Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

# Functional Reactive Programming

Lecture 24 Tuesday, April 28, 2014

Pure programs (i.e., programs that are free from side-effects such as reading or writing memory or producing input and output) can be easier to reason about than programs with side-effects. This is because when composing two pure programs $e_1$ and $e_2$ together (e.g., in sequence, in parallel, etc.) we don't need to think about how the side effects of $e_1$ and $e_2$ might interact, and change the behavior of the programs.

However, many of the program we write *should* have side effects, because we want these programs to interact in interesting ways with the external environment. For example, we want programs to read user input, receive messages over the network, produce output on a screen, and send messages to servers. Only rarely do we want to write completely pure programs, where the only interesting computation is the final result.

*Functional Reactive Programming* (FRP) is a paradigm that combines pure functional programming approaches with the ability for programs to interact with their external environments. A key concept in FRP is a *signal*, which is a time-varying value. Signals can be used to represent the current value of something in the environment, such as the co-ordinates of a mouse, or the current value in a user input field. An FRP program can compute new signals from existing ones, and the output of an FRP program can be a signal, e.g., the contents of a window, that may contain images and text that change over time.

Conceptually, you can think of an FRP program as being continually recomputed so that as the value of input signals change, the value of the output signals also changes. For example, suppose that $\mathsf{s}$ is an input signal (for example, the current horizontal position of the mouse). The following FRP program computes a signal whose current value is two times the current value of $\mathsf{s}$.

$$\mathsf{lift}\ (\lambda x.\, x + x)\ \mathsf{s}$$

The primitive $\mathsf{lift}$ takes a pure function $(\lambda x.\, x + x)$ and evaluates the function on each value that signal $\mathsf{s}$ takes on. The result of program is itself a signal, i.e., a time-varying value. As time passes and signal $\mathsf{s}$ takes on new values, the output signal also changes over time.

## 1 Elm

To further explore FRP, we will focus on the Elm programming language. Elm is a functional-reactive programming language that specifically targets graphical user interfaces (GUIs). That is, in additional to FRP language constructs, it also has a rich library focused on GUIs. Elm was designed and implemented by Evan Czaplicki '12. You can find out more about Elm at `http://elm-lang.org/`.

The following syntax describes a simple core calculus for Elm. It includes integers $n$, unit $()$, let expressions, and conditionals (where $\mathsf{if}\ e_1\ e_2\ e_3$ evaluates $e_2$ if $e_1$ evaluates to a non-zero integer, and evaluates $e_3$ if $e_1$ evaluates to zero). We assume there is some set of primitive input signals $i$, representing, for example, the current position of the mouse, the current size of the window, etc. In Elm, signals can be thought of as a stream of discrete values. So, for example, if we had a signal representing the current position of the mouse, the signal would in essence be a stream of pairs of integers, with perhaps a new pair being created whenever the position of the mouse changed.

This core calculus for Elm also has primitive operations $\mathsf{liftn}_n$ and $\mathsf{foldp}$, which we describe below.

| | |
|---|---|
| Numbers | $n \in \textsf{\textbf{Int}}$ |
| Variables | $x \in \textbf{Var}$ |
| Input signals | $i \in \textsf{Input}$ |
| Expressions | $e ::= () \mid n \mid x \mid \lambda x{:}\eta.\, e \mid e_1\ e_2$ |
| | $\mid \textsf{if}\ e_1\ e_2\ e_3 \mid \textsf{let}\ x = e_1\ \textsf{in}\ e_2 \mid i$ |
| | $\mid \textsf{lift}_n\ e\ e_1\ ...\ e_n \mid \textsf{foldp}\ e_1\ e_2\ e_3$ |

The primitive operation $\textsf{liftn}_n$ takes a pure function that takes $n$ arguments, and $n$ signals, and "lifts" the pure function to the signals, producing an output signal that varies as any of the input signals vary. (This is actually a family of primitive operations, one operation for each $n \geqslant 1$.) For example, in the following program, assume that $i$ and $j$ are input signals of integers. The expression evaluates to a signal that is the sum of the two input signals.

$$\textsf{lift}_2\ (\lambda a.\, \lambda b.\, a + b)\ ij$$

The lift primitives are where pure functions interact with signals. They allow signals to be transformed (e.g., $\textsf{lift}_1\ (\lambda x.\, x + x)\ i$) and combined (as in the example immediately above).

