Froid: Functional Reactive Android

James Parker
Department of Computer Science
University of Maryland, College Park
College Park, MD
jp@jamesparker.me

ABSTRACT

In this paper, we present Froid (Functional Reactive Android). Froid is a proof of concept Android library that addresses the problem introduced by callbacks in GUI programming. The library allows developers to create applications using the functional reactive programming model. Specifically, programmers using Froid are able to take advantage of *EventStreams* and *Behaviors* in their programs. *EventStreams* are discrete valued objects that correspond to events, such as button clicks or manipulating text within a text box. On the other hand, *Behaviors* model continuous valued entities, such as the current time. *EventStreams* and *Behaviors* are convenient for developers because functional reactive programs automatically update whenever new events occur. To show Froid in action, we also provide a few demo applications and compare them to traditional applications.

1. INTRODUCTION

A problem with GUI programming is that it can be painful due to its use of callbacks. Callbacks split up control flow and make code harder to read. This forces the programmer to know the event model of the system and remember how every piece of his application interacts. Oftentimes, this can be extremely difficult and leads to confusion. As a result, code is written which introduces bugs affecting behavior and stability. We call this situation *callback hell*. Android application programming is also a kind of GUI programming, and thus it suffers from this problem as well.

One solution to *callback hell* is functional reactive programming (FRP). The main idea behind FRP is that it introduces *EventStreams* and *Behaviors* [3]. These objects are special because their values can change over time. This is convenient for programmers since they can work with these objects abstractly, while computations are automatically updated whenever the value of one of these objects changes. What makes this possible is that each *EventStream* or *Behavior* has a set of dependencies. Each dependency is either an *EventStream* or *Behavior* whose value depends on the original object's value. This approach addresses the problem of *callback hell* because it allows a sequence of callbacks to be composed directly. The resulting control flow of a program becomes more apparent. In the end, the developer is left with code that is more compositional since it is easy to compose a current computation with another subcomputation that refines it in some way.

Functional reactive programming has typically been applied in functional programming languages (notably Haskell and Scheme). For instance, Cooper introduced FrTime as an FRP implementation for Racket, a derivative of Scheme [1]. On the other hand, FRP has seen little use in Java, which is the programming language of Android. Moreover, functional reactive programming has never been used for Android in particular. In this project, we set out to design a FRP library on top of the standard Android APIs and to evaluate its utility by using it to build several simple applications. Through this process, we found that by supplementing Java generics, we could define reusable FRP abstractions to develop some proof-of-concept applications.

This paper begins by describing FRP in greater detail. It then introduces Froid, our FRP library for Android. Next, we go over a few example applications which show the library in action. The following section compares our FRP examples to traditional Android programs. Finally, we draw conclusions and consider future work.

2. FUNCTIONAL REACTIVE PROGRAMMING

The core of functional reactive programming relies on the concepts of *EventStreams* and *Behaviors*. The basic principle is that these two structures represent values over time. *EventStreams* are discretely defined with respect to time and can represent events such as button clicks. Conversely, *Behaviors* are continuously defined and model events like the current time. Whenever the value of an *EventStream* or *Behavior* changes, an event fires and propagates to the

```
EditText textbox;
EventStream<String> eventStream =
    Utilities.text( textbox);
Behavior<String> behavior =
    Utilities.hold( eventStream, "");
```

Listing 1: Android code that initializes an *EventStream* and a *Behavior* to the values of a text box.

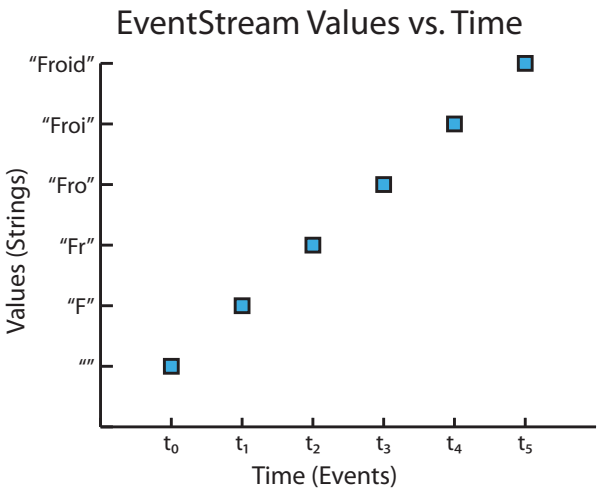

Figure 1: *EventStream* values as a user types "Froid" into a text box.

