Seminar Paper

Functional Reactive Programming with Elm

Jan-Patrick Baye

WS 2014/2015

Advisor: Sandra Dylus
## Contents

1. Introduction ................................................. 1
   1.1. Motivation ............................................ 1

2. Functional Reactive Programming .......................... 3

3. Elm .................................................................. 5
   3.1. Introduction ............................................. 5
   3.2. Syntax ...................................................... 6
   3.3. Signals ..................................................... 8
      3.3.1. Circle at constant position ....................... 8
      3.3.2. Circle following the mouse ....................... 9
      3.3.3. Circle spinning around the corner ............... 9
      3.3.4. Circle spinning around the mouse ............... 10
   3.4. Concurrency ............................................. 11
      3.4.1. Signal handling .................................... 11
      3.4.2. Async ............................................... 12

4. Conclusion ................................................... 15
   4.1. FRP ....................................................... 15
   4.2. Elm ...................................................... 15

A. Source Code ................................................ 17
1. Introduction

In this paper we will take a look at Functional Reactive Programming (FRP) and its implementation in Elm. Elm is a declarative programming language designed for FRP, especially GUI programming, which compiles to JavaScript.

In a lot of domains reactivity is a core requirement. Such domains include audio processing, animation, robotics and of course graphical user interfaces. In such domains FRP has already shown some potential [San]. Since Elm is specifically designed for graphical user interface programming we will primarily focus on the GUI side of FRP. In addition we will look at Concurrent FRP, which is the second major feature of Elm. Concurrent Functional Reactive Programming tries to maximize the responsiveness of programs.

As a declarative approach to GUI design, FRP and Elm give us the common benefit of declarative programming: a high level of abstraction. This way we can focus on what we want our program to do and not on how to do this. This level of abstraction is an essential difference to other GUI frameworks. Most common graphical user interface libraries and frameworks are of imperative nature, such as Java’s SWING or C++’s QT. We will see that FRP gives us a simpler way of implementing GUIs than these frameworks.

After a short motivation for Functional Reactive Programming in the next section, we will take a look at the concept of FRP. After which we will dive into Elm and show how these concepts are transferred. To demonstrate these concepts in Elm we will take a look at some small programs to demonstrate the syntax and core features of Elm.

1.1. Motivation

The behavior and input of programs can be described as a function of time. For every point in time during the program execution the mouse is at a specific position, although we can only determine the current mouse position directly. For the past we would have to save the mouse movement and for future points in time wait, until we arrive there. Nevertheless functional reactive programming uses this modelling technique. In chapter 3 we will see how this enables us to write a responsive GUI program. Figure 1.1 shows a series of screenshots of our final program. A red circle spinning around the mouse. We will develop this program step by step and see how FRP enables us to focus on small subproblems to solve the big one. The two subproblems we will solve is a circle following the mouse and a circle spinning around a fixed point and then combine the two solutions to create our final program. As a confirmation of FRPs expressiveness, we will see that the described program is just about 35 lines of code long.
1. Introduction

Figure 1.1.: ball spinning around mouse
2. Functional Reactive Programming

The concept behind FRP was first introduced in a paper called *Functional Reactive Animation* (Fran) [EH97] in 1997. Fran is a collection of data types and function for constructing interactive animations. The authors introduced two types of values: Behaviors and Events. The idea behind these two types is to model the program and especially the user input as functions of time.

**Behavior**
Behaviors represent continuous, time-varying values. They are modelled as a function from time to a value.

$$\text{Behavior } \alpha = \text{Time} \rightarrow \alpha$$

The mouse position is a good example for a value, which can be represented as a signal.

**Event**
Events happen at discrete points in time and as such are represented by a list of time-value-pairs. Each pair is one concrete event.

$$\text{Event } \alpha = [(\text{Time}, \alpha)]$$

The timestamps in this list must increase monotonically. Examples for events are mouse clicks and key presses.

