Using Functional Reactive Programming for Robotic Art: An Experience Report

Eliane Irène Schmidli
OST University of Applied Sciences of Eastern Switzerland
Rapperswil, Switzerland
eliane.schmidli@ost.ch

Farhad Mehta
OST University of Applied Sciences of Eastern Switzerland
Rapperswil, Switzerland
farhad.mehta@ost.ch

Abstract

The control software for robotics applications is usually written in a low-level imperative style, intertwining the program sequence and commands for motors and sensors. To describe the program's behavior, it is typically divided into different states, each representing a specific system condition. This way of programming complicates the comprehension of the code, making changes to the program flow a tedious task. Functional Reactive Programming (FRP) offers a composable, modular approach for developing reactive applications. To examine the strengths and limitations of FRP compared to the conventional imperative style, the control software for a robotic artwork was implemented using a form of FRP in the Haskell programming language. The resulting design separates the control of the hardware from the implementation of its behavior. In addition, state transitions are presented more clearly, resulting in code that is more understandable, especially for people with little programming experience.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Applied computing** → *Media arts*; *Performing arts.*

Keywords: Functional Reactive Programming, Robotics

ACM Reference Format:
Eliane Irène Schmidli and Farhad Mehta. 2024. Using Functional Reactive Programming for Robotic Art: An Experience Report. In *Proceedings of the 12th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '24), September 2, 2024, Milan, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3677996.3678288

FARM '24, September 2, 2024, Milan, Italy

ACM ISBN 979-8-4007-1099-5/24/09
https://doi.org/10.1145/3677996.3678288

1 Introduction

The conventional approach to implementing control software for robotic art is often based on encoding a state machine in a low-level imperative language, whereby commands are sent to motors and sensors based on the system's current state. This way of programming limits the complexity of what can be done and leads to challenges in understanding where state transitions are triggered, making changes to the code error-prone. Additionally, it seems that programming in this style is not something that artists are confident about or enjoy.

The main motivation of this work was to explore how well functional programming would be received by artists: Given that artists have a strong appreciation of beauty and elegance, could it be true that the area of art is well suited to the use of functional programming, which similarly emphasizes simplicity, beauty, and elegance? The application of functional programming for art has a rich history [3, 5, 12].

FRP is a composable, modular alternative for programming reactive applications. In FRP, instead of sending commands directly to motors, a signal is created that represents the current behavior of the artwork. This signal is dynamically adapted based on inputs such as sensor data and can be interpreted by various outputs. For example, the signal can indicate the current positions to which the motors of the real artwork should move. Alternatively, the positions of the same signal can be used in a simulation whereby the Graphical User Interface (GUI) places the elements of the artwork accordingly.

To illustrate the application of FRP in the programming of robotic art, a design for the control software of the artwork Edge Beings was developed. The concept for this artwork was created by Pors & Rao [15] and is currently being realized. The idea features a composition of different-sized panels mounted on walls (see Figure 1). Over time, round, black silhouettes of creatures move slowly over the edges of the panels.

Close observation reveals that there are social constellations among the beings. They belong to different, overlapping groups, of which only one can be in view at a time. Each group has a suppressor, displacing weaker groups to make room for its own group. If the suppressor is weaker than the currently visible group, it must retreat.

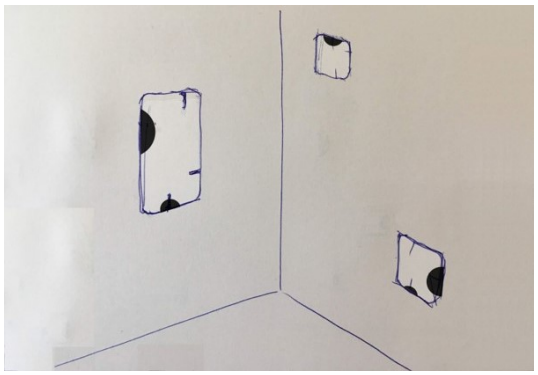

Figure 1. A sketch of Edge Beings by Søren Pors [15]. © Søren Pors, Pors & Rao.

When viewers closely approach the artwork, they observe the beings becoming aware of their presence. In response, the beings behave more nervously, refraining from emerging fully. Instead, they peek briefly over the edge before hiding again. As soon as no one is too close anymore, the creatures resume their normal behavior.

A special feature of Pors & Rao's artworks[1] is the lifelike appearance of the creatures' movements. To achieve this, they develop complex behavioral algorithms, which are neglected in this paper to maintain focus on the concepts of FRP and its significance and implications. Instead, a simplified design was created and implemented once in imperative style and once in FRP style using Yampa [6] (see Sections 3.1 and 3.2). Section 2 provides an initial introduction to the concept of FRP and the use of Yampa. The resulting code design in FRP was discussed with the artists (see Section 3.3).

2 FRP and Yampa

This section provides an overview of the key concepts in FRP and Yampa [6] that are relevant to understand the content of this paper.

