FRP in C++

Xiangtian Dai, Gregory D. Hager, John Peterson

January 24, 2010

Abstract

Functional Reactive Programming (FRP) is a programming framework for constructing interactive applications in a declarative manner. FRP is now used in animation, vision, robotics and other control systems. *Behaviors* and *Events* – values and conditions which are functions of time – play a key role in FRP. In this paper we present an implementation of main FRP features in C++, an widely used imperative programming language. By exploiting C++'s versatile template class and operator overloading, a strictly typed and concise approach is proposed, and it is taking advantage from both FRP and C++ sides in applications.

1 Introduction

Functional Reactive Programming (FRP) is a programming framework for constructing interactive applications in a declarative manner. Developed originally as a *Domain Specific Language (DSL)* for reactive animation [8], FRP is now also used for vision, robotics and other control systems applications among the programming community [4] [15] [13].

However, all previous implementation of FRP is based on Haskell, a pure functional language, until recently. FRP benefits much from Haskell's higher-order functions, strict, polymorphic type system, and lazy evaluation strategy. However, we demonstrate that it is actually possible to implement FRP in C++, a widely used imperative programming language. Moreover, the C++ approach could benefit from the following:

- Some tasks are more straightforwardly described in a non-functional manner. FRP in C++ allows to code them more expressively and intergrate with other tasks; this way the whole application is more clear and easy to maintain.
- Many of existing libraries are in C/C++. Current hybrid approaches, such as FVision [14], combine Haskell code of FRP and C++ code of libraries through special wrapping. Our approach eliminates boundary between languages, so it is both effective and easy to use.

- Requirements of background for both users and applications are alleviated in our approach: not every application wants an embedded Haskell environment inside after all. Also, there are much more C++ programmers around than Haskell ones.

There has been a recent effort to implement FRP in Java. The result integrates the FRP event/behavior model with the Java Beans event/property model [2]. However, the lack of a strictly typed polymorphic system in Java becomes a limitation. Furthermore, that implementation of behaviors heavily depends on an event propagation ("push" mode), somehow different from existing Haskell ones. The latter mainly make use of the lazy evaluation feature ("pull" mode). This makes significant differences when recursive or mutually recursive behaviors being defined.

In this paper we present a novel C++ approach that aims to provide

- A pure standard C++ implementation. This fits easily with most existing programming environments.
- Concise and straight-forward FRP expressions. Both "functional" and "reactive" aspects are perserved
- Fully extensible modules for generality and customizability from the merits of object oriented programming.
- An analog of well establish FRP implementations. This makes understanding easy and conversions fast. The analog includes
  - A stream-based implementation of Behaviors and Events. It has been formally proved asymptotically "correct" [16].
  - A strictly typed framework. It is easier and safer to do a compile-time type checking than a run-time one.
  - The "Pull" mode stream graphic. Data flow is initiated by sink and propagated to source.

2 A Brief Introduction to FRP Model

Functional Reactive Programming (FRP) is a programming framework for constructing interactive applications in a declarative manner. Since FRP is originally developed from FRAN (Functional Reactive Animation) and FROB (Functional Robotics) as an Embedded Domain Specific Language on Haskell, it is hard to distinguish the core of FRP from the features of Haskell language itself. Nevertheless, here we present our description based on the preliminary FRP User's Manual[5], and all our discussions in this section are in (simplified) Haskell syntax:

- `x :: a` declares that `x` has type `a`

- `()` denotes a void type, which has only one possible dummy value `()`; `(a,b)` denotes a pair type, which has first element of type `a` and second element of type `b`; `[a]` denotes a homogeneous list type of any (include infinite) length whose each element is of type `a`.
- `a -> b` denotes a function type which takes a type `a` argument and gives out the type `b` result. Functions that takes more than one argument are usually treated as higher order functions, for example, `a -> b -> c` denotes a function type which takes a type `a` argument and results in a function which takes a type `b` argument and results in type `c`. In other words, it is a function which takes one argument of type `a` and the other argument of type `b` and results in type `c`.
- *Polymorphic types* describe families of types: if `T` is a polymophic type and `a` is a type, `T a` means "T of a", such as "List of Integers". Types belongs to the same family usually share same functions. One example is that both "List of Integers" and "List of Characters" can apply the same list operations.

2.1 Behaviors

The key terms in FRP are `Signal`, including `Behavior` and `Event`. Conceptually, a `Behavior t` is a value of type `t` that varies over continuous time. The simplest behaviors are constant behaviors, which always has the same value over time, such as `1 :: Behavior Int`, or `red :: Behavior Color`. More complex and interesting examples including animations as `Behavior Picture`, or position vectors of a visual tracker as `Behavior (Int,Int)`.

Sometimes Behaviors are derived from constants or ordinary functions and existing behaviors. This kind of operation is called a "lift", for example,

```
   lift0 :: a -> Behavior a
   lift1 :: (a -> b) -> (Behavior a -> Behavior b)
   lift2 :: (a -> b -> c) -> (Behavior a -> Behavior b -> Behavior
c)
```

Since functions that take no argument always return the same values, `constB`, which makes a constant behavior, is same as `lift0` (on lifting constants to behaviors). `lift1` returns a function which applies the function argment of `lift1` to all values of a behavior to create a new behavior; it works like `map` in functional languages.

