system, we make a copy of the image and translate the top-left corner of the frame to the origin and then scale it to get a bigger image `imgBig`. In the meantime, we put this transformed copy in a nested svg `magnified`. Because the nested svg has an smaller size than the scaled image, we virtually get a cropped and magnified image.

Listing 4.4: Magnifier

```
1  fun compose(user) {
2      var mdE = mouseDownLE(user);
3      var muE = mouseUpLE(user);
4      var mmE = mouseMoveLE(user);
5      var mmB = mouseMoveLB(mmE);
6
7      # ------------------------------------
8      # dragging effect
9      # ------------------------------------
10     fun drag (td, _) {
11         fun afterDown(tu, _) {
12             tu > td
13         }
14         mmB `lswitcher`
15             mapLE(const,
16                   filterLE(afterDown, muE) `lsnapshot2` mmB)
17     }
18     var drag2mmB = const((100.0, 100.0)) `lswitcher`
19                                          handleLE(drag, mdE);
20
21     # ------------------------------------
22     # image and a small frame on its top
23     # ------------------------------------
24     var img = image() `withImage` const("images/kenting.jpg")
25                       `at` const((0.0, 0.0))
26                       `sizeof` const((400.0, 300.0));
27     var frame = rect() `at` drag2mmB
28                     `sizeof` const((80.0, 60.0))
29                     `withStrokeWidth` const(2.0)
```

```
30                      `withStroke` const("gray")
31                      `withColor` const("none");
32
33      # ----------------------------------
34      # transform the image
35      # ----------------------------------
36      var negx = const(0.0) `fSubB` fstB(drag2mmB);
37      var negy = const(0.0) `fSubB` sndB(drag2mmB);
38      var imgBig = img `translate` toPairB(negx, negy)
39                       `scale` const((4.0, 4.0));
40
41      # ----------------------------------
42      # place the transformed image in a nested svg
43      # ----------------------------------
44      var magnified = svg(imgBig) `at` const((410.0, 0.0))
45                                  `sizeof` const((320.0, 240.0));
46
47      topSVG(svg_child_id,
48          frame `over` img `over` magnified,
49          const(800.0), const(600.0))
50  }
```

Chapter 5

Related Work

Many researches after Classic FRP surround the topic of efficiency. They exert themselves to define more suitable semantics. Conal's latter paper *Push-Pull Functional Reactive Programming* presents a way to implement FRP that combines data-driven(push) and event-driven(pull) evaluation, in which values are recomputed only when necessary[4]. More recent works turn to *Arrowized FRP*(*AFRP*) based on the arrow combinators in Haskell to construct the reactive system[16; 18].

There are efforts being made to bring FRP to general purpose programming languages such as *Frappé* that integrates FRP into the Java programming language[19]. With a similar goal to our framework, *FlapJax* attempts to bring FRP to general Web application on top of JavaScript[23]. Comparing to our work, FlapJax has a broader applicative usage in that it is a general purpose library in JavaScript whereas this project focuses on the domain of animation in Links. Despite that, the two projects adopt similar semantics for behaviours and event streams.

Chapter 6

Future Development

A working framework has been prototyped and tested on a few applications, but there is room for improvement. As to the framework itself, we have a few directions:

- Extending the functionality:
  Many visual effects rely on velocity behaviours. This relies on the semantics of integration that involves accumulating the values that a behaviour had over a period. The framework will be more powerful if it is implemented.

  Currently, the APIs of the framework only deal with past events. It is expected to have ones that handle future events.

  In addition, it will be more advantageous if several combinators like choice operators and predicates(section 2.1.2) can be developed in the future.

- Refining the types:
  Given that Links is a strongly typed language, it is a shame that we have not made the most use of it. Many types check because they are defined imprecisely. For instance, `String` is used at where the type of `Color` would be more appropriate. This raises the difficulty of debugging if a program is incorrectly written.

- Dealing with the space leak:
  We have endeavoured to make sure the performance of the framework to be acceptable. However, we could be at stake of a space leak. A program with a space leak is one that uses memory space so quickly that it detrimentally affects performance[13]. Since we store the whole history of events, the steams keep growing as more and more events are received. To deal with this problem,

we may need to switch to the successors of Classic FRP or other FRP incarnations(section 5).

Moreover, Links is a young programming language. As more and more features are implemented in Links, our framework may be able to use them and thus need to alter the design of its own. For example, one feature we are longing for is a overloading mechanism as it would boost the expressiveness of our framework and therefore fundamentally change our design of APIs.