In 2001 Paul Hudak introduced *Real-time FRP*. As FRP programs can rely on past values, space leaks are a possibility, as it may be necessary to remember all past values of an event or behavior. As the implementation of Fran was done in Haskell, programs also had the possibility of time-leaks. Behaviors may be inspected infrequently in FRP. In these cases Haskell’s lazy evaluation may result in either space-leaks (the build-up computation needs more and more memory) or time-leaks as the evaluation may cause a significant delay. Real-time FRP aims to solve these problems. To do these a isomorphism between behaviors and events was introduced:

$$\text{Event } \alpha \cong \text{Behavior } (\text{Maybe } \alpha)$$

If at time $t$ the event has value $v$, the corresponding behavior has value $\text{Just } v$, otherwise it holds $\text{Nothing}$. Consequently, we can now model *Events* and *Behaviors* as a single type. As a result a new datatype *Signal* represents both behaviors and events in real-time FRP.

$$\text{Signal } \alpha = \text{Time} \rightarrow \alpha$$
Real-time FRP then defines an unrestricted base language and a limited reactive language for signal manipulation. These changes resolve the problems of Fran, but also the restrictions of the reactive language part, made real-time FRP less expressive. In 2002 Event-Driven FRP was proposed. It is a derivation of real-time FRP and introduces discrete Signals. These are signals that only change at the occurrence of events. As such Event-Driven FRP only further restricts real-time FRP.

Another variation of FRP was also formulated in 2002: Arrowized FRP. Just like real-time and event-driven FRP, Arrowized FRP uses signals. But to resolve the space and time leaks of Fran, the programmer has no direct access to signals and can only use them via signal functions.

$$SF \alpha \beta = \text{Signal } \alpha \rightarrow \text{Signal } \beta$$

Although Arrowized FRP regains FRPs level of expressiveness [Cza12], it still has one problem: continuous signals. To faithfully implement the semantics of signals, the update interval of all signals has to be zero. Meaning that the resulting computation of a changing signal has to be done instantaneously.

These problems with continuous signals are the reasons, why Elm uses discrete signals in its implementation. (See subsection 3.4.1.) Even though Elm follows Event-Driven FRP with this design choice, its author claims, that Arrowized FRP could be fully embedded in Elm [Cza12], making it more expressive than real-time FRP and Event-Driven FRP. Furthermore, Elm introduces concurrency into functional reactive programming.
3. Elm

In this chapter we will examine Elm and its implementation of functional reactive programming. After a quick look at what Elm does, we will ease into Elm programming with some small functions needed for our program, but yet unrelated to signals. Thereafter we will work with some basic signals to create the afore-mentioned (Figure 1.1) program: the around the mouse spinning circle. At last we address Elm’s notion of concurrency.

3.1. Introduction

Elm is a young programming language released in April 2012 that is written in Haskell and compiles to HTML and JavaScript. Accordingly, the compiled "executable" is independent of specific operating systems, as it only needs a current browser to run.

Elm combines the basics of a functional language with the concept of signals and provides primitive functions to combine signals. As such Elm features the basic data types one can expect from functional languages, such as tuples, lists and records. As graphical user interface programming is the main aspect of Elm, it provides a series of functions and types for graphics programming as part of its core language. The strong focus on graphical programming and the fact that it compiles to HTML and JavaScript is also apparent in the type of an Elm-program’s main function. More exactly types, since the main function can have one of four: Element, Html, Signal Html, Signal Element. 

Element is a graphical building block, which represents a rectangle of known width and height. These Elements can be combined to new ones and thus form a complex layout. Html is an abstract representation of HTML documents. As all input in Elm is represented by Signals, Html as well as Element represent constant programs without any user inputs. If some kind of input is used Signal Element and Signal Html are the resulting main function’s possible types.

Listing 3.1 shows a small Elm program displaying the number 42. asText converts its argument to a textual representation and then embeds this representation into an Element. import Text (..) imports the Text module’s functions into the main program’s namespace. asText is one of those functions.

```
import Text (..)

main = asText 42
```

Listing 3.1: text.elm
3. Syntax

Although the basic syntax is similar to Haskell’s and many basics like let- and lambda-expressions are syntactically the same, there are a number of differences in the details. We will, however, just cover a few basics that are relevant to the following examples. The discovery of the rest is left to the interested reader.

Firstly the meaning of \( :: \) and \(:\) are swapped in comparison with Haskell. Meaning \( :: \) is the list constructor and \(:\) is used for type annotations, respectively. Listing 3.2 shows a list of integers \( a \) with elements one through four. Furthermore, although \( // \) can be used to construct lists as seen in the listing. It cannot be used for type annotations. \( List \alpha \) is used for those.