Most arithmetic operators can be overloaded in Haskell so that same "lifted" form can be used conveniently, for example,

```
   a, b, c :: Behavior Real
   c = ( a + b ) / 2
```

Here "+" and "/" are both functions lifted to behaviors level, and "2" is used as a constant behavior.

Sometimes value of one behavior at current time relies on values of this and/or other behaviors at previous times. We can form a "delayed" behavior of existing one to allow this. `delayB` delays a behavior for one sampling step, given an initial value.

```
delayB :: a -> Behavior a -> Behavior a
```

2.2 Events

Conceptually, an `Event t` is a time-ordered sequence of event occurrences, each carrying a value of type `t`. For example, a left button press `lbp :: Event ()` and a keyboard press `key :: Event Char`.

```
constE :: a -> Event a
neverE :: Event a
```

`constE` lift a constant to an event that is always happening and `neverE` construct an event that never happens. Lifted functions can also apply on events.

Events can be derived from behaviors and vice versa. `whileE` turns boolean behaviors to events using the rule that the event happens if and only if the value of the behavior is true, and `whileByE` turns behaviors to events by specified functions. `whenE` is same as `whileE` except that it discards repeating events.

```
whileE :: Behavior Bool -> Event ()
whenE :: Behavior Bool -> Event ()
whileByE :: (a -> Maybe b) -> Behavior a-> Event b
stepB :: a -> Event a -> Behavior a
```

Given an initial value, `stepB` holds the value of the most recent event as the current value of the derived behavior:

```
snapshotE_ :: Event a -> Behavior b -> Event b
timeOfE :: Event a -> Event Time
```

`snapshotE_` captures the value of a continuous behavior at the time of an event occurance, and `timeOfE` captures the current time.

An FRP *program* is just a set of mutually recursive behaviors and events.

2.3 Switches

A rich set of operators is provided for users to compose new behaviors and events from existing ones. Some of the operators describe the way they react. Event mapping operators, `==>` and `-=>`, can be used to create an event of behaviors, and the `switchB` and `tillB` operators switch active behaviors according to such event of behaviors.

```
(==>) :: Event a -> (a -> b) -> Event b
(-=>) :: Event a -> b -> Event b
switchB :: Behavior a -> Event (Behavior a) -> Behavior a
tillB :: Behavior a -> Event (Behavior a) -> Behavior a
```

`switchB` switches to the behavior carried by the event each time it occurs; `tillB` switches on the first ocurrence once and for all.

A more complicated example follows:

```
color :: Behavior Color
color = red `tillB` ( lbp -=> blue .|. rbp -=> green )
```

This reads as “initially behave as red, after the left button is pressed change to blue, or after the right button is pressed change to green”.

2.4 Tasks

Another way to express reactivity is using the concept of *Task*. There is actually a family of different types of tasks. Here we only focus on their common aspects. A task combines a behavior and a terminating event, and can be composed either sequentially or in parallel.

```
(>>) ::  Task a b -> Task a c -> Task a c
(|||) ::  Task a x -> Task b y -> Task (a,b) (Either x y)
```

Tasks can be sequenced using `>>`, or put in parallel using `|||`. Sequenced tasks are executed in order: the second one begins to be evaluated when the first one ends (that is its terminating event occurs). Parallel tasks are execuated simultaneously: they begins at the same time, and the end of either one of them also termiantes the other.

```
mkTask ::  Behavior b -> Event x -> Task b x
```

The `mkTask` function constructs a task from its behavior and terminating event:

```
tillT_ ::  Task b x -> Event x -> Task b x
```

`tillT` terminates a task by an extra event. The occurence of this event terminates the associated task if it is not already terminated.

2.5 The Streams Implementation of FRP in Haskell

Practically, an implementation of FRP has to sample continuous behaviors. In other words, behaviors are presented by infinite streams of values sampled at an infinite stream of times [3]:

```
type Behavior a = Stream Time -> Stream a
```

Events are defined similarly, as infinite streams of value of type `Maybe`, each indicates whether or not the event occurs (`Just x` or `Nothing`) and the value it is carrying if it does, sampled at an infinite stream of times:

```
data Maybe a = Nothing | Just a
type Event a = Stream Time -> Stream (Maybe a)
```

Then it’s easy to define behavior and event operations on the base of stream and functional operations, such as `constB` is defined as a function for any input of stream of time, return the stream of repeating some same value. The complexity of manipulating infinite streams is hidden by lazy evaluation feature of Haskell, as the value of an element in a stream will not be computed until it is needed immediately.