It is also possible that we can take part of our work to other general purpose functional programming languages. Many of them have proved to be powerful in the context of Web services. When it comes to the user interface for the Web application, a function animation framework can be one of the choices if presentation and user interaction are both taken into account.

Chapter 7

Evaluation

We have reviewed the semantics of Classic FRP. A animation framework is implemented to show how to make sense of FRP in the context of Web applications. The details for encapsulating SVG as behaviours have been discussed. Utilities that assist composing behaviours have been explained. With the consideration of efficiency, we used lazy lists to be the backbone of our design of event streams and reactivity. Finally, four applications have been displayed to demonstrate how our work can facilitate the modeling process of animations. Overall, we have achieved the objectives set up in section 1.3. Namely,

- A framework comes with a set of APIs conforming to the semantics of FRA. The complete list of APIs is listed in Appendix A.
- APIs contains a combinatoric library for most of SVG primitives to create, manipulate and compose SVG shapes. Either static or dynamic content can be modeled by the APIs.
- Four working application are developed to demonstrate the ability to manipulate SVG primitives dynamically and shows the reactivity of the framework.

Chapter 8

Conclusion

Programming SVG directly in Links by DOM manipulation undermines the declarative nature of both languages. Our approach of incorporating FRP into Links is to design a functional animation framework that separates the model and presentation of animations. It enables developers to easily produce animations in the program. Based on our work so far, applications of several kinds can be produced, In the future, we can improve the framework and continue to explore the possibility of FRP.

