Seif Haridi

Assignment 4, CS2104: RC-circuits

Due date: 2002-10-27
Points: 8 from 100 exam points

This lab assignment uses streams as a model for simple electrical circuits. It exercises lazy evaluation, streams, and higher-order programming techniques.

Streams as Signals

We use streams to directly model signal-processing systems by representing the values of a signal at successive time intervals as consecutive elements of a stream.

For instance, we can implement an integrator or summer that, for an input stream $x = (x_i)$, an initial value C, and a small increment dt, accumulates the sum

$$S_i = C + \sum_{j=1}^{i} x_j \, dt$$

and returns the stream of values $S = (S_i)$.

Figure 1 shows a model of a signal-processing system that corresponds to the integral function. The input stream is scaled by dt and passed through an adder, the output of which is passed back through the same adder.

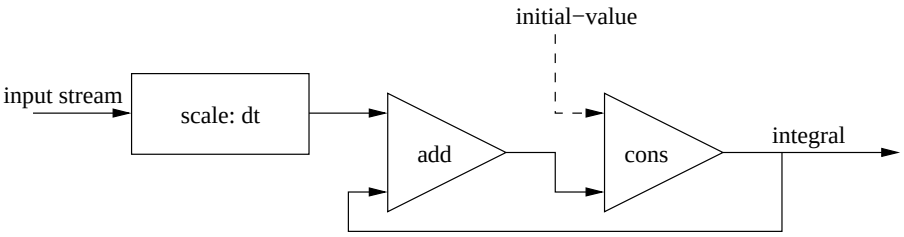

Figure 1: The integral procedure viewed as a signal-processing system.

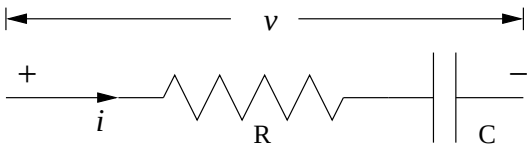

Figure 2: An RC-circuit.

Modeling Electrical Circuits

We can model electrical circuits using streams to represent the values of currents or voltages at a sequence of times. For instance, suppose we have an RC circuit consisting of a resistor of resistance R and a capacitor of capacitance C in series (see Figure 2). The voltage response v of the circuit to an injected current i is determined by the following equation

$$v = v_0 + \frac{1}{C}\int_0^t i(t)\,dt + Ri$$

where v_0 is the initial capacitor voltage (see the signal-flow diagram in Figure 3).

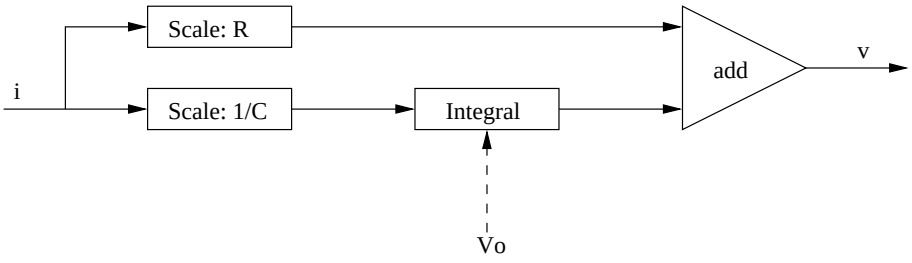

Figure 3: A signal-flow diagram for the RC-circuit.

Assignment

To implement the simulation, you need to develop six functions and store them in the file `lab4.oz`. In the following, we use conventions: `S`, `S1`, `S2` refer to streams, `F` to a function, and `SF`, `SF1`, `SF2` to streams of floating-point numbers (*floats*).

Mapping Streams

`{StreamMap S F}` lazily applies `F` to all elements of `S` returning a new stream.
For example,
`{StreamMap 1|2|3|4|··· MultiplyWithFive}`
returns `5|10|15|20|···`.

Tip. In order to request the first `N` elements of a stream `S`, you can execute `{Nth S N _}`. Explain what this does!