2.6 Data Graph View of FRP

The other way to look at the FRP model is the view as data graph. Each vertex, namely behavior or event, provides a typed time-varying value; each directed edge is a data dependency. From time to time, data flows along the edges and is

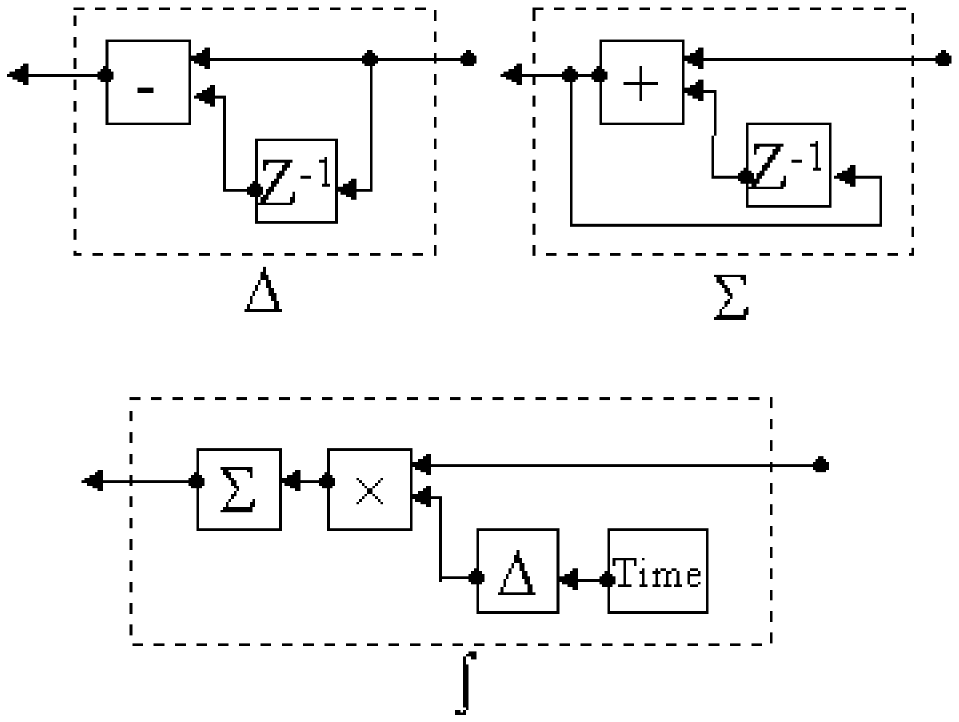


Figure 1: How Euler integral module is composed by a differentiator and an accumulator[2].

processed by some functions at vertices. For example, at any specific time, any lifted `(+)` operator is a vertex whose value is the sum of current values of the two vertices that are on the other sides of the operator's current two incoming edges. The reactivity implies that the graph itself is a function of time.

In this sense, FRP shares common merits with graph or dataflow languages, except that it is non-visual. Its layered modular structure allows rapid prototype developing and reduces errors. Furthermore, since the graph structure changes dynamically instead of being statically predefined, it is able to do more complex tasks.

This data graph view plays an important role in our understanding and implemention of FRP in C++.

3 A User's View of FRP in C++

Obviously, it is neither desirable nor necessary to implement the Haskell syntax in C++. Our implementation keeps most of the basic FRP sementics, while employs the syntax notation as similiar as possible.

[2]The symbol Z^{-1} denotes the delay operation of one sample length.

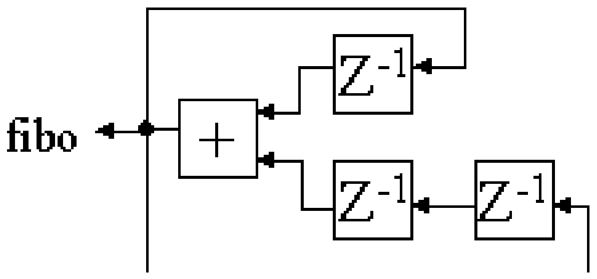


Figure 2: Data graph view of the fibonacci series example.

3.1 Behavior and Event

There are two basic data types in FRP in C++: `Behavior<T>` and `Event<T>`. These mirror their Haskell counterparts, `Behavior` and `Event`. `Behavior<T>` is the type to declare behavior of type `T`, and `Event<T>` is to declare event of `T`. Constants and functions can be lifted to behavior level by `constB` and `constE` or overloaded `liftB` and `liftE` respectively. Common operators such as "+", "-", "*" and "/" already have their lifted version overloaded. Thus, for example, we can write

```
Behavior<double> a, b, c ;
c = ( a + b ) / 2.0 ;
```

Here is an example of fibonacci series, which is defined as each succeeding term is the sum of the two immediately preceding:

```
Behavior<int> fibo ;
fibo = delayB(0)( fibo ) + delayB(0)( delayB(1)( fibo ) );
```

Here the `delayB` function delays a behavior for one sampling step, effectively get immediate preceding of every term. The behavior `fibo` itself is the whole fibonacci series.

The simplest way to run a FRP program is to repeatly evaluate the current value of a behavior, for example, `fibo.run()`.

3.2 Switches and Tasks

The `switchB` and `tillB` operators switch active behaviors according to some events, which can be constructed by `ThenB`.

The same expression in the last section turns out to be:

```
Behavior<Color> color, red, blue, green ;
Event<void> lbp, rbp ;
color = red TillB ( lbp ThenB blue || rbp ThenB green )
```

Here the "`||`" opeartor merges two events.