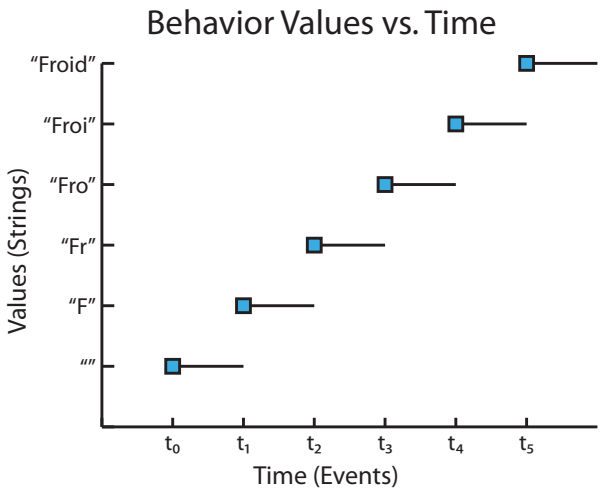

Figure 2: *Behavior* values corresponding to the same EventStream as before.

EventStream's or *Behavior*'s dependencies. Since the event contains the value of the *EventStream* or *Behavior*, the dependencies are able to recompute their values. They then trigger a new event with their updated values, which creates a chain reaction until all subsequent dependencies have been updated.

To illustrate the differences between *EventStreams* and *Behaviors*, we refer to Figures 1 and 2. Each graph displays the values of either an *EventStream* or a *Behavior* over time. The values on the y-axis correspond to string values as a user types the word "Froid" into a text box. In the graphs, each event t_i occurs whenever the text box's value is modified. As the reader can see, the *EventStream* is only defined at the instants when these typing events occur. It is also apparent that the *Behavior* is continuously defined, and it takes on the value associated with the latest event. Listing 1 shows code to initialize such an *EventStream* or *Behavior* using the Froid library. Section 3 will go into more detail about the API, but basically the method *text* creates an *EventStream* out of a text box, while the function *hold* lifts the *EventStream* to a *Behavior* with an initial value of the empty string.

In actual implementations, *Behaviors* cannot truly be continuous over time due to hardware limitations. Therefore, implementations are approximations of true *Behaviors*. This could potentially cause problems because it could change how a program is actually executed. Wan et al. address this issue by showing that if time is discretized enough, implementations of FRP programs satisfy formalized, continuous semantics, as long as *Behaviors* satisfy certain conditions [6].

3. FROID

Froid (Functional Reactive Android) is a proof of concept Android library that allows developers to create applications using the functional reactive programming model. Other applications can make use of Froid in their projects by importing it as an Android library.

```
public class EventStream<T> {
  // Constructor that sets the type of the
  // Behavior's generic.
  public EventStream( Class<T> t);

  // Returns the type of the Behavior's generic.
  public Class<T> getType();

  // Sets value for current event.
  public final void setValue(T v);

  // Add listener l.
  public final void addListener( Listener l);

  // Remove listener l.
  public final void removeListener( Listener l);
}
```

Listing 2: *EventStream* class' method signature.

```
public class Behavior<T> extends EventStream<T> {
  // Constructor that just calls
  // super's constructor.
  public Behavior(Class<T> t);

  // Returns the event corresponding to
  // the behavior's current value.
  public final Event<T> getEvent();

  // Force an update to listeners.
  public final void forceUpdate();
}
```