Zipping Streams

`{StreamZip S1 S2 F}` lazily zips two streams `S1` and `S2` by applying `F` to elements at the same position in `S1` and `S2` (the name *zip* is inspired by the fact that the operation joins elements of two streams like a zipper on trousers does).

For example,

```
local
   fun {Multiply X Y} X*Y end
in
   {StreamZip 1|2|3|··· 4|5|6|··· Multiply}
end
```

returns `4|10|18|···`.

Note. The following functions capture the functionality from the signal-flow diagram in Figure 3. Use mapping, folding, and zipping for streams as defined above.

Scaling Streams

`{StreamScale SF Factor}` lazily scales the stream elements by multiplying them with the factor (a float) `Factor`.

For example,

```
{StreamScale 1.0|2.0|3.0|4.0|··· 2.0}
```

returns the stream `2.0|4.0|6.0|8.0|···`.

Integrating Streams

`{StreamIntegrate SF InitValue Dt}` lazily returns a stream obtained by integrating the values of the float-stream `SF` using the initial float-value `InitValue` and the time increment float-value `Dt`. How integration is performed can be seen in Figure 1.

For example,

```
{StreamIntegrate 1.0|1.0|1.0|··· 5.0 1.0}
```

returns `5.0|6.0|7.0|···`.

Adding Streams

`{StreamAdd SF1 SF2}` lazily computes a float stream obtained by element-wise addition of the elements of `SF1` and `SF2`.

For example,

```
{StreamAdd 1.0|1.0|1.0|··· 2.0|3.0|4.0|···}
```

returns `3.0|4.0|5.0|···`.

Simulating RC-Circuits

`{MakeRC R C Dt}` takes resistance `R`, capacitance `C`, and time-step `Dt` as arguments (all are floats). It returns a function `RC`. The function `{RC SF V0}` takes the float-stream `SF` representing injected electrical current and the initial capacitor voltage `V0` and returns a stream of output voltages.

For example,

```
declare
fun lazy {MakeOnes} 1.0|{MakeOnes} end
RC = {MakeRC 5.0 1.0 0.2}
Vs = {RC {MakeOnes} 2.0}
{Nth Vs 5 _}
{Browse Vs}
```

shows `7.0|7.2|7.4|7.6|7.8|_<Future>` in the Browser.

A Simulator-Graph for Electrical RC-Circuits

To ease testing and understanding, we provide a Simulator-Graph (`SG`) for electrical RC-circuits.

You have to construct a *module* that contains the implementation of the abstract data type. A module is a record where the features give the interface names and the fields are the implementations (functions in our case). Construct the module the following to your file `Lab4.oz`:

```
Streams = streams(streamMap:       StreamMap
                  streamZip:       StreamZip
                  streamScale:     StreamScale
                  streamIntegrate: StreamIntegrate
                  streamAdd:       StreamAdd
                  makeRC:          MakeRC)
```

To use `SG`, download the file `SimGraph.oz` from the course webpage and copy into your current working directory. Compile the file by typing the following command in a DOS window situated at your current working directory:

```
ozc -c SimGraph.oz
```

This command will create the compiled file `SimGraph.ozf` in the current directory. Load the procedure `Test` as follows:

```
Test={Module.link [´SimGraph.ozf´]}.1.make
```

Call the procedure `Test` with your module `Streams` as argument:

```
{Test Streams}
```

If you do not get any error messages a window pops up. From the window you can choose the type of current-function (i.e. `el-current`) for the simulation. You can step trough the simulation by pressing the buttons `step` (i.e. one time-step) or `tenSteps` (i.e. ten time-steps). The Browser displays the expected values

(i.e. the values that you should get) and the values that are calculated with your RC-simulator module. If they match you are done, otherwise you have to work some more...

Acknowledgments

The idea and the initial problem formulation is taken from: Harold Abelson and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, second edition. The MIT Press, Boston, MA, USA, 1996.