[3]For illustration purpose, these are not their real prototypes, but are transparently equivalent. `T`, `X` and `Y` are types. Same applies for the other lists of functions.

```
Behavior<T>    constB     ( T ) ;
Behavior<T> (* liftB      ( T (*)() )         )() ;
Behavior<T> (* liftB      ( T (*)(X) )        )( Behavior<X> ) ;
Behavior<T> (* liftB      ( T (*)(X,Y) )      )( Behavior<X>, Behavior<Y> );
Behavior<T>    integral   ( Behavior<T> ) ;
Behavior<T>    derivative ( Behavior<T> ) ;
Event<T>       constE     ( T ) ;
Event<T>       neverE     () ;
Event<T>    (* liftE      ( T (*)() )         )() ;
Event<T>    (* liftE      ( T (*)(X) )        )( Event<X> ) ;
Event<T>    (* liftE      ( T (*)(X,Y) )      )( Event<X>, Event<Y> ) ;
Behavior<T> (* stepB      ( T )               )( Event<X> ) ;
Event<T>       snapshotE  ( Event<X>, Behavior<T> ) ;
Event<Time>    timeOfE    ( Event<X> ) ;
Event<T>    (* whileByE   ( Maybe<T> (*)(X) ) )( Behavior<X> ) ;
Event<T>       whileE     ( Behavior<bool> ) ;
Event<T>       whenE      ( Behavior<bool> ) ;
```

Figure 3: List of some functions on behaviors and events [3].

`Task<T,Y>` is the type of task composed by a behavior of type `T` and a terminating event of type `Y`. `mkTask` function constructs a task from its behavior and terminating event. Here is an example of task. '

```
Task<Color,void> display ;
display = mkTask(blue, lbp) >> mkTask(green, rbp) >> display;
```

The above expression reads as "display blue until left button is pressed, then display green until right button is pressed, then repeat".

```
Behavior<T> switchB     ( Behavior<T>, Event< Behavior<T> > );
Behavior<T> tillB       ( Behavior<T>, Event< Behavior<T> > );
Event<T>    thenB       ( Event<X>, T (*)(X) );
Event<T>    thenB       ( Event<X>, T );
Task<T,Y>   mkTask      ( Behavior<T> x, Event<Y> y ) ;
Task<T,Y>   tillT       ( Task<T,Y> x, Event<Y> y ) ;
Task<T,Y>   operator >> ( Task<T,Y> x, Task<T,Y> y ) ;
Task<T,Y>   operator || ( Task<T,Y> x, Task<T,Y> y ) ;
```

Figure 4: List of some functions on switches and taskes

3.3 Notations on Syntax

Since the pair of parentheses of function application are required in C++, they may accumulate and impair expressiveness of the program (as it arguablely does in Lisp). To counter it, We introduce the "`<<=`" operator for unary function as an additional feature. Like "$" operator does in Haskell, `a <<= b` means a(b). This also makes data-flow intuitive. For the same example of fibonacci series, we can now write

```
fibo = (delayP(0) <<= fibo) + (delayP(0) <<= delayP(1) <<= fibo);
```

C++ allows neither defining new opeartor symbols nor making function application in "infix" style, so synonymous macros are provided for infix usage in addition to existing functions such as `tillB`, while new macro name `ThenB` are introduced for both `==>` and `-=>`[4].

4 Some Examples

In this section, our implementation is put as a part of a developing C++ library for real time image processing and visual tracking, called *XVision2*[6], so it has such integrated fuctions, besides I/O support routines. Data types begin with "XV" prefix are defined by XVision2. Most of them are self-explanatory, like `XVPosition` for position and `XVImageRGB<XV_RGB>` for native RGB image. Beside these, our FRP implementation provides wrapper behaviors, events and behavior combinators for I/O functionalities in XVision2.

This statement declares a behavior combinator `display` that takes its argument, whose type is behavior of RGB image, and display it to a window. Its output is also a behavior of RGB image that is the overall content of the window, as additional drawings may be put on the window in addition to the input behavior. Events of the window, such as button presses, can also be derived from it, for instance, `display.lbp()` returns the left button press event.

```
WindowDisplay<XV_RGB> display ;
```

Video sequences can be represented by behavior of image. Most common used video sources include video capture devices (from input of cameras or video tapes) and stored MPEG files. The following statements declares a behavior of RGB image `videoSource` connected with an IEEE 1394 frame grabber.

```
XVImageRGB<XV_RGB> videoSource ;
XVDig1394<XVImageRGB<XV_RGB> > grabber(DC_DEVICE_NAME,"S1R1");
videoSource = video(grabber);
```

4.1 Image Processing and Visual Tracking

Our example problem is to track the position a specified template image in a video sequence by the means of SSD (sun-of-square differences) tracking. This

[4]The reason of combine two operator into one is that we are running out of meaningful operator symbol space. Because overloaded functions on the same operator symbol can be distinguished by its usage, and in this case both their similarity in meaning and difference in form is obvious, to share the same symbol here does not harm.

process can be illustrated as following:

$$\begin{tikzcd}
\text{position} & \boxed{\text{SSD Tracker}} \arrow[l] & \text{gray image} \arrow[l] & {} \arrow[l] \\
{} & \boxed{\text{RGB to Scalar}} \arrow[l] & \text{color image} \arrow[l] & \boxed{\text{Video Camera}} \arrow[l]
\end{tikzcd}$$

Our program directly reflects above:

```
position = ssdTracker <<= &RGBtoScalar<XV_RGB,int> <<= videoSource;
```

where `position` is of type `Behavior<XVTransState>` (`XVTransState` is a pair of `double` that represent position in translation), and `ssdTracker` is a behavior combinator based on SSD tracker in XVision2, which takes a template image and a source image and returns the position of the former in the latter.