Listing 3: *Behavior* class' method signature.

```
public final class Utilities {
  // Lifts a function call to a Behavior.
  // All the arguments can optionally be wrapped by
  // a Behavior, but not an EventStream!
  // Throws an IllegalArgumentException if any
  // arguments are EventStreams.
  // 'a -> String ('a -> ... -> 'r) -> ... ->
  //     Behavior 'r
  public static Behavior lift( final Object target,
      String func, final Object... args);

  // Calls a function with static arguments when
  // EventStream i updates. The return value of
  // the function is ignored. Typically used to
  // invoke side effects.
  // EventStream -> a -> (a -> ... -> r) -> ... ->
  //     ()
  public static void trigger( EventStream i,
      final Object target, String func,
      final Object... args);

  // Get the time Behavior.
  // () -> Behavior Time
  public static Behavior<Time> time();

  // Returns an EventStream that subscribes v
  // for click events.
  // Throws IllegalArgumentException if v is null.
  // View -> EventStream View
  public static EventStream<View> clicks( View v);

  // Returns an EventStream that subscribes v for
  // key events.
  // Throws IllegalArgumentException if v is null.
  // View -> EventStream Character
  public static EventStream<Character> keys(
      View v);

  // Returns an EventStream that subscribes v for
  // changes to its text content.
  // Throws IllegalArgumentException if v is null.
  // TextView -> EventStream String
  public static EventStream<String> text(
      TextView v);

  // Returns an EventStream that subscribes s
  // for selections.
  // Throws IllegalArgumentException if s is null.
  // Spinner -> EventStream String
  public static EventStream<String> selects(
      Spinner s);

  /* Combinators */

  // Merges two EventStreams.
  // EventStream 'a -> EventStream 'a ->
  //     EventStream 'a
  static public <T> EventStream<T> merge(
      final EventStream<T> e1,
      final EventStream<T> e2);

  // Lift an object into a constant behavior.
  // 'a -> Behavior 'a
  static public <T> Behavior<T> constant( T o);

  // Acts like Behavior b1 until EventStream e
  // occurs. After that, it acts like Behavior b2.
  // Behavior 'a -> Event 'b -> Behavior 'a ->
  //     Behavior 'a
  static public <T> Behavior<T> until(
      final Behavior<T> b1, final EventStream e,
      final Behavior<T> b2);

  // Snapshot of a behavior when an event occurs.
  // EventStream a -> Behavior b ->
  //     EventStream (a*b)
  static public <T,U> EventStream<Tuple<T,U>>
      snapshot( EventStream<T> e,
      final Behavior<U> b);

  // Filter the events that satisfy the boolean
  // function f.test.
  // EventStream 'a -> Filter 'a -> EventStream 'a
  static public <T> EventStream<T> filter(
      EventStream<T> e, final Filter<T> f);

  // Converts an event stream into a behavior.
  // Initializes the value of the behavior to init.
  // Throws IllegalArgumentException if init
  // is null. (maybe)
  // EventStream 'a -> 'a -> Behavior 'a
  static public <T> Behavior<T> hold(
      EventStream<T> e, T init);

  // Converts a behavior into an event stream.
  // Behavior 'a -> EventStream 'a
  static public <T> EventStream<T> changes(
      Behavior<T> b);
}
```

Listing 4: Method signatures for the *Utility* class.

As an Android library, Froid is implemented in Java, which is imperative rather than functional. The library uses the class *EventStream* to represent *EventStreams*, and the class *Behavior* for *Behaviors*. The *EventStream* and *Behavior* class' method signatures can be found under Listings 2 and 3 respectively. The framework aims to make the creation and manipulation of *EventStream* and *Behavior* objects intuitive and compositional.

It is important to note that in the implementation, *EventStreams* and *Behaviors* are both discrete. For instance, the time *Behavior* should theoretically be continuous, but its value is actually just updated every half second. This makes it reasonable for the *Behavior* class to inherit from the *EventStream* class with a few addition methods. In particular, the *Behavior* class adds a method called *getEvent* that returns an *Event* object containing the current value of a *Behavior*. This is needed because *Behaviors* are defined over all time, while *EventStreams* are not. It is also worth mentioning that the *Event* class is basically a tuple of a value and a timestamp of when the event occurred.

The reader should also notice that constructors for *Behaviors* and *EventStreams* require the types of their values on creation. This allows the API to perform some type checking when the library is loaded. This is also required because Java removes all generic types at compile-time, and some methods in the library use type information at run-time.

Froid handles the propagation of events behind the scenes. Each *EventStream* or *Behavior* maintains a list of its dependencies. Whenever its value is manipulated, a new event is created. This event is then sent to all the dependencies so that subsequent *EventStreams* and *Behaviors* can update.

