Javascript as an Intermediate Language for FRP

Subtitle Text, if any

Name1
Affiliation1
Email1

1. Introduction

Functional Reactive Programming (FRP) is an exciting and popular way to make interactive applications in a functional style. However, creating production level applications with FRP remains prohibitivly difficult, especially for mobile platforms. Each mobile platform only supports particular languages and archetictures, so many cross-compiling tools must be installed and managed. While many users are able to write FRP applications, the lack of deployment options has limited mobile FRP apps to only the most dedicated FRP researchers.

We propose using Javascript as an intermediate language to eliminate most of the technical hurdles to publishing applications written with FRP, allowing more programmers to use FRP in production. We have developed a tool that automatically creates a mobile app package. It works for any any mobile platform and takes an Elm program as input, though other languages like Haskell could also be supported. The user can develop exclusively in the FRP language, use a forgein function interface (FFI) to interface with lower level languages and hardware, or use FRP for only isolated pieces of the app.

2. Background

Althought deployment is difficult, development of interactive apps can be faster and easier using FRP. One particular strength is that prototypes can be generated very quickly, and these prototypes are easily extented to full applications. This has been used in many interactive multimedia applications such as music, robotics, and games[1]. As an example, see Figure 1 for a very short drawing application written in Elm. To write this in another style would be very verbose and likely difficult.

FRP implementations include many libraries in Haskell, the dedicated FRP language Elm, and the Sodium project which ports FRP to more language like Java and C[]. FRP research is most developed for functional languages like Haskell and Elm, but each mobile platform requires device specific languages (e.g Java for Android). Ideally, FRP programs could be cross-compiled to the target platform without significant interference in the development process.

A GHC cross-compiler exists for Apple's iOS platform that could possibly provide a way to use a Haskell FRP library to develop iOS apps. Perez, in work with Keera Studios, has succesfully cross-compiled Haskell to Android and published the game *Magic Cookies*[]. Although promising, the source code for this remains closed source and the setup has been difficult to reproduce due to many installation steps. Windows phone development remains largely unexplored. These however require that all tool chains be installed and managed seperatly - a large cost for an average user.

3. Results

To simplify the process, we compile FRP programs to an intermediate language, then use a *platform agnostic* tool to compile the intermediate language to any target device desired. The intermediate language presented here is Javascript and platform agnostic tool is Phonegap. For Haskell, ghcjs can compile program to Javascript, while Elm is compiled to Javascript by default. Phonegap allows users to create apps for any platforms by writing Javascript. We instead send the generated (and incomprehensible) Javascript to Phonegap to create a package for any mobile device. Here Javascript is functioning as an intermediate language - the user does not need to look at it for the system to work.

Mobile development also relies on the use of APIs. Many platform specific APIs have similar Javscript implementation, which can be used via the FRP languages FFI. For APIs that require native code, Phonegap provides texitplugins as a foregin function interface (FFI) from javscript to the lower level language. The Javascript plugin code can then be accessed via the FRP langauge through that FFI. Both ghcjs and Elm provide a Javascript FFI to enable this chain.

Performance is a concern when using an intermediate language without optimizations. Preliminary testing shows optimizations are not needed for simple game development using this tool. A common measure for performance in FRP systems is the frames per second (fps) that can be displayed. Any number above 32 fps is generally acceptable, and our proof of concept app maintains $/gt150$ fps on a Nexus 5 running Android 5.1.1. However, this is only for a 2D game, so performance measures for more complex systems will be needed in future work.

Our demo app uses FRP to read the touch interface native to the mobile hardware and control a simple game. The app also uses a plugin FFI to show native ads, mixing Javascript code with Elm. The source code and app files are avaible at `github.com/santolucito/elm_games`.

This approach is unique because it treats the high-level language of one tool as an intermediate language for another. It is unique in its simplicity because it leverages Phonegap to unify the workflow for all platforms - iOS, Android and Windows apps are all created from the same system. There is no need to install a different cross-compiler for each target device. This approach also works well in

```
main =
  Signal.map2 scene Window.dimensions clickLocations

clickLocations =
  Signal.foldp (::) [] Mouse.position

scene (w,h) locs =
  let drawPentagon (x,y) =
          ngon 5 20
            |> filled (hsla (toFloat x) 0.9 0.6 0.7)
            |> move (toFloat x - toFloat w / 2, toFloat h / 2 - toFloat y)
  in
      layers
        [ collage w h (List.map drawPentagon locs) ]
```

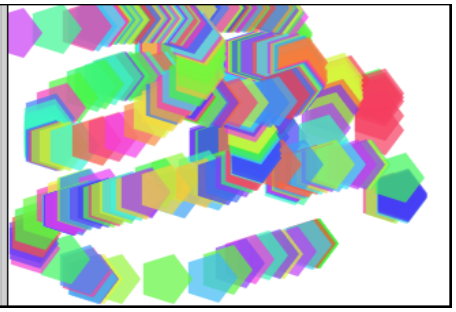

Figure 1. Easy code for a drawing app that traces the mouse (or finger for a touch screen) with colored shapes.

practice, providing good performance and access to low-level APIs when needed.

The development of FRP and functional languages in general relies on adoption by industry. Simplifying the workflow to bring FRP code to market will encourage future work in FRP.

References

[1] P. Hudak. *The Haskell School of Music – from Signal to Symphonies*. Version 2.6, January 2014.