2.1 FRP

FRP is a style of programming for implementing reactive systems using the principles of functional programming. Reactive systems are systems whose behavior depends on the occurrence of external events. Functional programming is defined as programming using pure (mathematical, side-effect free) functions and values elegance, algebraic compositionality and mathematical precision over quick wins in speed, space and size of user base. Therefore, FRP proposes a declarative approach to implement the behaviors of reactive systems.

A simplified example of how to think of FRP is a spreadsheet. A spreadsheet consists of cells that can contain values or functions. For example, a function can check whether the value in a particular cell is larger than the value in another particular cell. When programming the function, the user describes the dependency of the result on the two cells (`result = cell1 > cell2`). If the value in a cell is changed, the result is automatically updated. The refreshing is done by the spreadsheet program. In imperative programming, the refreshing of the values has to be specified by the programmer [1, Chapter 1].

In 1997, the concept of FRP was proposed by Conal Elliott and Paul Hudak [2], and implemented in the Fran system. Fran is a domain-specific language (DSL) for animations in Haskell. Fran makes it possible to separate the presentation of animations from their descriptions.

Two years later, these concepts were applied to Frob [13], a DSL for use in robotic systems. Frob hides the details of low-level robot programming and makes programming more hardware independent. Unlike Fran, Frob must also manage hardware control, which adds an additional layer of complexity.

2.2 Yampa

Yampa is an FRP implementation inspired by Fran and Frob. Yampa has been used in various areas such as robotics, GUI applications, and games [6]. For instance, Pembeci, Nilsson, and Hager [11] have built vision-guided, semi-autonomous robots with FRP using Yampa.

2.2.1 Signals and Events.

The most important concepts of FRP in Yampa are `Signals` and `Events`. A `Signal` is a continuous, time-varying value. It can be understood as a mapping `Time -> a` of a value of type `Time` to a value of type `a`.

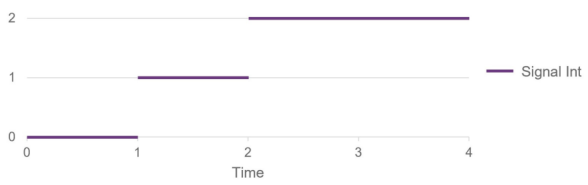

Figure 2. An example of an integer `Signal` that changes its value over time.

Figure 2 visualizes an example of an integer `Signal`. A `Signal` always contains a value at each point in time. An FRP program describes how the value of a `Signal` changes in response to events.

A stream of events is modeled by `Signal (Event b)` in Yampa, which is a `Signal` that yields either nothing or an `Event` with a value of type `b`. The value of type `b` is generated each time the event occurs. For example, the current position of the mouse can be attached to the event. A visualization

[1] An insight into the artworks and working methods of Pors & Rao can be found here [4, 16].

of an event stream `Signal (Event Int)` can be found in Figure 3.

Figure 3. An example of a `Signal` that yields nothing or an `Event` containing a value of type `Int`.

The concept of `Signal` enables the writing of programs that have time and space leaks. The reason for this is explained in [6, 9, 14]. Yampa solves this problem by not allowing Signals as first-class values. The value contained in a `Signal` can only be modified using signal functions. These cannot be constructed directly, but only using a set of given combinators. The given combinators ensure that time and space leaks are avoided. In Yampa, this concept was implemented with the help of arrows, a generalization of monads proposed by John Hughes [7].

A signal function `SF a b` is an instance of the `Arrow` class and can be thought of as a mapping `Signal a -> Signal b` of Signals to Signals.

2.2.2 Example. To better understand how signal functions work, this section provides a small example. It creates an animation of a straightforward motion of an object. The animation can be started and controlled using mouse clicks. Depending on whether the left or the right button is clicked, the current speed is slow or fast. The result is a signal that represents the position. Its value should be zero at the beginning and after a mouse click, increase with the corresponding speed. The resulting signal can be used to create an animation of an arbitrary object as shown in Figure 4.

Figure 4. Animation of a star depending on the position `pos`. The direction is determined by the definition of the coordinates in the GUI. E.g. the coordinates on the left correspond to a movement along the x-axis, whereas those on the right result in a movement along the y-axis.

Creation of a Position Signal. The example starts with the creation of a signal function with `constant` to represent the velocity. It is of type `constant :: b -> SF a b` and creates a signal function providing an output signal with a constant value.

```
1  type Velocity = Double
2
3  velocity :: Velocity -> SF () Velocity
4  velocity v = constant v
```

To calculate the position based on a constant velocity, the value of the input signal must be integrated with the signal function `integral` (see Figure 5).