We now highlight some of the commonly used methods in the library. All of these methods are found in the *Utilities* class and their method signatures are shown in Listing 4. $Lift$ is probable one of the more interesting methods. It takes a regular function and lifts it so that it can take *Behaviors* as parameters and returns a new *Behavior* itself. This is extremely useful because it allows developers to use any previously implemented methods in their FRP programs. Implementing this operation is a little tricky. To find the actual function being lifted, the function name and type signature must be examined to get the appropriate method back from the runtime. Then each of the arguments (and the caller) must be checked to see if they are *Behaviors*. All *Behavior* arguments must add the new lifted *Behavior* as a dependency so that events propagate to the method previously retrieved from the runtime.

Another library function, *time*, simply returns the time *Behavior*, except its type is the Android library's $Time$ class. This allows programmers to make use of built in operations when working with the time. This *Behavior* is also a singleton object because there only needs to be one of them.

Clicks, *text*, and *selects* are all fairly similar in that they are *EventStreams* whose values correspond to user interface items. *Clicks* sends an event when the specified button is pressed. $Text$ updates if the value of a textfield changes. *Selects* corresponds to the string value of a spinner, which is similar to an HTML radio form.

Froid also offers combinators to work with *EventStreams* and *Behaviors*. These functions are powerful in that they enable multiple *EventStreams* and *Behaviors* to join together. The result is an *EventStream* or *Behavior* that is dependent on events propagating from both sources.

An example of this is the $filter$ combinator which tests an *EventStream* against a boolean function defined by the programmer. If the *EventStream* fails the test, $filter$ will prevent that event from propagating, thereby halting any dependent computations.

The *merge* operation is another combinator that provides a way to combine two *EventStreams* of the same type into a single *EventStreams*. It works by making itself a dependent of both *EventStream* parameters, and then it passes along any event it receives.

The *until* operator works a little differently. When first initialized, it takes on the values of the first *Behavior*. Meanwhile, it listens for an event from the second parameter, which is an *EventStream*. Once an event occurs, it stops listening to the *EventStream* and starts behaving as the last *Behavior* parameter.

```
public class ClockActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate( savedInstanceState);

    LinearLayout layout = new LinearLayout(this);
    layout.setOrientation(LinearLayout.VERTICAL);

    TextView label = new TextView(this);
    label.setGravity( Gravity.CENTER_HORIZONTAL);

    Behavior<Time> time = Utilities.time();
    Behavior<String> s =
      Utilities.lift(time, "format", "%I:%M:%S%p");
    Utilities.lift(label, "setText", s);

    layout.addView(label);
    setContentView( layout);
  }
}
```

Listing 5: Clock application using Froid.

4. EVALUATION

Here we evaluate Froid by implementing a few simple programs using the library. We then compare these programs to traditional implementations of these same programs.

4.1 Demo Applications

To demonstrate an application that uses Froid, we present a simple clock application. Its source is displayed in Listing 5 and the program simply shows the current time on the screen. Besides setting up the user interface, the programmer only needs to perform a few steps. First, he requests the time *Behavior* from the library. Then he lifts the built-in function to format the time and receive a string *Behavior*. All that is left to do is display the resulting string by lifting the label's $setText$ method.

In addition to the clock application, we provide two more example applications. One is a program that calculates and displays the radius of a circle over time. Basically, what happens is the radius initially has a constant value of 5.0 until five seconds have passed. After that, the radius oscillates sinusoidally over time. This example is interesting in that it uses a filter to halt further propagation of events. Once the condition that five seconds have passed, events are once again allowed to propagate and the radius starts its oscillation.

The last program is a simple calculator that takes two numeric values from a text box and performs a mathematical operation specified by a spinner. Events are triggered by keystrokes in the text boxes or changes to the selected operation. The output label is then automatically updated based to the computed result that was propagated by the event. For example, if the inputs are initially `6`, `+`, and `3`, and the spinner operator is changed to `*`, this event propa-

```
public class ClockActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate( savedInstanceState);

    LinearLayout layout = new LinearLayout(this);
    layout.setOrientation(LinearLayout.VERTICAL);

    final TextView label = new TextView(this);
    label.setGravity( Gravity.CENTER_HORIZONTAL);

    final Handler h = new Handler();
    Runnable r = new Runnable() {
      public void run() {
        Time t = new Time();
        t.set( System.currentTimeMillis());
        label.setText( t.format( "%I:%M:%S%p"));

        h.postDelayed(this, 500);
      }
    };
    h.post(r);

    layout.addView(label);
    setContentView( layout);
  }
}
```

Listing 6: Standard Android clock application.

gates and the output label automatically changes from `9` to `18`. The code for these last two applications can be found in Appendex A.