Listing 3.2: lists and type annotations

\[
a : List Int \\
a = 1 :: 2 :: [3, 4]
\]

As does Haskell (\( \$ \)), Elm provides a special function application operator, more precisely two. Namely \( <| \) and \( |> \), which are defined as shown in Listing 3.3. Hence \( <| \) has the same meaning as Haskell’s \( \$ \) operator.

Listing 3.3: function application operators

\[
f <| x = f x \\
x |> f = f x
\]

Now, before we actually dive into signals in Elm, we have to take a look at some small helper functions to be able to draw a circle at a specific position on the screen. This also gives us an opportunity to further familiarize ourselves with Elm’s syntax. Required imports are ignored in all listings. The complete final program with imports can be seen in Listing A.1.

In order to start things of let us study the code in Listing 3.4. \( Form \) is a data type representing any kind of geometric form. Besides other information, \( Forms \) contain their position relative to the center of the canvas they are drawn upon. Since the mouse position is measured from the top left corner of the window and we want to use the whole window as our canvas, \( drawAt \)'s task is to translate these coordinates into the forms relative position coordinates and update the object accordingly. To do this translation, \( drawAt \) takes three parameters: two pairs of integers, the first one denoting the dimensions of the window we want to draw on and the other one specifying the position we want to draw at. This position is relative to the top left corner of the window with x- and y-axis pointing right and down, respectively. The third parameter is the \( Form \) which should be drawn.

Listing 3.4: drawAt and toFloatT

\[
drawAt : (Int, Int) -> (Int, Int) -> Form -> Form \\
drawAt (dx, dy) (x, y) f = \\
  let \\
  pos = toFloatT (x - dx // 2, dy // 2 - y) \\
in
\]
Elm provides us with the `move` operation on `Forms`. `move` updates the position of its second argument about the amount given by the first parameter. The `move` function expects as first argument a tuple of floating point values. Mouse position and window dimensions are returned as integer values by the Elm library. This type mismatch is the reason we need `toFloatT`.

**Listing 3.5: canvas**

```haskell
type alias Canvas = (List Form -> Element)

canvas : (Int. Int) -> Canvas
canvas (a. b) = collage a b
```

Since Elm only displays `Elements`, all `Forms` somehow have to be converted to this type. This can be done via `collage`. `collage`’s three parameters are its dimensions and a list of `Forms`. The return value of `collage` is an `Element` of the specified size containing the given `Forms`. Listing 3.5 shows an abstraction of `collage` which we name a `Canvas`. It can be constructed with `canvas`, which takes a tuple specifying its dimensions and using these as the first two arguments for `collage`.

The last function we want to define in this section is `circleAt` (Listing 3.6). As the name suggests the function purpose is to draw a circle at a specified position. In order to do this, it requires three information: a `Canvas` to draw upon, the `Canvas`’ dimensions and the position of the circle. `circleAt` then creates a `Form` (line 4), which represents a filled red circle with a radius of 15. `filled`, `red` and `circle` are functions provided by Elm to create basic `Forms`. Lastly `circleAt` uses the `Canvas` and `drawAt` to return an `Element` containing the circle as its only `Form`.

**Listing 3.6: circleAt**

```haskell
circleAt : Canvas -> (Int. Int) -> (Int. Int) -> Element

circleAt f dim pos = 1
  let 2
    c = filled red (circle 15) 3
    in 4
    f [drawAt dim pos c] 5
```

As mentioned before Elm provides us with a function application operator. Thus we can rewrite line 4 of Listing 3.6 using the `<|` operator as shown in Listing 3.7.

**Listing 3.7: using the function application operator**

```haskell
c = filled red <| circle 15
```
3. Signals

Now it is time to take a look at some signals. Elm gives us a range of signals, three of which we will use, and some functions to combine and manipulate signals. One of the most basic functions to manipulate signals is `map`. `map`'s type signature is shown in Listing 3.8. Map applies the function given as its first argument to its second parameter, a signal.

Listing 3.8: Signal.map

```haskell
map : (a -> result) -> Signal a -> Signal result
```

With this function in mind we can use our first signal: `Window.dimensions`. `Window.dimensions` is a signal of type `(Int, Int)`, thus we can use our `canvas` function and `map` to create a new variable of type `Signal Canvas` as shown in Listing 3.9.