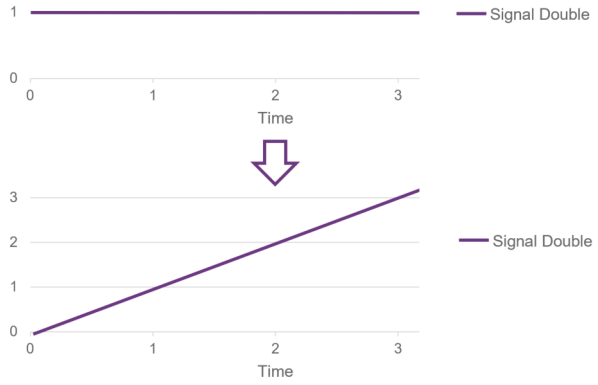

Figure 5. Visualization of `integral` accumulating a constant input value over time.

To pass the output signal of `velocity` to the `integral` signal function, as in Figure 6, a combinator is needed. One possibility to compose two signal functions is the operator `>>>`.

```
(>>>) :: SF a b -> SF b c -> SF a c
```

Now the function `positionSF` can be defined that will increase the value of the resulting signal steadily. This can be interpreted as a movement with a constant speed `v`.

```
1  type Position = Double
2
3  positionSF :: Velocity -> SF () Position
4  positionSF v = velocity v >>> integral
```

Figure 6. Visualization of `positionSF`. The arrow represents the signal that is modified by the signal functions `velocity` and `integral`.

Start on Mouse Click. We want the movement to start when a mouse click occurs. So, the velocity should first be zero. After the mouse click the velocity should be set to the slow speed (here ten). For this, the function `hold` is needed. This function receives a start value as a parameter and creates a signal function from it. This signal function takes an event stream as input and produces an output signal. In the beginning, the output signal holds the start value. When an event occurs, the value is replaced by the value of the event.

```
hold :: a -> SF (Event a) a
```

The output signal of `hold` should indicate the new speed after the click. For this, the velocity needs to be attached to the event using the function `tag`.

```
tag :: (Event a) -> b -> (Event b)
```

Figure 7 visualizes the transformation of the event stream to the position signal. First, the input signal yields nothing and `hold` produces a signal containing the value zero. Therefore, the output signal of `integral` also indicates zero. When a mouse click event occurs, ten is attached to it, and `hold` changes the value accordingly to ten. After the change, `integral` starts increasing the value of the position.

Figure 7. Visualization of `positionSF` transforming the event stream to the position signal. Note that `click` is of type `Event()`.

The new version of `positionSF` receives the mouse event of type `Event()` as an input signal. To simplify the readability of the implementation, the arrow notation is used here. An introduction to the arrow notation can be found in [8].

```
1  positionSF' :: SF (Event ()) Position
2  positionSF' = proc click -> do
3      v <- hold 0 -< tag click 10
4      pos <- integral -< v
5      returnA -< pos
```

The extracted value of the input signal is `click` of type `Event()`. The event is tagged with ten and is passed to the signal function (`hold 0`). This produces an output signal containing the current velocity. The velocity `v` can be passed to the function `integral` which produces the output position `pos`. With `returnA` the value `pos` is wrapped in the outgoing signal.

Switch between Speeds. To switch between two different speeds depending on the left and right click of the mouse, two event streams must be used. This can be realized by changing the input signal of `programSF'` to the type `(Event(), Event())`. Then a different speed is tagged to each of the two click events. To use `hold`, both event streams have to be merged into one. For this, a decision must be made as to which of the two events has priority when both occur simultaneously. With `rMerge`, the right event is always preferred as in Figure 8.

```
rMerge :: (Event a) -> (Event a) -> (Event a)
```

Here is the new implementation of `positionSF'`, where `lbp` corresponds to a left mouse click, and `rbp` to a right mouse click. Each time the left mouse button is pressed, `v` is changed to the value ten, and when the right button is pressed to the value twenty.

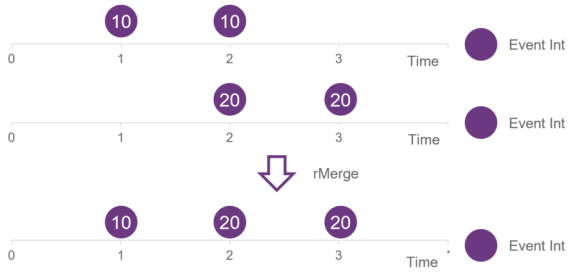

Figure 8. Visualization of `rMerge`. It merges the two event streams preferring the events from the second stream.

```
1  positionSF'' :: SF (Event (), Event()) Position
2  positionSF'' = proc (lbp, rbp) -> do
3      let slowEv = tag lbp 10
4      let fastEv = tag rbp 20
5      v <- hold 0 -< rMerge slowEv fastEv
6      pos <- integral -< v
7    returnA -< pos
```

3 Design of Edge Beings

To illustrate the differences in applying imperative programming versus FRP using Yampa, the following chapters present the behavior of a being in both approaches. The implementation of the Edge Beings artwork shown here is simplified to illustrate the concepts without going into fine details. In particular, the beings are currently assigned to a single group instead of several groups, and their movement patterns are not as complex.