4.2 Comparison

To compare FRP programs to traditional Android applications, we implement each example FRP application in normal, imperative Java. We find that the FRP code is more compositional and more intuitive to write. For instance, compare the two clock applications shown in Listings 5 and 6. In the FRP implementation, code is simply chained together based on what it is dependent on. The `time` *Behavior* is initially retrieved from the Froid library. Then the *Behavior* of a string `s` is defined, that updates based on the current value of `time`. Finally, the text of the view, `label`, continuously displays the current value of `s`. This chaining seems simpler and more straightforward than the standard Android version. It definitely achieves composition, which is one of the goals of this work.

To continuously update the clock label with the current time, a *Handler* `h` has to be introduced in the traditional application. The *Handler* `h` then calls *run* in the *Runnable* `r` which gets the current time, formats it, updates the label, and then tells `h` to call itself again in half a second. The FRP version is more straightforward because Froid takes care of all the dependencies on time and completely eliminates the use of callbacks and handlers.

Without Froid, the imperative implementations also tend to accumulate anonymous inner classes. This make code harder to read and can also create scoping issues. If a developer wishes to refer to variables defined outside of the anonymous inner class, he must declare it as `final` which limits his ability to modify that variable later. Looking at the clock example again, this problem is seen where `r` is defined as an anonymous inner class implementing the interface *Runnable* so both of the variables `label` and `h` are forced to be `final`.

Another property of FRP programs is that they can be expressed more concisely. The total number of lines of code for the traditional applications is 295, while the FRP analogs are only 221 lines. This is a 25% reduction in code which seems significant and would allow developers to be more efficient.

The other two applications implemented without Froid are shown in Appendix B. Comparing the two versions of the calculator applications again highlights the benefits of using Froid. For instance, the Froid version only has 71 lines of code while the traditional version has 104 lines of code. In addition, the traditional Android program has three anonymous inner classes, while the FRP implementation has none. This results in cleaner and more compositional code.

5. CONCLUSIONS AND FUTURE WORK

Froid's library serves as a proof of concept system for functional reactive programming in Android. It has even been shown to have benefits over traditional styles of programming by addressing the issue of *callback hell*. That being said, Froid still has a few limitations. One thing to notice is that Froid does incur some overhead. This may hinder application performance as programs grow larger and more complex. Another issue is that Froid cannot always type check objects when the library is loaded. This could lead to run-time errors, especially when objects have more complicated types.

It is left to future work to address these issues, however we propose a few possible solutions. To study the impact of overhead, one could develop a full-fledged Froid application and monitor its performance. It might also prove worthwhile to investigate whether optimizations can be made to the library. To improve the issue of type safety, a more advanced data structure could be developed that keeps track of advanced types. This would still have problems though because it would still be difficult to associate types with method signatures. For instance, assume a *Behavior* has type $Tuple < String, Integer >$. During run-time, these generics are given the general type *Object*. At this point, how does one tell whether the return type of the function $getFirst$ is a *String* rather than an $Integer$? Perhaps a better solution is to build a target language that would compile into Java using Froid. This language could perform static type checking and could even automatically lift primitives as needed.

Many other research projects have studied the viability of functional reactive programming. Elliot and Hudak laid much of the groundwork for FRP when they presented Functional Reactive Animation [3]. Their implementation is one of the most influential, and it describes many of the combinators used in Froid's library. Cooper's thesis presented FrTime, which was a significant contribution as an FRP implementation for Scheme [1]. Courtney's work on Frappe is more related to Froid because it targets Java programs, but

not the Android environment directly [2]. Nicolas Teirlinckx is currently a masters student at the Vrije Universiteit Brussel. He is studying FRP in Android so his thesis should be very related once complete. He also has a blog post that lists many prominent papers in FRP, which has served as a great resource [5]. FRP is not the only way to address *callback hell*. Khoo et al. address this problem in the case of JavaScript by using a generalization of monads to guide control flow [4].

The author would like to recognize his colleagues Nataliya Guts and Kristopher Micinski for their input while working on this project. He would also like to thank his advisors, Dr. Foster and Dr. Hicks, for their guidance during this independent study.