4.2 Interactive Animation

Here we'll see how to express a moving circle on the screen leading by mouse.

First we shall have behaviors of XVPosition for circle center positions, and event of XVPosition for where user clicks in the window. To make the circle move graduallly towards target position, we must have knowledge of how much time has elapsed since last mouse click, and what is total time we allow the circle to travel:

```
Behavior<XVPosition> current, last, target ;
Event<XVPosition> button ;
Behavior<Time> time, delay ;

button = display.lbp() ;
time = timeB - stepTimeOfE(button) ;
delay = Time(1.0) ; // const Behavior
```

Function `stepTimeOfE(x)` is defined as `stepB<Time>(0)(timeOfE(x))`, meaning the time of occurence of last event `x`.

Now the definition of positions are simple: current position is a linear interpolation of last position and target position, or is just target position if it reaches it; target position is where the button last clicks, and "last" position is where the circle is when the button last clicks.

```
current = switchB( (target*time+last*(delay-time))/delay,
                   whileE(time>=delay) ThenB target );
target = stepB(original) <<= button ;
last = stepB(original) <<= snapshotE( button, current ) ;
```

The order of definitions can be arbitrary. In fact, the above ones are mutual recursive.

Finally we can run the expressions by:

```
( display << (liftB(makeCircle) <<= current) <<= background )->run();
```

Where `makeCircle` is an ordinary C++ function returns a circle centered at given position and having a constant radius, and `background` is behavior of image serves as background.

4.3 Robot Control

The simplest way to control a robot is to assign the velocity of it two wheels. Meanwhile, we need input from its world, from sonar, infrared sensor, bumper, or video camera. High level abstraction is needed: If the robot is to go for a colored ball, we normally use a color-blob tracker on the data returned by mounted camera, and it must also be aware at the same time of the data returned by front sonar to avoid bumping into something. The code would be:

```
scoutDrive <<=
     (liftB(go_for) <<= liftB(center) <<= blobFeature <<= videoSource)
     TillB whenE( getScoutSonar(0) <= 10 ) ThenB scoutStop ;
```

`scoutDrive` is a behavior combinator which accepts behavior of `ScoutVel`, which is defined as a pair of `double`, as assigning speed of left and right wheels of Scout robot. `getScoutSonar(0)` returns the front sonar behavior. `go_for` is a ordinary C++ function given the position of tracked color blob feature, tells how robot shall go for it by returning a `ScoutVel`. Its detail depends on the type of camera and it settings. For example, if it is a omni-camera viewing 360 degrees of surroundings from high top of the robot, the code might looks like:

```
ScoutVel go_for( XVPosition obj ) {
  if( distance( obj, dest ) <= threshold ) { // close enough, stop
    return ScoutVel(0,0) ;
  }else if( angle( obj, dest ) <= -PI/4 ) { // at right, turn right
     return ScoutVel(1,-1) ;
  }else if( angle( obj, dest ) >= PI/4 ) { // at left, turn left
     return ScoutVel(-1,1) ;
  }else { // go strait
     return ScoutVel(1,1) * speed ;
  }
}
```

5 Implementation Details

Before an implementation take place, many decisions must be made on what level of similarity on both syntax and semantics of the original one on Haskell should and/or could be mantained, as well as on the ground of possible applications.

5.1 Implementation Considerations

5.1.1 Pure versus Impure

Compare with functional languages such as Haskell, C++ is is quite free: any piece of code may do anything it wants, including side-effect. Only when user limit such usage, our implementation of FRP framework on C++ would help to build declarative application. Especially, caution must be taken with objects having states, as functions may change the internal state of their parameters. Although sometimes it provides shortcuts to accomplish certain tasks, in terms of behaviors and events themselves, it is necessary to imitate pure-functional style: only one definition of a behavior is allowed, and the definition should not be restricted to appear before its first usage, as long as the type signature is declared (this is indeed a limitation of C++). Also, copies of existing behavior should be identical with the original: definition or reference of one be equivalent as another, even some of the variables refering this entity has been out of the scope or destroyed. This feature is accomplished by an envelope class wrapping a real behavior model and handling copy and assignment operations accordingly. The envelope looks as behavior class itself from user is perspective.

5.1.2 Pull versus Push

Maybe the most important implementation issue is the underlying mode of how changes of behaviors and events propagate. Basically, there are two of them: stream based "pull" mode as used in the original Haskell implementation, and GUI-like event propagation "push" mode as used in the Java implementation.

- In the "pull" mode, behaviors are driven from consumer, or the FRP engine. It takes sample either at a timely manner or as fast as possible, provided necessary and unavoidable computations are completed. Events are defined in terms of behaviors.
- In the "push" mode, events are driven from event sources and being propagated and processed. The sample of behaviors can be taken at any time, however, they may only change in time of events thus only after all previous events processed they can behave correctly.

While they are indistinguishable when the computation load is light, "pull" mode is more error-tolerant then "push" mode when computational load is heavy. For example, a robot equipped with a video camera which changes behavior at 30Hz may perform some tracking operation which requires more the 1/30 second to complete for some frames, so it is better to drop frames when necessary. In our implementation, we use the "pull" mode not only to make analog of the Haskell one, but also look for its advantages.