3.1 Imperative Style

In imperative programming, the translation of a problem description into code often involves the creation of a state machine. In the context of the Edge Beings artwork, a state is managed for each being. The main loop adjusts these states at each iteration based on events that have occurred. The code execution for each being is determined by its current state, with the following code specifying actions for each state per iteration. Figure 9 illustrates a state machine representing the code.

```
1  def step():
2    current_time = time.time()
3
4    if state == HIDE:
5      motor.go_to_position(0)
6      set_time_next_move(wait_before_peek)
7      state = PEEK
8
9    elif state == PEEK:
10     if (current_time >= time_next_move):
11       motor.go_to_position(peek_pos)
12       set_time_next_move(wait_before_stand)
13       state = WAIT_PEEK
14
15   elif state == WAIT_PEEK:
16     if (current_time >= time_next_move):
17       if (not motion and is_stronger_suppressor):
```

```
18          suppressor_events.insert(suppressor_rank)
19          state = STAND
20        elif (is_allowed_to_go_out):
21          state = STAND
22        else:
23          state = HIDE
24
25    elif state == STAND:
26      set_time_next_move(out_time)
27      motor.go_to_position(stand_pos)
28      state = WAIT_STAND
29
30    elif state == WAIT_STAND:
31      if (current_time >= time_next_move):
32        if (is_suppressor):
33          current_group = 0
34        state = HIDE
```

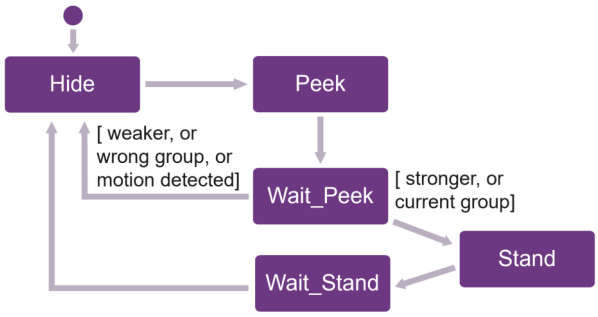

Figure 9. State Machine representing the state transitions for a being of the simplified Edge Beings implementation.

In the `'Hide'` state, a being's motor moves to the hiding position. After the transition to the `'Peek'` state, the being waits and moves to the peeking position after a specified time, followed by waiting again in the `'Wait_Peek'` state. Subsequently, the program checks if the being is permitted to emerge completely or if it must hide again based on the following conditions: If the motion sensor detects a visitor too close to the artwork, no being is allowed to move to the standing position. Otherwise, the code checks if the being is a suppressor. If not, the being can only creep out if it belongs to the currently permitted group. If the being is a suppressor, it can emerge only if it outranks the current group. In this case, the being registers for the suppressor event prompting the current group to hide.

In the `'Stand'` state, the motor moves the being to the standing position. The being then waits in the `'Wait_Stand'` state until eviction or the expiration of its allocated time. In the second case, suppressors reset the current group to zero before retreating, enabling other suppressors to emerge.

The code of the state machine only shows certain state transitions. For example, it is not apparent that all beings are set to the `'Hide'` state when a motion event occurs. For this insight, examining the event handling in the main loop is necessary. The events are gathered in a list which is processed with each main loop iteration. The function `handle_events` invokes the handlers for the current events.

```
1  def handle_events(event_list):
2    handle_suppressor_events(event_list)
3
4    motion_event = last_motion_event(event_list)
5    if is_motion_event(motion_event):
6      handle_motion_events()
7
8    if is_no_motion_event(motion_event):
9      handle_no_motion_events()
```

The function `handle_motion_events` sets all beings to the `'Hide'` state, while the function `handle_suppressor_events` manages the suppressors that were previously registered in the `'Wait_Peek'` state. The function determines the current group by the suppressor with the highest rank. Then, beings beyond the peeking position belonging to the previous group are set to the `'Hide'` state.

The sequence of event handler calls in the function `handle_events` is critical and depends on state changes within the event handlers. To illustrate this, the bad handling of motion events in the following code can be considered.

```
1  def handle_events_wrong(list):
2    # ...
3    motion_event = has_motion_event(list)
4    if motion_event:
5      handle_motion_events()
6
7    no_motion_event = has_no_motion_event(list)
8    if no_motion_event:
9      handle_no_motion_events()
```

Instead of handling only the last movement event, both event handlers are called here. If the sensor emits the three events `Motion`, `No_Motion`, and `Motion`, the bad implementation would inaccurately set the variable `motion` to `False`. The beings are allowed to emerge even if the last event signifies that a visitor is indeed present in the area.

Therefore, understanding the complete dependencies between states and events is crucial to making changes to this imperative implementation. Failure to do so may result in unintended side effects. The next chapter proposes a solution to this using FRP in Yampa.

3.2 Design in Yampa

In FRP it is possible to separate the behavior of the beings and the instructions given to the motors. The behavior is defined within a signal function, generating a signal that indicates the current positions of the beings. This signal can be consumed by motors that move to the corresponding positions. For instance, if a being is instructed to wait, the same position is repeatedly sent during this interval, causing the motors to remain stationary.