6. REFERENCES

[1] Gregory Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, 2002.

[2] Antony Courtney. Frappe: Functional reactive programming in java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, UK, 2001. Springer-Verlag.

[3] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[4] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. Directing JavaScript with arrows. In *Proceedings of the ACM SIGPLAN Dynamic Languages Symposium (DLS)*, pages 49–58, October 2009.

[5] Nicolas Teirlinckx. Functional reactive programming and further explorations. http://nteirlin.tumblr.com/post/10416917102/thesis-functional-reactive-programming-and-further, 2011.

[6] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. ACM.

APPENDIX

A. APPLICATION CODE WITH FROID

```
public class CircleActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate( savedInstanceState);
    setContentView( R.layout.circle);

    // Find "Finish" button and create click stream.
    Button b =
        (Button) findViewById( R.id.button1);
    EventStream<View> clicks =
        Utilities.clicks( b);
    Utilities.trigger( clicks, this, "finish");

    // Initial constant radius of 5.
    Behavior<Double> r0 = Utilities.constant(
        new Double( 5.0));

    // Compute the radius as a continuous function
    // of time.
    Behavior<Time> time = Utilities.time();
    Behavior ms = Utilities.lift(
        time, "toMillis", true);
    Behavior<Double> rt = Utilities.lift( this,
        "computeRadius", ms);

    // Initialize r as a constant for 5 seconds,
    // and then switch to a function of time.
    EventStream filtered = Utilities.filter( time,
        new Filter<Time>() {
      long startT = System.currentTimeMillis();
      public boolean test( Time t) {
        return ( t.toMillis(true) - startT > 5000);
      }
    });
    Behavior<Double> r = Utilities.until( r0,
        filtered, rt);
    Behavior<String> rString = Utilities.lift( r,
        "toString");

    // Set the text.
    TextView l = (TextView) findViewById(
        R.id.textView1);
    Utilities.lift( l, "setText", rString);

    // Force an update to the computation.
    r0.forceUpdate();
  }

  public Double computeRadius( Long t) {
    return Math.sin( 1000 * t) + 5;
  }
}
```

Listing 7: Code for the circle application.

```
public class CalcActivity extends Activity {
  public void compute( String s1, String op,
      String s2) {
    TextView v = (TextView) findViewById(
        R.id.textView1);

    try {
      float n1 = new Float( s1).floatValue();
      float n2 = new Float( s2).floatValue();
      float r;

      if ( op.equals("+"))
        r = n1 + n2;
      else if ( op.equals("-"))
        r = n1 - n2;
      else if ( op.equals("*"))
        r = n1 * n2;
      else if ( op.equals("/"))
        r = n1 / n2;
      else
        throw new IllegalArgumentException();

      v.setText( ( new Float(r)).toString());
    }
    catch ( Exception e) {
      v.setText( "Undefined");
    }
  }

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView( R.layout.calc);

    // Find "Finish" button and
    // create click stream.
    Button b =
        (Button) findViewById( R.id.button1);
    EventStream<View> clicks =
        Utilities.clicks( b);
    Utilities.trigger( clicks, this, "finish");

    // Create the calculator streams.
    EditText t1 =
        (EditText) findViewById( R.id.editText1);
    EventStream<String> s1 = Utilities.text( t1);

    Spinner s =
        (Spinner) findViewById( R.id.spinner1);
    String[] ops = { "+", "-", "*", "/"};
    ArrayAdapter<String> adapter =
        new ArrayAdapter<String>( this,
        android.R.layout.simple_spinner_item, ops);
    adapter.setDropDownViewResource(
     android.R.layout.simple_spinner_dropdown_item);
    s.setAdapter( adapter);
    EventStream<String> op = Utilities.selects( s);

    EditText t2 = (EditText) findViewById(
        R.id.editText2);
    EventStream<String> s2 = Utilities.text( t2);

    Behavior<String> b1 = Utilities.hold( s1, "");
    Behavior<String> b2 = Utilities.hold(
        op, ops[0]);
    Behavior<String> b3 = Utilities.hold( s2, "");
    Utilities.lift( this, "compute", b1, b2, b3);
  }
}
```

Listing 8: Code for the calculator application.