5.1.3 Finite versus Infinite

At first glance, to implement an infinite stream of values has no way in C++, an empirical language does not provide lazy evaluation, which is essential not to compute or store ever-lasting sequences of values. However, further investigation reveals that only current value need to be computed and store for any stream. Although recursively defined streams make use of its previous values, this can be solved by introducing *delay* operator, which constructs new behaviors which are one "tick" delay of old ones. In this way our graph structure is able to emulate resursive behaviors.

5.1.4 Parameters versus Signals

In the Haskell FRP, time and input are treated as parameters to singals, where inputs are classified and united. However, in our implementation, they are treated as seperate behaviors or events. This is for the following reasons:

- One of our designing aim is the openness of the system, in other words, we are not goint to limit the variety of types of input which we cannot predict.
- In practice most behaviors and events are combinations of others, where the combinator itself (such as "+") are not directly functions of time or input.
- For those singals that are directly function of time and/or input, explicit declarations are also easy to write by lifting the function to behavior or event level.

5.1.5 Derived Class versus Template

Behaviors can be any types by definition. There are two mechanisms in C++ to apply polymorphic algorithms: class hierarchy and template class or function. We employ the latter in our implementation, for the following reasons:

- Template classes and functions provide static type checking in compile-time, in addition of possible dynamic run-time type checking. It is not only safer, but also more declarative.
- It would be inconvenience if user has to define their behavior as a derived class, in particular, many behaviors can be represented by basic types in C++, which are not in any class hierarchy unless wrapped.

5.2 Syntax Sugar

It is one thing to implement FRP ideas, while it is another thing to do it in a neat way. The inherent lack of compile-time type inference is an obstacle for all C++ packages that more or less touch the functional world. Obviously, to force users to write out all type signatures invloved in the computation is the easiest solution, but this will not only harm the readablility of the program, but also unnecessarily expose internal implementation details, which harms modularity and scalability. Since our work is not to provide a full-fledge functional package for C++, we found these methods useful to provide adequate "syntax sugar":

- Overloading of existing functions and operators as auto-lifting for expressiveness.
- Defining cast operators, constructors and assignment operators, plus overloading of FRP functions, to hide implementation details.
- Using template partial specialization as static type look-up table, which is often referred as *Type Traits*.

5.2.1 Type Traits

In C++, a type is not a object, so it is not possible to write a function on types. However, when we write signatures, it is not uncommon that one type is a function of another. For example, the "<<=" operator applies its first argument, which is either a (pointer to) function or a function-like object (usually called functor or functoid), to its second argument. To define "<<=" we must declare its return type, which depends on the return type of its first argument. Since there are many forms of function-like object and users can define their own, it is better to write one defining body of "<<=" and use a static type look-up table to establish this type dependency in a more maintainable way. Same applies for "lift".

To implement such type look-up table, "typedef" are used inside a partial specialized template class:

```
template<class T>
struct CombinatorTraits {
  typedef typename T::Result Result ;
};
template<class T1, class T2>
inline typename CombinatorTraits<T1>::Result
operator <<=( const T1& x1, const T2& x2 ) {
  return x1(x2);
}
```

This assumes the first argument of "<<=" belongs to a type T inside which defines T::Result as its result type. While this is true for all function-like object classes defined in our implementation, C++ native (pointer to) function type doesn't fit into this category. To allow "<<=" works on the variety of native function forms, we shall define:

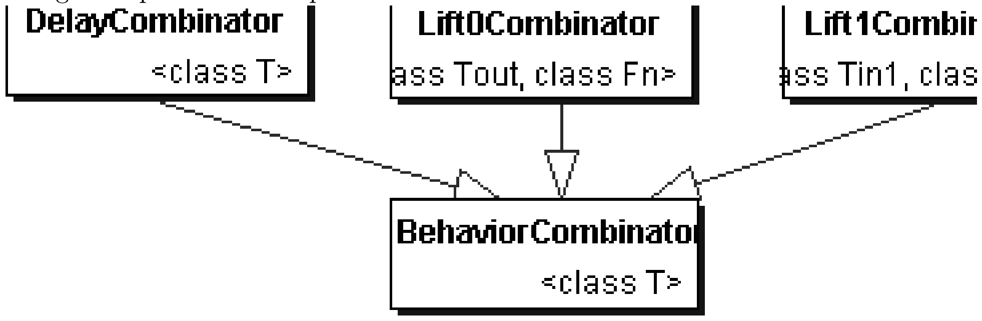
```
template<class Tout, class Tin>
struct CombinatorTraits<Tout (*)(Tin)> {
  typedef Tout Result ; };
  template<class Tout, class Tin>
  struct CombinatorTraits<Tout (*)(const Tin&)> {
    typedef Tout Result ;
    };
```

And so on. Similiarly we can add new entries into the table for third-party functional packages without modifying existing entries and the package itself.

5.3 System Architecture

The understanding above leads to our implementation of FRP in C++. Any kind of behavior and event is abstracted as *module*, while modules are interconnected forming a graph structure, which is the FRP program. Each module has a time-varying value of its type, and maintains its "frame counter" indicating what time of the value it is having. Each module also knows how to update its value in need, including which other modules it depends on. When one of the module is requested for a value that is not ready yet (the existing value the module has is of a previous time), it asks them for their updated values then computes its own based upon those ones. Such update request propagates from the original reqeusting module throughout all those modules in the graph which have an impact on the value of the original one, as the name "pull" indicates. One phase of propagation must be completed before another begins to ensure the synchronism of the graph.