The signal function dynamically changes the behavior and therefore the positions of the beings based on the input signals. For instance, when the motion sensor near the artwork detects someone in its proximity, it sends a message to the FRP service. This message is then integrated into the

input signal of the ongoing signal function and used to generate an event. This event triggers a change in the behavior leading to a change in position and motor movement. Then, the viewer of the artwork can observe how the figures are hiding. The beings resume normal movement as soon as the viewer moves out of the range of the sensor.

3.2.1 External Interface. All the panels' motors and the sensors are orchestrated by a Python framework running on a Raspberry Pi. Simultaneously, the FRP implementation is launched on the same Raspberry Pi. Sockets are used to establish a connection between the Python service and the Haskell service, enabling the exchange of input and output information.

This design makes it possible to separate the description of the beings' behavior from the representation of the output. Modifications to the behavior therefore become immediately visible without the need to edit the motor controls. Furthermore, the Python service can be replaced with an alternative solution. Specifically, a simulation was developed for the Edge Beings artwork. This enables users to visually observe the effects of modifications in the program without access to the physical artwork hardware. Figure 10 shows a screenshot of the GUI from the resulting application.

The simulation interfaces with the Haskell service via the same socket connection. Keyboard inputs emulate the motion sensor, while the GUI adjusts the coordinates of the beings instead of controlling physical motors. The shift in positions is perceived by the viewer as an animation.

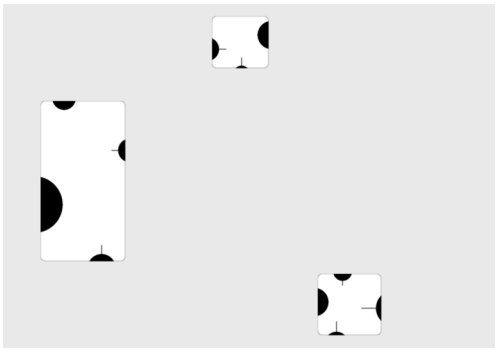

Figure 10. GUI of the Edge Being simulation

The following sections provide a deeper insight into the implementation of the beings' behavior.

3.2.2 Execution. The behavior of the beings within the Haskell service is described in a signal function. In Yampa, this function can be executed indefinitely using the `reactimate` function, enabling the reception of new inputs and the processing of outputs.

```
reactimate :: Monad m
  -- Initialization action
  => m a
  -- Input sensing action
  -> (Bool -> m (DTime, Maybe a))
  -- Output processing action
  -> (Bool -> b -> m Bool)
  -- Signal function
  -> SF a b
  -> m ()
```

Further information about the usage of the `reactimate` function can be found in [10]. For Edge Beings, the function is used as follows:

```
type Pos = Double
type Vel = Double
edgeBeingsBehavior :: SF (Maybe Int) [Pos]

reactimate
  (initializationAction)
  (\_ -> inputSensingAction)
  (\_ positions -> outputProcessingAction positions)
  (edgeBeingsBehavior)
```

The `edgeBeingsBehavior` function first interprets the motion sensor data and transforms it into an event stream of type `SF (Maybe Int) (Event(), Event())`. Then the `edgeBeingsBehavior` function starts the `behavior` signal function for each being and passes the motion event stream as input. In the absence of numeric values from the motion sensor, no event is generated. If a person is detected in the proximity of the artwork, the left event is fired; conversely, the right event is fired if there is no person close.

Each being has its own motion parameters, causing the beings to move differently from each other. These parameters are stored in instances of the `Being` data type:

```
type Time = Double
type Rank = Int
type Group = Int
type Mov = (Pos, Vel)

data Being = Being
  { waitToPeek :: Time,
    waitToStand :: Time,
    waitToGoBack :: Time,
    hiding :: Mov,
    peeking :: Mov,
    standing :: Mov,
    socialGroup :: Group,
    rank :: Rank -- 0 if not suppressor
  }
```

Each being is assigned to a group, and if it belongs to the suppressors, a corresponding rank is determined. The rank of the suppressor corresponds to the priority of its group, enabling it to displace lower suppressors and their groups.

To allow for dynamic changes in the beings' movement parameters and waiting times, the list of `Being` instances is integrated into the input signal. Within the function `edgeBeingsBehavior`, each being receives its own instance, coupled with sensor events.

3.2.3 Beings' Behavior. The beings exhibit two distinct patterns of behaviors: normal social behavior and a disturbed state. During social behavior, the beings appear in groups, with weaker groups being driven away by the suppressors of more powerful groups. If the viewer approaches the artwork

too closely, a shift to the disturbed behavior is triggered. As soon as the viewer reverts to a more distant position, the beings resume their normal social behavior.