Elm is pure, in that the only side effects that Elm programs have are via signals to and from the external environment. However, it is useful for a program to have state. Elm provides the primitive foldp, which performs a "fold from the past" on the values of a signal. The name is by analogy with foldl and foldr which perform folds over a list from the left and right respectively. Expression foldp $e_1\ e_2\ e_3$ expects $e_3$ to be a signal, and $e_2$ to be the initial value of the accumulator. Expression $e_1$ is expected to be a function $\tau \to \tau' \to \tau'$, where the signal produces values of type $\tau$, and the accumulator is of type $\tau'$. The foldp expression evaluates to a signal that produces values of type $\tau'$, i.e., it is the sequence of values of the accumulator.

For example, the following program counts the number of times that primitive signal $i$ is equal to zero. (For example, if $i$ is the horizontal mouse position, it counts the number of times the mouse is at the left edge of the window.) Note that the initial value of the accumulator is 0, and each time the value of $i$ changes, if the new value is zero, then the accumulator $c$ is incremented.

$$\textsf{foldp}\ (\lambda x.\, \lambda c.\, \textsf{if}\ x\ c\ (c + 1))\ 0\ i$$

### 1.1 Type system

Consider the following program, where $i$, $j$, and $k$ are signals of integer values.

$$\textsf{let}\ s = \textsf{lift}\ (\lambda x.\, \textsf{if}\ x\ j\ k)\ i\ \textsf{in}\ \ldots$$

The value $s$ is a signal whose values are themselves signals (of integers). This is not a problem in and of itself, but can cause problems in some circumstances. For example, suppose that instead of signal $k$ above, we used a more complex signal expression, perhaps one that depended, via foldp on the entire history of some input signal width (e.g., the width of the window.

$$\textsf{let}\ s = \textsf{lift}\ (\lambda x.\, \textsf{if}\ x\ j\ (\textsf{foldp}\ (\lambda w, a.\, a + w)\ 0\ \textsf{width}))\ i\ \textsf{in}\ \ldots$$

Suppose we run this program for 10 minutes, and then signal $i$ becomes zero for the first time. At that time, signal $s$ should produce the signal foldp $(\lambda w, a.\, a + w)\ 0$ width. But what is the current value of this signal? It should be the sum of the window widths throughout the entire execution of the program up to this point. To compute this value, we would have had to save all of the history of the signal width, even thought we don't know whether signal foldp $(\lambda w, a.\, a + w)\ 0$ width will be used. Alternatively, we could compute the current value of the signal just using the current value of width (i.e., ignoring the history). But this would allow the possibility of having two identically defined signals that have different values, based on when they were created. We avoid these issues by ruling out signals of signals.

Elm's type system is stratified, separating "simple types" and "signal types". Signal types include signals over simple types, and functions that produce signal types. Note that the types do not allow signals of signals, nor functions that consume signals and produce values of primitive types. Elm's type system also ensures that signals and values are used correctly.

| | |
|---|---|
| Simple types | $\tau ::= \textbf{unit} \mid \textbf{int} \mid \tau \longrightarrow \tau'$ |
| Signal types | $\sigma ::= \textbf{signal}\ \tau \mid \tau \longrightarrow \sigma \mid \sigma \longrightarrow \sigma'$ |
| Types | $\eta ::= \tau \mid \sigma$ |

$$\textsc{T-Unit}\ \frac{}{\Gamma \vdash ()\!:\!\textbf{unit}} \qquad \textsc{T-Number}\ \frac{}{\Gamma \vdash n\!:\!\textbf{int}} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \eta}{\Gamma \vdash x\!:\!\eta}$$

$$\textsc{T-Input}\ \frac{\Gamma(i) = \textbf{signal}\ \tau}{\Gamma \vdash i\!:\!\textbf{signal}\ \tau} \qquad \textsc{T-Lam}\ \frac{\Gamma, x\!:\!\eta \vdash e\!:\!\eta'}{\Gamma \vdash \lambda x\!:\!\eta.\, e\!:\!\eta \to \eta'} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1\!:\!\eta \to \eta' \quad \Gamma \vdash e_2\!:\!\eta}{\Gamma \vdash e_1\ e_2\!:\!\eta'}$$

$$\textsc{T-Cond}\ \frac{\Gamma \vdash e_1\!:\!\textbf{int} \quad \Gamma \vdash e_2\!:\!\eta \quad \Gamma \vdash e_3\!:\!\eta}{\Gamma \vdash \textsf{if}\ e_1\ e_2\ e_3\!:\!\eta} \qquad \textsc{T-Let}\ \frac{\Gamma \vdash e_1\!:\!\eta \quad \Gamma, x : \eta \vdash e_2\!:\!\eta'}{\Gamma \vdash \textsf{let}\ x = e_1\ \textsf{in}\ e_2\!:\!\eta'}$$