Listing 3.9: resizing canvas

```haskell
c = map canvas Window.dimensions
```

Now we have created a `Canvas` which automatically resizes, if the browser’s dimensions change. Elm also defines an infix operator `∼` as alias for `map`. Therefore we can rewrite Listing 3.9 as `c = canvas ∼ Window.dimensions`.

3.3.1. Circle at constant position

Since we now have our canvas on which to draw, we can define the position at which to draw the circle. Given that we might want to change the position over time, we should model it as a signal. Nevertheless, we will start with a constant position. To create a constant signal, meaning a signal, which values does not change over time, Elm provides the function `constant : a -> Signal a`. The circle’s position is thus easily defined as shown in Listing 3.10.

Listing 3.10: constant position

```haskell
pos : Signal (Int, Int)
pos = constant (15, 15)
```

We now have everything in place to call `circleAt`: a `Canvas` to draw, the `Canvas`’s dimensions and the position. The types however do not match exactly, since all parameters are wrapped inside `Signals`. In order to circumvent this problem we would need a `map`-like function which works with multiple `Signals` and a function that takes the appropriate number of arguments. In this case we can use `map3` (Listing 3.11).

Listing 3.11: Signal.map3

```haskell
map3 : (a->b->c->result) -> Signal a -> Signal b -> Signal c -> Signal result
```

We can now put everything together and write our first main function using `Signals`. Listing 3.12 shows the result. The main function has type `Signal Element` and displays a red circle in the top left corner. As we used `Window.dimensions`, the functions output is updated every time we resize the window, although in this case it has no visible effect.
3. Elm

Listing 3.12: circle at constant position

```haskell
main =
let
  c = canvas <- Window.dimensions
  pos : Signal (Int, Int)
  pos = constant (15, 15)
  in
    map3 circleAt c Window.dimensions pos
```

In our first program we used `map3` to combine multiple signals with our `circleAt` function. Though Elm defines `map` through `map5` it is a rather inflexible way to use these. An alternative is to use the following two functions (Listing 3.13).

Listing 3.13: `∼` and `∼`

```haskell
(<~) : (a -> b) -> Signal a -> Signal b
(∼) : Signal (a -> b) -> Signal a -> Signal b

<~ was already used in our program. ∼ is specifically designed to be combined with <~ and allows us to combine any number of `Signals` with a suitable function. This allows us to rewrite Listing 3.12 line 8 as `circleAt <∼ c ∼ Window.dimensions ∼ pos`.

3.3.2. Circle following the mouse

Now that the basic structure is in place and we have some experience with signals, let’s gradually change the program so that the circle is spinning around the mouse, following it, as the user moves the mouse. First off all, let us make the circle follow the mouse; for this we have to redefine our position signal. Elm defines a signal `Mouse.position : Signal (Int, Int)`, which returns the current mouse position. As the types already match, we can just replace our constant signal function with `Mouse.position` and get the program in Listing 3.14. From now on the circle follows the mouse around as the user moves it.

Listing 3.14: follow mouse

```haskell
main =
let
  c = canvas <- Window.dimensions
  pos : Signal (Int, Int)
  pos = Mouse.position
  in
    circleAt <∼ c ∼ Window.dimensions ∼ pos
```

3.3.3. Circle spinning around the corner

Next, we try to produce a spinning circle and then we can try to combine both behaviors to gain our desired result. Once again we only have to redefine our position signal. This time the value has to change in dependence of time. Thus, let us define
3. Elm

a simple function `spinningPoint` of type `Time -> (Int, Int)`. Since `Time` actually is a
type in Elm, we can use this definition directly in our program. Elm also defines a func-
tion `inSeconds : Time -> Float`, which converts the time to seconds. Now we can define
`spinningPoint` as shown in Listing 3.15.

Listing 3.15: spinningPoint

```elm
spinningPoint : Time -> (Int, Int)
spinningPoint t =
    let
        x = 50 * cos ((inSeconds t) * 2 * pi)
        y = 50 * sin ((inSeconds t) * 2 * pi)
    in
        (round x, round y)
```

Although `spinningPoint` has type `Time -> (Int, Int)` and therefore resembles our def-
inition of a signal, to actually create a signal, we have to combine it with one of Elm’s
signals. In this case we need a signal that gives us the current time and updates period-
ically. One possibility is the function `every`. `every` takes an interval `t` of type `Time` and
then updates periodically. Accordingly, `every 50` will update every 50 milliseconds (Elm
uses milliseconds for `Time`). Hence, we can create a periodically updating signal using
`spinningPoint` and `every`: `spinningPoint ` ~ `every 50`.

The resulting main function can be seen in Listing 3.16.

Listing 3.16: spinning circle