Appendix A

APIs

```
# ====================
# Type definitions
# ====================
typename Time = Float;

# Behaviour
typename Beh(a) = (Time){}i>a;

typename FPair = (Float, Float);
typename Points = [FPair];

typename TForm = [|Rotate:Beh((Float, FPair))
                  | Translate:Beh(FPair)
                  | Scale:Beh(FPair)
                  | SkewX:Beh(Float)
                  | SkewY:Beh(Float)|];

typename FontWeight = [|Normal:() | Bold:()|];
typename FontStyle = [|Normal:() | Italic:()|];

typename Attrs = (posX:Beh(Float), posY:Beh(Float),
                  height:Beh(Float), width:Beh(Float),
                  fill:Beh(String), hrefImg:Beh (String),
                  stroke:Beh(String), strokeWidth:Beh(Float),
                  transform:[TForm],
                  points:Beh(Points),
                  text:Beh(String),
                  ffamily:Beh(String), fsize:Beh(Float),
                  fweight:Beh(FontWeight),
```

```
           fstyle:Beh(FontStyle));

# Shape behaviour
typename SBeh = (Attrs){}⇝ Beh(Xml);

# Lazy list
typename LLst(a) = mu x . (){}~>[|Nil|Cons:(a,x)|];

# Event stream
typename LEvent(a) = Beh(LLst((Float, a)));

# Record of event stream of all kinds
typename ELLst = (mmEvts:LLst((Float, FPair)),
                  mdEvts:LLst((Float, ())),
                  muEvts:LLst((Float, ())));

# ======================================
# Behaviour APIs(section 3.1)
# ======================================
sig fstB : (Beh((a, b))) -> Beh(a)
sig sndB : (Beh((a, b))) -> Beh(b)
sig toPairB : (Beh(a), Beh(b)) -> Beh((a, b))
sig toBPair : (Beh((a, b))) -> (Beh(a), Beh(b))
sig toListB : ([Beh((a, b))]) -> Beh([(a, b)])

# Uniry lifting operator
sig const : (a) -> Beh(a)

# Integer arithmetic operators
sig iAddB : (Beh(Int), Beh(Int)) -> Beh(Int)
sig iSubB : (Beh(Int), Beh(Int)) -> Beh(Int)
sig iMulB : (Beh(Int), Beh(Int)) -> Beh(Int)
sig iDivB : (Beh(Int), Beh(Int)) -> Beh(Int)
sig iModB : (Beh(Int), Beh(Int)) -> Beh(Int)

# Float arithmetic operators
sig fAddB : (Beh(Float), Beh(Float)) -> Beh(Float)
sig fSubB : (Beh(Float), Beh(Float)) -> Beh(Float)
sig fMulB : (Beh(Float), Beh(Float)) -> Beh(Float)
sig fDivB : (Beh(Float), Beh(Float)) -> Beh(Float)
sig fModB : (Beh(Float), Beh(Float)) -> Beh(Float)
```

```
# Convert Integer behaviour to Float behaviour
sig itofB : (Beh(Int)) -> Beh(Float)
# Convert Float behaviour to Integer behaviour
sig ftoiB : (Beh(Float)) -> Beh(Int)
# Convert Integer behaviour to String behaviour
sig itosB : (Beh(Int)) -> Beh(String)
# Convert Float behaviour to String behaviour
sig ftosB : (Beh(Float)) -> Beh(String)

# Time behaviour
sig time : () -> Beh(Float)

# Time transformation (section 3.1.1)
sig slowerB : (Beh(a), Beh(Float)) -> Beh(a)
sig fasterB : (Beh(a), Beh(Float)) -> Beh(a)
sig delayB : (Beh(a), Beh(Float)) -> Beh(a)

# SVG shape functions (section 3.2)
sig polyline : () -> SBeh
sig polygon : () -> SBeh
sig rect : () -> SBeh
sig ellipse : () -> SBeh
sig text : () -> SBeh
sig image : () -> SBeh
sig svg : (SBeh) -> SBeh

# Shape behaviour manipulation (section 3.2.1)
sig at : (SBeh, Beh(FPair)) -> SBeh
sig sizeof : (SBeh, Beh(FPair)) -> SBeh
sig over : (SBeh, SBeh) -> SBeh

# Set up attributes of shape behaviours (section 3.2.1)
sig withImage : (SBeh, Beh(String)) -> SBeh
sig alongPoints : (SBeh, Beh(Points)) -> SBeh
sig withColor : (SBeh, Beh(String)) -> SBeh
sig withText : (SBeh, Beh(String)) -> SBeh
sig withFontFamily : (SBeh, Beh(String)) -> SBeh
sig withFontSize : (SBeh, Beh(Float)) -> SBeh
sig withFontWeight : (SBeh, Beh(FontWeight)) -> SBeh
sig withFontStyle : (SBeh, Beh(FontStyle)) -> SBeh
sig withStroke : (SBeh, Beh(String)) -> SBeh
sig withStrokeWidth : (SBeh, Beh(Float)) -> SBeh
```

```
# Transformation (section 3.2.2)
sig skewX : (SBeh, Beh(Float)) -> SBeh
sig skewY : (SBeh, Beh(Float)) -> SBeh
sig scale: (SBeh, Beh(FPair)) -> SBeh
sig translate : (SBeh, Beh(FPair)) -> SBeh
sig rotateAbout : (SBeh, Beh(Float), Beh(FPair)) -> SBeh
sig rotate : (SBeh, Beh(Float)) -> SBeh

# ======================================
# Lazy list APIs(section 3.3.1)
# ======================================
sig lnil: () -> LLst(?)
sig lcons: (a, ?) -> LLst(a)
sig lhd: (LLst(a)) {}~> a
sig ltl: (LLst(a)) {}~> LLst(a)
sig llen: (LLst(a)) {}~> Int
sig lmap: ((a) {}~> b, LLst(a)) {}~> LLst(b)
sig lfilter: ((a) {}~> Bool, LLst(a)) {}~> LLst(a)
sig lselect: (LLst(a), Int) {}~> a
sig ltakeWhile: ((a) {}~> Bool, LLst(a)) {}~> LLst(a)
sig ldropWhile: ((a) ~b~> Bool, LLst(a)) -> LLst(a)
sig lfoldl: ((b, a) {}~> b, b, LLst(a)) {}~> b
sig lfoldr: ((a, b) {}~> b, b, LLst(a)) {}~> b
sig lappend: (LLst(a), LLst(a)) {}~> LLst(a)
sig lreverse: (LLst(a)) {}~> LLst(a)
sig lzip: (LLst(a), LLst(a)) {}~> LLst((a, a))
sig lunzip: (LLst((a, b))) {}~> (LLst(a), LLst(b))

# ======================================
# Event APIs(section 3.3.1)
# ======================================

# Event Handlers(section 3.3.3)
sig handleLE : ((Float, a){}~>b, LEvent(a)) -> LEvent(b)
sig mapLE : ((a){}~>b, LEvent(a)) -> LEvent(b)
sig justLE : (LEvent(a), b) -> LEvent(b)
sig filterLE : ((Float, a){}~>Bool, LEvent(a)) -> LEvent(a)

# Switch combinators(section 3.3.3)
sig lswitcher : (Beh(a), LEvent(Beh(a))) -> Beh(a)
sig lstepper : (a, LEvent(a)) -> Beh(a)
```

```
sig lsnapshot : (LEvent(a), Beh(b)) ~> LEvent((a, b))
sig lsnapshot2 : (LEvent(a), Beh(b)) ~> LEvent(b)

# Event stream constructors(section 3.3.2)
sig mouseUpLE: (Beh(ELLst)) {}~> LEvent(())
sig mouseDownLE: (Beh(ELLst)) {}~> LEvent(())
sig mouseMoveLE: (Beh(ELLst)) {}~> LEvent(FPair)
sig mouseMoveLB: (LEvent(FPair)) -> Beh (FPair)
```