$$\textsc{T-Lift}\ \frac{\Gamma \vdash e\!:\!\tau_1 \to \cdots \to \tau_n \to \tau \qquad \Gamma \vdash e_i\!:\!\textbf{signal}\ \tau_i \ \ \forall i \in 1..n}{\Gamma \vdash \textsf{lift}_n\ e\ e_1\ \dots\ e_n\!:\!\textbf{signal}\ \tau}$$

$$\textsc{T-Fold}\ \frac{\Gamma \vdash e_f\!:\!\tau \to \tau' \to \tau' \qquad \Gamma \vdash e_b\!:\!\tau' \qquad \Gamma \vdash e_s\!:\!\textbf{signal}\ \tau}{\Gamma \vdash \textsf{foldp}\ e_f\ e_b\ e_s\!:\!\textbf{signal}\ \tau'}$$

### 1.2 Operational semantics

The operational semantics of Elm are actually in two phases. In the first phase, execution of the program occurs according to standard call-by-value semantics, without any evaluation of signals. This first phase of execution essentially defines a "signal graph", that is the plumbing that describes how output signals are computed by computation over the values of input signals. In the second phase of execution, as the values of the input signals vary, the computation is performed on the new input values to produce the output values, i.e., the value of the output signals. Essentially, the first phase of computation allows us to perform computation that is independent of the values of signals, and put the program in a form where it will be possible to efficiently compute new output values as the input values change.

We define an intermediate language such that a term in the intermediate language is the result of evaluating a source language program. The grammar below defines the syntax of the intermediate language.

| | |
|---|---|
| Values | $v ::= () \mid n \mid \lambda x\!:\!\tau.\, e$ |
| Signal terms | $s ::= x \mid \textsf{let}\ x = s\ \textsf{in}\ u \mid i \mid \textsf{lift}_n\ v\ s_1\ \dots\ s_n \mid \textsf{foldp}\ v_1\ v_2\ s$ |
| Final terms | $u ::= v \mid s$ |

We define an evaluation-context-based call-by-value semantics to reduce Elm core-language expressions to final terms.

$$\begin{aligned} E ::= {} & [\cdot] \mid E\ e \mid \textsf{if}\ E\ e_2\ e_3 \mid \textsf{let}\ x = E\ \textsf{in}\ e \mid \textsf{let}\ x = s\ \textsf{in}\ E \mid \\ & \textsf{lift}_n\ E\ e_1\ \dots\ e_n \mid \textsf{lift}_n\ v\ s_1\ \dots\ E\ \dots\ e_n \mid \\ & \textsf{foldp}\ E\ e_2\ e_3 \mid \textsf{foldp}\ v_1\ E\ e_3 \mid \textsf{foldp}\ v_1\ v_2\ E \\ F ::= {} & [\cdot]\ e \mid \textsf{if}\ [\cdot]\ e_2\ e_3 \mid \textsf{let}\ x = [\cdot]\ \textsf{in}\ e \mid \textsf{lift}_n\ [\cdot]\ e_1\ \dots\ e_n \mid \textsf{foldp}\ [\cdot]\ e_2\ e_3 \mid \textsf{foldp}\ v_1\ [\cdot]\ e_3 \end{aligned}$$

$$\textsc{Context}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \textsc{Cond-True}\ \frac{v \neq 0}{\mathsf{if}\ v\ e_2\ e_3 \longrightarrow e_2} \qquad \textsc{Cond-False}\ \frac{}{\mathsf{if}\ 0\ e_2\ e_3 \longrightarrow e_3}$$

$$\textsc{App}\ \frac{}{(\lambda x\!:\!\eta.\, e_1)\ e_2 \longrightarrow \mathsf{let}\ x = e_2\ \mathsf{in}\ e_1} \qquad \textsc{Reduce}\ \frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \longrightarrow e[v/x]}$$

$$\textsc{Expand}\ \frac{x \notin fv(F[\cdot])}{F[\mathsf{let}\ x = s\ \mathsf{in}\ u] \longrightarrow \mathsf{let}\ x = s\ \mathsf{in}\ F[u]}$$

The semantics has some interesting features. First, application $(\lambda x:\eta.\,e_1)\ e_2$reduces to a let expression $\mathsf{let}\ x = e_2\ \mathsf{in}\ e_1$ (Rule APP), and then expression $e_2$ is evaluated (using context $\mathsf{let}\ x = E\ \mathsf{in}\ e$. If $e_2$ evaluates to a non-signal value, then substitution is performed (Rule REDUCE). Otherwise, if $e_2$ evaluates to a signal value, then evaluation continues on expression $e_1$, without performing substitution (due to context $\mathsf{let}\ x = s\ \mathsf{in}\ E$).