```elm
main =
    let
        c = canvas `~` Window.dimensions
        pos : Signal (Int, Int)
            pos = spinningPoint `~` every 50
    in
        circleAt `~` c `~` Window.dimensions `~` pos
```

3.3.4. Circle spinning around the mouse

We can now concentrate on combining the mouse following circle and the spinning circle.
This is were functional reactive programming has its strength compared to imperative
frameworks. In our case all we really need to do, is use vector addition on our two signals.
Since we can combine normal functions with signals using `map`, we only have to worry
about doing this for `(Int, Int)`, instead of `Signal (Int, Int)`. We call this function `plus`
(Listing 3.17).

Listing 3.17: plus

```elm
plus : (Int, Int) -> (Int, Int) -> (Int, Int)
    plus (a, b) (c, d) = (a + c, b + d)
```

Putting it all together we end up with a new main function shown in Listing 3.18.
Once again we only had to change our definition of `pos`. In this case using `map2` and
`plus` to combine the current mouse position with our spinning point signal.
3. *Elm*

Listing 3.18: spinning circle, following mouse

```elm
main : Signal Element
main =
  let
    c = canvas ◁ Window.dimensions

  spinning : Signal (Int, Int)
  spinning = spinningPoint ◁ every 50

  pos : Signal (Int, Int)
  pos = map2 plus Mouse.position spinning
  in
    circleAt ◁ c ◁ Window.dimensions ◁ pos
```

3.4. Concurrency

In the previous sections we took a look at the basic usage of Elm and signals. What we have not seen yet is any type of concurrency. As Elm calls itself a language for *concurrent* functional reactive programming, it is now time to attend to this feature. For this it is important to understand a little about Elm’s handling of signals. After addressing signal handling, we will take a look at how we can manipulate Elm’s control flow with `async`.

3.4.1. Signal handling

The concept of FRP is based on the *instant update assumption*. This means that everything is updated instantaneously, meaning computation takes no time. In practice this is not possible. Furthermore since *behaviours*, and as a result signals, are meant to be continuous, the whole program has to be recomputed as fast and as often as possible. If the instant update assumption would hold, this would not be a problem, but in reality this introduces a variety of issues. Since computation does take time and depending on the functionality may take quite a lot of it, the program may not be updated fast enough, if the signals values actually do change.

To avoid this unnecessary computations, Elm implements all signals as discrete events. This way only if some signal’s value actually does change, the program has to be recomputed. More specific, re-computation only has to be done for parts dependent on the changed signal.

Elm also assumes that all signals have a distinct order in which they occur, especially no two signals change at the same instant. Thus events happen one at a time and can be processed as such.

Every Elm program can be represented as a signal graph. Reactive primitives are depicted as nodes and directed edges show indicate the flow of information.

Figure 3.1 shows the basic components of a signal graph. This representation helps to understand Elm, since Elm uses *threads* and *channels*\(^1\) to implement programs. Each node is a concurrent thread and each edge a channel. Since signal changes are distinctly

---

\(^1\)A channel is a form of message passing between processes, following the FIFO concept
ordered, a global event dispatcher updates all input nodes. So every time a new event occurs the dispatcher updates all input nodes. Nodes that are not affected by the event, are not modified and propagate a "value not changed" token. If on the other hand a node is affected by an event, its value is updated and passed on to its child nodes. Nodes that depend on the result of other nodes, are not updated until all preceding nodes propagated a value, either a new one or a "not changed". This prevents inconsistent and unpredictable states. Let us consider the program in Listing 3.19. The program uses the x-position of the mouse and prints this value and its square as a tuple. The tuple constructor depends on two values, which in turn depend on the same signal, namely Mouse.x. If the tuple-constructor node would not wait for a value update from both its predecessors, one could already be updated and the other not, resulting in inconsistent output. Since the pair of values printed on the screen would not match the expected result.

Listing 3.19: inconsistent states?