Technical details, such as constructors and destructor, are ignored in all following code pieces due to space limitation.



5.3.1 Basic Modules

Abstract class `GenericBehavior` and its derived templated abstract class `BehaviorBase<T>` defines the behavior modules sample and dependency handling interfaces.

```
class GenericBehavior {
 public:
  virtual void update( Sample now ) = 0 ; // update this and all dependencies
  virtual void compute(void) = 0 ; // do local computation
  virtual void attach( GenericBehavior* b ) = 0 ; // add to dependency
  virtual void detach( GenericBehavior* b ) = 0 ; // remove from dependency
};

template<class T>
```

```
class BehaviorBase :  public GenericBehavior {
 public:
  virtual const T& get(void) const = 0 ; // get current behavior
  virtual void get( T* addr ) = 0 ; // (potentially) delayed get
};
```

Class `BehaviorModule<T>` implements the above interfaces. It serves as root class of all behaviors and events classes, as well as representing constant behaviors itself. `Behavior` is the envelope class mentioned before, so `x :: Behavior Int` now translates as `Behavior<int> x ;`. Most other derived classes of `BehaviorModule` are supposed to override `compute()` member function, which updates time-varying behavior. They include behaviors lifted from C++ functions with 0, 1, 2, or more arguments. Different from Haskell version, functions with no argument are not necessarily constant here.

5.3.2 Lifted Functions

However, a function with one or more arguments can have more than one form of their inputs, among which, the most frequently-used are type `T` and `const T&`. It is impractical to list all combinations of them, so this function type are also templatized:

```
template <class Tout, class Tin1, class Fn>
class Lift1Module :  public BehaviorModule<Tout> {
 protected:
  Fn fn ;
  BehaviorBase<Tin1> *p1 ;
 public:
  void compute(void) { value = fn( p1->get() ) ; }
};
```

Functions with more arguments are lifted in the same manner. To make concise expression instead of to define variables for lifted functions explicitly, auxiliary overloaded functions `lift` are defined to lift any function to behavior level, by making use of a static type looking-up table implemented by type traits:

```
template<class T>
struct LiftTraits { // const lift as default
  typedef T Arg ;
  typedef T Agent ;
  typedef Behavior<T> Result ;
};

template<class T>
```

```
inline typename LiftTraits<T>::Result lift( T f ) {
  return typename LiftTraits<T>::Result
    ( typename LiftTraits<T>::Agent(f) );
}

template<class Tout, class Tin1>
struct LiftTraits<Tout (*)(Tin1)> {
  typedef Tout (*Arg)(Tin1) ;
  typedef Arg Agent ;
  typedef LiftCombinator1<Tout,Tin1,Arg> Result ;
};

template<class Tout, class Tin1, class Tin2>
struct LiftTraits<Tout (*)(Tin1,Tin2)> { ...  }

...
```

5.3.3 Delay Module

To enable recursive and mutually resursive behaviors, *delay* modules are introduced as an unary behavior operator, constructs new behaviors which are one sample delay of old ones. The idea is to store previous value of original behavior as current value of new one:

```
template<class T>
class DelayModule :  public BehaviorModule<T> {
  BehaviorBase<T>* p ;
  T prefetch ;
 public:
  void compute(void) { value = prefetch ; p->get(&prefetch); }
};
```

While stepping forward to next sample, the increasing of frame counter is propagating among the graph structure of behaviors, so not all modules are having the same sample count at same time while in transition. To deal with this temporary discrepancy, asynchronous `get()` mechanism is employed here. Each `BehaviorModule` maintains a list of those modules which are waiting for a value that is currently not available in this module. Asynchronous `get` add modules in the list if necessary, and whenever a new value is generated, the list are updated by completing possible requests. This asychronous `get` is only used in `DelayModule`, since all recursive or mutually recursive behavior definitions must have at least one of them in the loop, or it will never terminate.

Similarly, events are defined as behaviors of `Maybe` type. Since there is no data type in C++, we emulate it by a general `Maybe<T>` class as a wrapper of

T*, plus consideration of effectiveness on copying and assignment, as well as a specialized `Maybe<void>` class as a wrapper of `bool` for `Maybe ()` in Haskell:

```
template<class T>
class Maybe {
 protected:
  T * value ;
  char data[sizeof(T)];
 public:
  bool hasValue(void) const { return value ; }
  T& getValue(void) { return *value ; }
  const T& getValue(void) const { return *value ; }

  Maybe() :  value(0) {}
  Maybe( const T& x ) :  value(new((void*)data) T(x)) {}
  Maybe( const Maybe<T>& x ) :
    value(x.hasValue()?new((void*)data) T(x.getValue()):0) {}
  Maybe<T>& operator = ( const Maybe<T>& x ) { ...  }
   Maybe() { if( value ) value-> T() ; }
};

template<>
class Maybe<void> {
 protected:
  bool value ;
 public:
  ...
};
```

Now the implemetation of events is simple:

```
template<class T> class Event :  public Behavior<Maybe<T> > {...};
```

Thus we can define interactive primitives on behaviors and events, such as `whileE`, `whileByE`, `whenE`, `onceE`, and so on. In fact, `whileByE` is just a `lift` and `whileE` is just a lifted `cast` from `bool` to `Maybe<void>`.