Rule EXPAND expands the scope of a let expression to allow evaluation to continue. Contexts $F$ describe where Rule EXPAND can be applied, and includes all contexts where a simple value $v$ is required. For a context $F$, we write $fv(F[\cdot])$ for the set of free variables in $F$. We assume that expressions are equivalent up to renaming of bound variables, and an appropriate variable $x$ can always be chosen such that $x \notin fv(F[\cdot])$.

Let's see an example of the first phase evaluation. Suppose that $i$ is an input signal of integers.

$$\begin{aligned}
& \mathsf{let}\ \mathsf{double} = \lambda f.\,\lambda x.\,f\ (f\ x)\ \mathsf{in} \\
& \mathsf{let}\ t = \mathsf{lift}\ (\mathsf{double}\ (\lambda y.\,y+1))\ i\ \mathsf{in} \\
& \mathsf{lift}_2\ (\lambda a.\,\lambda b.\,a+b)\ i\ t \\
\longrightarrow\ & \mathsf{let}\ t = \mathsf{lift}\ ((\lambda f.\,\lambda x.\,f\ (f\ x))\ (\lambda y.\,y+1))\ i\ \mathsf{in} \\
& \mathsf{lift}_2\ (\lambda a.\,\lambda b.\,a+b)\ i\ t \\
\longrightarrow\ & \mathsf{let}\ t = \mathsf{lift}\ (\lambda x.\,(\lambda y.\,y+1)\ ((\lambda y.\,y+1)\ x))\ i\ \mathsf{in} \\
& \mathsf{lift}_2\ (\lambda a.\,\lambda b.\,a+b)\ i\ t
\end{aligned}$$

This last expression is a final term. No more evaluation can be performed without having some values from signals. Conceptually, this final term can be thought of as a graph that indicates which signals are consumed by which computation. The graph below shows that there are 3 signals in the program. One is the input signal $i$. The signal $t$ (defined as $\mathsf{lift}\ ((\lambda f.\,\lambda x.\,f\ (f\ x))\ (\lambda y.\,y+1))\ i$) uses the input signal $i$. The program evaluates to the signal $\mathsf{lift}_2\ (\lambda a.\,\lambda b.\,a+b)\ i\ t$, which uses both signal $i$ and signal $t$.

$$\begin{tikzcd}
\boxed{i} \arrow[rd] \arrow[dd] & \\
& \boxed{t = \mathsf{lift}\ (\lambda x.\,(\lambda y.\,y+1)\ ((\lambda y.\,y+1)\ x))\ i} \arrow[ld] \\
\boxed{\mathsf{lift}_2\ (\lambda a.\,\lambda b.\,a+b)\ i\ t} &
\end{tikzcd}$$

For the second phase evaluation, the final term of the first phase is repeatedly evaluated as new values of input signals arrive. We won't dig into the semantics here, but you can't find a semantics in the paper "Asynchronous Functional Reactive Programming for GUIs."[1] In the paper, the semantics for the second phase evaluation is given by a translation to a CML program, since in that paper we are concerned with a concurrent semantics for signal evaluation.

[1]E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

## 2 Continuous FRP

The Elm language assumes that signals are discrete. That is, the value produced by a signal changes discretely. For some signals, such as mouse clicks, key presses, etc., this is clearly the right choice, as the signal represents discrete events in the external environment. Indeed, any practical FRP language needs to be able to represent discrete signals. But some signals are naturally continuous. For example, the current position of the mouse could be seen as a continuous signal, as could other signals such as time, or the position of a glyph on a canvas.

Clearly, even in a language with continuous signals, in implementation such signals will be approximated by discrete values. However, continuous signals can have an elegant semantics that are amenable to "rendering" them at different resolution. That is, based on the computing power of a particular machine on which you are running an FRP, the machine may be able to sample an external signal (e.g., the mouse position, a sensor reading) at higher or lower frequency. In a FRP language with continuous signals, a higher sampling frequency means that the implementation is more closely approximating the "ideal" semantics, analogous to how a higher-resolution monitor may render a vector graphic more accurately than a lower-resolution monitor. By contrast, in a FRP language with discrete signals, the frequency of sampling may impact the meaning of the program. For example, if we are performing a foldp on the horizontal mouse position, summing up the values, then sampling the mouse position with higher frequency will result in larger sums. (Elm provides a primitive signal that will attempt to sample at a specified frequency, but will adjust based on the machine's capabilities.)

Something for you to think about: what is the equivalent of foldp for a continuous signal?