This procedure can be represented in Yampa as follows.

```
type StartPos = Pos
type Events = (Event(), Event())

socialBehavior, disturbedBehavior
 :: [StartPos] -> SF [Being] [Pos]

motionEv, noMotionEv :: SF Events (Event ())

behavior
  :: [StartPos]
  -> SF ([Being], Events) [Pos]
behavior startPos =
  socialBehavior startPos `doUntil'` motionEv
  `switch` \pos ->
    disturbedBehavior pos `doUntil'` noMotionEv
  `switch` \pos -> behavior pos
```

First, the `socialBehavior` signal function is executed, until an event in the `motionEv` event stream triggers a switch to the signal function `disturbedBehavior`. An event in the `noMotionEv` event stream leads to a switch back to the initial behavior. The `switch` function enables the transition between these behaviors and is defined by the following type signature:

```
switch
  :: SF a (b, Event c)
  -> (c -> SF a b)
  -> SF a b
```

In this signature, the first parameter represents the signal function initially executed, combined with the event stream that triggers the switch. As soon as an event of type `Event c` occurs, the signal function in the second argument is invoked. The value of the event serves as input for creating the new signal function, allowing for adaptive behavior transitions.

In Yampa, the signal function following the switch begins afresh from time zero [6]. In the context of Edge Beings, the output positions are therefore reset to zero after each switch. To avoid this, the `doUntil'` function retrieves the position values from the running signal function and attaches them to the event from the event stream. These positions are then passed to the new signal function, allowing it to commence from the specified positions.

```
doUntil'
  :: SF a b
  -> SF a' (Event ())
  -> SF (a, a') (b, Event b)
```

3.2.4 Disturbed Behavior.

When the disturbed behavior is invoked, individual signals are generated for each being using the `disturbedBeing` function.

```
integrate :: StartPos -> SF Vel Pos

disturbedBeing :: StartPos -> SF Being Pos
disturbedBeing startPos = proc state -> do
  rec
    let input = (state, noGroup)
    v <- check -< (input, pos)
    pos <- integrate startPos -< v
  returnA -< pos
  where
    noGroup = 0
```

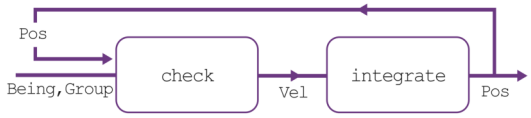

Figure 11. Visualization of `disturbedBeing` transforming the input signal into a position signal.

This function calculates the position of the being by integrating the current velocity, determined by the `check` function. If the being is in motion with a specific velocity, the position is increased accordingly. When the being stops, the velocity becomes zero, and the position remains constant. To stop at a designated target position, the calculated position is fed back to the `check` function (see Figure 11). To enable this, the keyword `rec` is used in the arrow notation.

The `disturbedBeing` signal function shares some functions with the signal function defining the normal behavior. In the latter, an input signal denotes the current group allowed to be present. As group membership does not influence the disturbed behavior, the current group is set to zero using the variable `noGroup`.

The `check` function describes the movement behavior of the disturbed being:

```
type CurrPos = Pos
type Input = ((Being, Int), CurrPos)

check :: SF Input Vel
check = goHiding
    `switch` (\par -> waitBeforePeek par
    `switch` (\_ -> goPeeking
    `switch` (\par -> waitBeforeStand par
    `switch` const check)))
```

At the beginning, the being hides, then it waits for its individual waiting time before moving to the peeking position and waiting there again. The function is called recursively to repeat the behavior from the beginning.

During each movement, a signal indicating the current velocity is generated. As an example, the implementation of `goPeeking` is shown here:

```
moveOut :: (Being -> Mov) -> SF Input Vel
arrivalPeeking :: SF Input (Event Time)

goPeeking, goHiding
  :: SF Input (Vel, Event Time)
goPeeking =
  moveOut peeking `doUntil` arrivalPeeking
```

Using `doUntil`, the signal is combined with the event stream, producing an event as soon as the target position is reached. The `doUntil` function uses the Yampa operator (`&&&`) for this purpose.

```
(&&&) :: SF a b -> SF a b' -> SF a (b, b')

doUntil
  :: SF a b
  -> SF a (Event c)
  -> SF a (b, Event c)
doUntil behavior event = behavior &&& event
```

When an arrival event occurs, the current waiting time is retrieved from the `Being` input signal and is tagged to the generated event. This time value is then used to create a wait signal function. These functions operate similarly to the movement functions, producing a signal indicating zero speed. This signal is then combined with the event stream, which generates an event after the specified time has elapsed.

```
waitBeforePeek, waitBeforeStand
  :: Time -> SF Input (Vel, Event ())
```

3.2.5 Normal Behavior. If no visitor stands in close proximity to the artwork, the beings exhibit their normal social behavior, with distinct actions initialized for suppressors and other beings. A signal of type `Group` signifies the current group and is used as input by normal beings. If a suppressor with a higher rank than the current group appears, it replaces the value of the group signal with its rank.