```elml
main = asText <- (map2 (,) Mouse.x <| (\x -> x * x) <| Mouse.x)
```

### 3.4.2. Async

Although the synchronization of all inputs per node is often necessary to avoid undesired states, it can also lead to problems concerning reactivity. Let’s take a look at a small variation of our spinning circle program. Figure 3.2 shows the modified example’s signal graph. For clarity we only look at drawAt. In this variation drawAt’s third argument changed from a circle to some expensive operation dependent on the Mouse.click signal. This operation somehow computes a new form to draw each time the user clicks the mouse. Now lets assume that the user constantly moves the mouse around, but just sporadically clicks.
Since `drawAt` waits for all inputs to generate a value, whenever the expensive operation is performed, all effects of further mouse movement is delayed until the new form is calculated. In this program one could argue, that the user would have a more positive experience, if the position updates where not delayed by other calculations. To achieve this Elm provides the `async` primitive. `async` takes as argument a `Signal`. It then performs a local graph rewrite adding a new input node, basically splitting the graph in two. The resulting graph is shown in Figure 3.3. Now whenever `expensive` is calculated, mouse movements will not be ignored, since the new input node only changes its value, when `expensive` is finished and the global event dispatcher propagates the result. The corresponding code would look like Listing 3.20.

Listing 3.20: usage of async

```elm
drawAt <- Window.dimensions ~ Mouse.position ~ async (map expensive Mouse.click)
```

Figure 3.2.: signal graph `drawAt`
Figure 3.3.: signal graph drawAt - using async
4. Conclusion

My conclusion is split into two parts. First about the general concept of functional reactive programming and then a conclusion about the programming language Elm specifically.

4.1. FRP

My conclusion concerning functional reactive programming is definitely positive. The basic concept behind it is easy to understand and if familiar with functional programming there should be no trouble seeing, that FRP provides the benefit of a high level of abstraction for reactive programming. Since graphical user interfaces are by definition reactive programs, this is an area of programming that can definitely take advantage of FRP.

4.2. Elm

Elm’s goal is to be a concurrent language for GUI programming. This is a goal it definitely meets. On the other hand, it is questionable to what extend it really provides a practical way. Though it provides a multitude of examples on the website, Elm’s documentation is really poor. Many of the examples do not have any kind of documentation in form of comments or otherwise. Their is also no API documentation. As such one often times has to look into Elm’s library code to get an idea what functions exists and what signatures they have. This results in a relatively hard to learn language and some form of familiarity with functional programming is absolutely advisable before learning Elm. As a result it might be easier to just use a library in a language one already knows. Besides the hard learning curve (mainly if you have no experience with functional programming), Elm introduces some new concepts to FRP, especially with its support for concurrency. As such Elm might have a bright future or at least the new ideas introduced with it.
Bibliography


A. Source Code

Listing A.1: Spinning circle, following mouse - complete program

```haskell
import Graphics.Element (. . )
import Graphics.Collage (. . )
import Color (. . )
import Mouse
import Signal (. . )
import Time (. . )
import Window

drawAt : (Int , Int ) -> (Int , Int ) -> Form -> Form
drawAt (dx , dy) (x , y) f =
  let
    pos = toFloatT (x - dx // 2 , dy // 2 - y)
  in
    move pos f

toFloatT : (Int , Int ) -> (Float , Float)
toFloatT (a,b) = (toFloat a , toFloat b)

type alias Canvas = (List Form -> Element)
canvas : (Int , Int ) -> Canvas
canvas (a , b) = collage a b

main : Signal Element
main =
  let
    c = canvas <- Window.dimensions

    spinning : Signal (Int , Int)
    spinning = spinningPoint <- every 50

    pos : Signal (Int , Int)
    pos = map2 plus Mouse.position spinning
  in
    circleAt <- c ~ Window.dimensions ~ pos

circleAt : Canvas -> (Int , Int ) -> (Int , Int ) -> Element
circleAt f dim pos =
  let
    c = filled red (circle 15)
  in
    f [drawAt dim pos c]
```

17
A. Source Code

`spinningPoint : Time -> (Int, Int)`

`spinningPoint t =`  
`  let`  
`    x = 50 * cos ((inSeconds t) * 2 * pi)`  
`    y = 50 * sin ((inSeconds t) * 2 * pi)`  
`  in`  
`    (round x, round y)`

`plus : (Int, Int) -> (Int, Int) -> (Int, Int)`

`plus (a, b) (c, d) = (a + c, b + d)`