Appendix B

Online Materials

This project is hosted on www.github.com. All the examples can be browsed online:

- Karate Kicks:
  Video: http://www.youtube.com/watch?v=V4usz6N5NQ4
  Source: https://github.com/cfchou/links/blob/phase2/cgi-bin/ex_karate.cgi

- Chasing Cursor:
  Video: http://www.youtube.com/watch?v=pQi3QSyc4jA
  Source: https://github.com/cfchou/links/blob/phase2/cgi-bin/ex_cursor.cgi

- Pressing Button:
  Video: http://www.youtube.com/watch?v=9VH0gRQTm4g
  Source: https://github.com/cfchou/links/blob/phase2/cgi-bin/ex_button.cgi

- Magnifier:
  Video: http://www.youtube.com/watch?v=fkvkLnJAYlo
  Source: https://github.com/cfchou/links/blob/phase2/cgi-bin/ex_magnifier.cgi

Bibliography

[1] C. Elliott and P.Hudak, *Functional reactive animation*, ACM SIGPLAN Notices, NewYork, NY, USA: ACM, 1997, pp. 263-273.

[2] *Functional Reactive Programming*, http://www.haskell.org/haskellwiki/, August 19, 2011 retrieved.

[3] E. Cooper, S. Lindley, P.Wadler, and J.Yallop, *Links: web programming without tiers*, Proceedings of the 5th international conference on Formal methods for components and objects, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 266-296.

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

[5] *Links: Linking Theory to Practice for the Web*, http://groups.inf.ed.ac.uk/links/, August 19, 2011 retrieved.

[6] *Links Quick Help: Function types and effects* http://groups.inf.ed.ac.uk/links/quick-help.html, August 19, 2011 retrieved.

[7] *Scalable Vector Graphics*, http://www.w3.org/Graphics/SVG/, August 19, 2011 retrieved.

[8] *Animation SVG 1.1 (Second Edition)*, http://www.w3.org/TR/SVG/animate.html, August 19, 2011 retrieved.

[9] *SMIL Animation*, http://www.w3.org/TR/smil-animation/, August 19, 2011 retrieved.

[10] J. D. Eisenberg, *SVG Essentials: Producing Scalable Vector Graphics with XML*, OReilly Media, Inc., 2002.

[11] Z. Wan, P. Hudak *Functional Reactive Programming from First Principles*, In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, British Columbia, Canada, 2000, pp. 242252. OReilly Media, Inc., 2002.

[12] S. Thompson, *A functional reactive animation of a lift using Fran*, Journal of Functional Programming, 10, pp 245-268

[13] P. Hudak, *The Haskell school of expression: learning functional programming through multimedia*, Cambridge University Press, 2000

[14] 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 Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York, NY, USA, 1-20.

[15] N. Sculthorpe and H. Nilsson, *Safe functional reactive programming through dependent types*, SIGPLAN Not. 44, 9 (August 2009), 23-34.

[16] H. Nilsson, A. Courtney, and J. Peterson, *Functional reactive programming, continued*, In Proceedings of the 2002 ACM SIGPLAN workshop on Haskell (Haskell '02), ACM, New York, NY, USA, 51-64

[17] P. Wadler and S. Blott, *How to make ad-hoc polymorphism less ad hoc*, In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '89), CORPORATE New York, NY Association for Computing Machinery (Ed.). ACM, New York, NY, USA, 60-76

[18] J. Hughes, *Generalising Monads to Arrows*, SCIENCE OF COMPUTER PROGRAMMING, vol. 37, p. 67–111, 1998.

[19] A. Courtney, *Frappé: Functional Reactive Programming in Java*, IN PROCEEDINGS OF SYMPOSIUM ON PRACTICAL ASPECTS OF DECLARATIVE LANGUAGES. ACM, p. 29–44, 2001.

[20] *elerea-2.2.0: A minimalistic FRP library*, http://hackage.haskell.org/package/elerea August 19, 2011 retrieved.

[21] E. Amsden *A Survey of Functional Reactive Programming: Concepts, Implementations, Optimizations, and Applications*, http://blog.edwardamsden.com/2011/05/survey-of-functional-reactive.html August 19, 2011 retrieved.

[22] P. Gergely *Efficient and Compositional Higher-Order Streams*, Functional and Constraint Logic Programming, vol. 6559, J. Mario, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 137-154.

[23] *Flapjax*, http://www.flapjax-lang.org/, August 19, 2011 retrieved.