Normal Being. The `normalBeing` signal function is started for non-suppressors. The function operates similarly to the `disturbedBeing` signal function, with the difference that its input signal contains the current group allowed to be outside.

```
normalBeing :: StartPos -> SF (Being, Group) Pos
normalBeing startPos = proc input -> do
  rec
    v <- normalBehavior -< (input, pos)
    pos <- integrate startPos -< v
  returnA -< pos
```

If the current group aligns with that of the being, it may move out to the standing position. Otherwise, it behaves similarly to the disturbed behavior. This procedure can be seen in the `normalBehavior` function.

```
currentGroupEvents :: SF Input (Event())
notCurrentGroupEvents :: SF Input (Event())

normalBehavior :: SF Input Vel
normalBehavior =
  check `doUntil` currentGroupEvents
    `switch` \_ ->
      (creepOut `doUntil` notCurrentGroupEvents)
    `switch` const normalBehavior
```

When the event stream `currentGroupEvents` generates an event, the behavior switches from `check` to `creepOut`. If a suppressor displaces the group, the input stream's group changes, and `notCurrentGroupEvents` generates an event, restarting the `normalBehavior` recursively.

The `creepOut` function has a similar structure to the `check` function. However, after reaching the peeking position, the being moves to the standing position.

```
waitBeforeGoBack
  :: Time
  -> SF Input (Vel, Event ())
goStanding :: SF Input (Vel, Event Time)

creepOut :: SF Input Vel
creepOut = getWaitingTime
    `switch` (\par -> waitBeforePeek par
    `switch` (\_ -> goPeeking
    `switch` (\par -> waitBeforeStand par
    `switch` (\_ -> goStanding
    `switch` (\par -> waitBeforeGoBack par
    `switch` (\_ -> goHiding
    `switch` const creepOut))))))
```

To retrieve the wait parameter for `waitBeforePeek` at the start of `creepOut`, the `getWaitingTime` step is introduced.

```
getWaitingTime :: SF Input (Velocity, Event Time)
```

It generates an event immediately and reads the required waiting time from the `Being` state. This time value is then tagged to the event and passed to the `waitBeforePeek` function.

Suppressor. The `suppressor` signal function is initiated for each suppressor.

```
suppressor
  :: StartPos
  -> SF (Being, Group) (Pos, Rank)
suppressor startPos = proc input@(state, _) -> do
  rec
    (v, rank') <- supBehavior -< (input, pos)
    pos <- integrate startPos -< v

    isGoingOut <- goingOut -< (v, pos, state)
    let rank = if isGoingOut then rank' else 0

  returnA -< (pos, rank)
```

The distinct feature of the `suppressor` function is that its output signal contains beside the position also the rank, indicating zero unless the suppressor is visible. The ranks of the apparent suppressors are compared, with the highest determining the current group.

Similar to the non-suppressors, the `supBehavior` function defines the movement of the suppressor.

```
supBehavior :: SF Input (Vel, Rank)
```

The `supBehavior` function follows a similar pattern to the other beings' behavior but sets the `rank'` accordingly. During `check`, the output `rank'` is set to zero and during `creepOut`, it corresponds to the rank of the suppressor.

At the end of `creepOut`, the suppressor withdraws. The signal function `goingOut` ensures that the other beings can already emerge when the suppressor is retreating.

```
goingOut :: SF (Vel, Pos, Being) Bool
```

To achieve this, `goingOut` indicates `False` when the suppressor is moving backward. Therefore, the output signal of the `suppressor` function indicates a rank of zero.

3.2.6 Comparison to State Machine. In contrast to the state machine implementation, functions such as `behavior`, `normalBehavior`, or `creepOut` offer a more intuitive representation of the program flow and a better understanding of what reactions are triggered by certain events.

To highlight the advantages of this implementation over a state machine, the introduction of a restart event to the program is considered. In the state machine, all beings must be set to the correct state when the event occurs. This requires careful placement of the event handling to prevent states from being overwritten by other events afterward. These modifications require familiarity with the code of the state machine and the event handler. To enable an immediate restart, additional complexity is introduced by interrupting motors in motion.

In natural language, the problem can be described as executing the behavior until a restart event occurs and then restarting from the beginning. In Yampa, this description can be represented elegantly in the code, as the `restartBehavior` function demonstrates.

```
1  type Events' = (Events, Event())
2
3  restartEvent
4    :: SF ([BeingState], Events') (Event ())
5
6  startPos :: [Pos]
7
8  restartBehavior
9    :: SF ([BeingState], Events') [Pos]
10 restartBehavior =
11   (behavior startPos `doUntil` restartEvent)
12     `switch` const restartBehavior
```

It is only necessary to include the event in the `Events` type of the input signal and to ensure that `behavior` starts with the `goHiding` function. Other events and the deeper implementation of `behavior` do not need to be considered. Implementing changes in this manner reduces the likelihood of errors.

3.3 Interaction with Artists

The design for the control software was developed in close cooperation with the artists. Their developers normally program the artworks in the state machine style as described in Section 3.1. It is therefore not surprising that the artists' written description of the behavioral logic was not a problem description in natural language. Instead, it exhibited a structure that is suitable for programming a state machine.