B. APPLICATION CODE WITHOUT FROID

```
public class CalcActivity extends Activity {
  public void compute( EditText t1, Spinner s,
      EditText t2) {
    String firstArg = t1.getText().toString();
    String op = s.getSelectedItem().toString();
    String secondArg = t2.getText().toString();

    this.compute( firstArg, op, secondArg);
  }

  public void compute( String s1, String op,
      String s2) {
    TextView v = (TextView) findViewById(
        R.id.textView1);

    try {
      float n1 = new Float( s1).floatValue();
      float n2 = new Float( s2).floatValue();
      float r;

      if ( op.equals("+"))
        r = n1 + n2;
      else if ( op.equals("-"))
        r = n1 - n2;
      else if ( op.equals("*"))
        r = n1 * n2;
      else if ( op.equals("/"))
        r = n1 / n2;
      else
        throw new IllegalArgumentException();

      v.setText( ( new Float(r)).toString());
    }
    catch ( Exception e) {
      v.setText( "Undefined");
    }
  }

  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView( R.layout.calc);

    // Find "Finish" button.
    Button b = (Button) findViewById(
        R.id.button1);
    b.setOnClickListener( new OnClickListener() {
      public void onClick(View v) {
        CalcActivity.this.finish();
      }
    });

    // Get the first text.
    final EditText t1 = (EditText) findViewById(
        R.id.editText1);

    // Setup the spinner.
    final Spinner s = (Spinner) findViewById(
        R.id.spinner1);
    String[] ops = { "+", "-", "*", "/"};
    ArrayAdapter<String> adapter =
        new ArrayAdapter<String>( this,
        android.R.layout.simple_spinner_item, ops);
    adapter.setDropDownViewResource(
     android.R.layout.simple_spinner_dropdown_item);
    s.setAdapter( adapter);

    // Get the second text.
    final EditText t2 =
        (EditText) findViewById( R.id.editText2);

    // Set the text watcher.
    TextWatcher watcher = new TextWatcher() {
      public void afterTextChanged( Editable t) {
        CalcActivity.this.compute( t1, s, t2);
      }

      public void beforeTextChanged(
          CharSequence arg0,
          int arg1, int arg2,
          int arg3) {
        // Do nothing.
      }

      public void onTextChanged( CharSequence arg0,
          int arg1, int arg2, int arg3) {
        // Do nothing.
      }
    };
    t1.addTextChangedListener( watcher);
    t2.addTextChangedListener( watcher);

    // Set the spinner listener.
    s.setOnItemSelectedListener(
        new OnItemSelectedListener() {
      public void onItemSelected(
          AdapterView<?> parent, View view,
          int pos, long id) {
        CalcActivity.this.compute( t1, s, t2);
      }

      public void onNothingSelected(
          AdapterView<?> arg0) {
        // Do nothing.
      }
    });
  }
}
```

Listing 9: Code for the calculator application without Froid.

```
public class CircleActivity extends Activity {
  public void onCreate( Bundle savedInstanceState) {
    super.onCreate( savedInstanceState);
    setContentView( R.layout.circle);

    // Find "Finish" button.
    Button b = (Button) findViewById( R.id.button1);
    b.setOnClickListener( new OnClickListener() {
      public void onClick(View v) {
        CircleActivity.this.finish();
      }
    });

    // Create filter.
    final Filter<Time> f = new Filter<Time>() {
      long startT = System.currentTimeMillis();

      public boolean test( Time t) {
        return ( t.toMillis( true) - startT > 5000);
      }
    };

    // Initial constant radius of 5.
    Double rad = new Double( 5.0);

    // Set the text.
    final TextView l = (TextView) findViewById(
        R.id.textView1);
    l.setText( rad.toString());

    // Compute the radius every 500ms.
    final Handler h = new Handler();
    Runnable r = new Runnable() {
      public void run() {
        Time t = new Time();
        t.set( System.currentTimeMillis());

        // Check that the time object
        // passes the filter.
        if ( f.test( t)) {
          long ms = t.toMillis( true);
          Double rad = CircleActivity.this.
              computeRadius( ms);
          l.setText( rad.toString());
        }

        h.postDelayed(this, 500);
      }
    };
    h.post(r);
  }

  public Double computeRadius( Long t) {
    return Math.sin( 1000 * t) + 5;
  }
}
```

Listing 10: Code for the circle application without Froid.