In the original FRP, `Task` are type classes, which have instances of different kinds of tasks. Here we take a simplified implementation, to define task as a single aggregate type of a behavior and an event.

```
template<class T, class Y>
class Task :  public BehaviorCombinator<T> {
 protected:
  BehaviorBase<T>* p ;
  BehaviorBase<Maybe<Y> >* e ;
 public:
  ...
};
```

6 Related Work

Implementations of Functional Reactive Programming (FRP) on Haskell have been pioneered by Elliott[3] and Hudak[9]. Our implementation in C++ share the same stream-based model and some of working mechanism with their work. Recently, Courtney[2] implemented (a sub-set of) FRP in Java, focused on the relationship between the FRP event/behavior model and the Java Beans event/property model.

On the other hand, the idea of dataflow-based (graph) programming exists for a long time. Lucid[1], ECOS[7] and Singal[10] are such languages. In our work, we explore the connection between dataflow and functional programming and build our implemetation upon this connection.

C++ Programming in functional style, including higher-order functions, is already well-known to the community[11] [12]. Our implementation is not directly based on existing works: It is neither our purpose to re-invent a functional programming language over C++, nor to depend to arbitrary syntax notations of third parties.

7 Conclusions

Here we have presented an implementation of FRP in C++. The work demonstrate that the idea of FRP falls beyond functional programming world and it is not only possible but also beneficial of having an C++ implementation of FRP, which faithfully keeps the core feature of the original FRP in Haskell by intelligent exploiture of standard C++ programming language.

Our work has significant potential impact on eliminating the barrier between new emerging functional reactive programming community and massive conventional empirical programming community and providing an alternative understanding of FRP. It also enables "apple to apple" comparative analysis of the two approaches.

7.1 Limitations

Compare with original FRP in Haskell, which itself is evolving, our implementation has certain limitations:

- The inherent limitation of not being able to define a function "on fly" in C++. It can be alleviated, in fact, by introduction of lambda expression of signals, which is not mentioned in this article.

- The implementation of task operations is incomplete. Tasks themself are less functional from programming point of view, so it is still under discussion how much of it shall be useful in our mixed environment.
- This implementation is not thread-safe. If multiple threads is to run the same expression at the same time, explicit mutual exclusive machanism must be employed.
- Memory leak of data graph nodes. It only occurs on the dynamic construction and destruction of the graph itself, not the procession of data, so the it is only a minor problem. Due to potentially recursive nature of behaviors, the leak cannot be resolved simply by reference counting. A garbage collector would solve it completely.

Our implementation is also sharing the same limitation with the Haskell version due to event sampling, that such event as `whileE( timeB == 1.0 )` will never be detected.

7.2 Future Work

Our directions of future include:

- Further exploration of the definition of FRP and its relation with its existing Haskell expression, and the extension of such definition and expression fit into empirical world.
- Full applications of this implementation which utilize its merits and display its possible limitations, as for functional reactive programming.

References

[1] Edward Ashcroft and William Wadge. *Lucid, the Data-Flow Programming Language.* Acadamic Press, 1985.

[2] Antony Courtney. Functional reactive programming in java. In *Proceedings of Third International Symposium on Practical Applications of Declarative Languages (PADL '01)*, March 2001.

[3] Conal Elliott. Functional implementations of continuous modeled animation. *Lecture Notes in Computer Science*, 1490:284–??, 1998.

[4] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, volume 32(8), pages 263–273, June 1997.

[5] The Yale Haskell Group. The Yale FRP user's manual. **http://haskell.cs.yale.edu/frp/manual.html**.

[6] Gregory Hager and Kentaro Toyama. The XVision system: A general-purpose substrate for portable real-time vision applications. *Computer Vision and Image Understanding*, 1998.

[7] J. C. Huang, Jos Muoz, Hal Watt, and George Zvara. ECOS graphs: a dataflow programming language. In *Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 911–918, March 1992.

[8] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[9] Paul Hudak. *The Haskell School of Expression - Learning Functional Programming through Multimedia.* Combridge University Press, Cambridge, UK, 2000.

[10] Richard Kieburtz. Real-time reactive programming for embedded controllers. In *ACM International Conference on Functional Programming (submitted)*‘, Sept. 2001.

[11] Konstantin Läufer. A framework for higher-order functions in C++. In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)*, pages 103–116, June 1995.

[12] Brain McNamara and Yannis Smaragdakis. FC++: Functional programming in C++. In *Proceedings of International Conference on Functional Programming (ICFP)*, Montreal, Canada, September 2000.

[13] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with haskell. In *Practical Aspects of Declarative Languages*, pages 91–105, 1999.

[14] John Peterson, Paul Hudak, Alastair Reid, and Gregory Hager. FVision: A declarative language for visual tracking. In *Proceedings of Third International Symposium on Practical Applications of Declarative Languages (PADL'01)*, March 2001.

[15] Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *International Conference on Software Engineering*, pages 484–493, 1999.

[16] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, Vancouver, British Columbia, Canada, June 2000.