When the implementation in FRP reached the state described in this paper, both versions of solutions were presented to one of the artists. The explanation of the code was accompanied by the illustration of the state machine shown in Figure 9 and similar graphics to Figure 11. The implementation in Yampa was convincing with its comprehensible state transitions. The artist recognized the potential to make parts of the programmed algorithm easier to understand for people without programming knowledge. This, in turn, offers a certain degree of clarity concerning the developers' understanding of the problem.

It is important to mention while Yampa facilitates comprehensible code, the possibility of producing poorly written and unreadable code still exists. Therefore, in the current state, knowledge of Haskell and Yampa is required to create or make larger modifications to the code. But the improved comprehensibility allows artists to be more involved in this process. It is also conceivable that artists could make minor changes to existing code themselves. However, this would require implementing an interface or code convention that hides certain details.

Further advantages of using FRP in art can be seen in the Edge Beings simulation. The simple addition of the simulation allowed the artists to easily experiment with various movement parameters for the creatures. To facilitate this process, sliders were integrated into the GUI for adjusting the values in the input signal `Being`. The application of FRP enables the dynamic adaptation of values in a signal and thus allows immediate modifications to the beings' movements.

4 Conclusion

The implementation using FRP offers several advantages. The modular architecture makes it easier to supplement the implementation with a simulation, enabling independent testing of code changes without the need for hardware interaction. The use of Yampa provides clear and comprehensible state changes that can also be understood by people with little programming experience. In addition, the real-time response to alterations in movement parameters is a valuable addition to the simulation's functionalities.

The implementation in Yampa requires prior knowledge of Haskell, including some advanced concepts like Arrows. Furthermore, getting started with FRP can be challenging due to varying definitions and frameworks based on different approaches. In contrast, employing a state machine approach is less complex and more standardized, aligning with the general education of programmers.

Although the initial design and major code changes still require knowledge of Haskell and Yampa, the potential for better collaboration and minor customization by the artists themselves was recognized. This suggests that functional programming, with the further development of user-friendly interfaces, could improve the state of the art in the programming of robotic art by allowing artists to better collaborate with programmers, and in the future, possibly bring programming closer to an artist's inherent need to express beauty and elegance in their work.

Acknowledgments

The authors would like to thank Søren Pors and Aparna Rao from Pors & Rao for their support and openness towards

this project, as well as the anonymous reviewers for their valuable comments and suggestions.

References

[1] Stephen Blackheath and Anthony Jones. 2016. *Functional Reactive Programming*. Manning. https://books.google.ch/books?id=aO0zrgEACAAJ

[2] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. *SIGPLAN Not.* 32, 8 (aug 1997), 263–273. https://doi.org/10.1145/258949.258973

[3] FARM. 2024. Workshop on Functional Art, Music, Modeling and Design. https://functional-art.org/. Accessed: 2024-05-03.

[4] Google Arts & Culture. 2021. PATHOS: ROBOTIC ANIMATION with Pors & Rao | Google Arts & Culture. https://youtu.be/_SRj-jpwbZ8?si=Y94cS6WexU8d-UoD. Accessed: 2024-05-09.

[5] Paul Hudak. 2000. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press.

[6] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[7] John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[8] John Hughes. 2005. Programming with Arrows. In *Advanced Functional Programming*, Varmo Vene and Tarmo Uustalu (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 73–129.

[9] Neelakantan R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. *SIGPLAN Not.* 48, 9 (sep 2013), 221–232. https://doi.org/10.1145/2544174.2500588

[10] Henrik Nilsson, Antony Courtney, and Ivan Perez. 2024. Yampa: Elegant Functional Reactive Programming Language for Hybrid Systems . https://hackage.haskell.org/package/Yampa-0.14.8/docs/FRP-Yampa.html#g:28. Accessed: 2024-05-08.

[11] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. 2002. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Pittsburgh, PA, USA) *(PPDP '02)*. Association for Computing Machinery, New York, NY, USA, 168–179. https://doi.org/10.1145/571157.571174

[12] Ivan Perez. 2023. The Beauty and Elegance of Functional Reactive Animation. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (<conf-loc>, <city>Seattle</city>, <state>WA</state>, <country>USA</country>, </conf-loc>) *(FARM 2023)*. Association for Computing Machinery, New York, NY, USA, 8–20. https://doi.org/10.1145/3609023.3609806

[13] John Peterson, Paul Hudak, and Conal Elliott. 1999. Lambda in Motion: Controlling Robots with Haskell. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*. Springer-Verlag, Berlin, Heidelberg, 91–105.

[14] Atze van der Ploeg and Koen Claessen. 2015. Practical principled FRP: forget the past, change the future, FRPNow! *SIGPLAN Not.* 50, 9 (aug 2015), 302–314. https://doi.org/10.1145/2858949.2784752

[15] Søren Pors and Aparna Rao. 2020. Pors & Rao. http://www.porsandrao.com/. Accessed: 2024-01-18.

[16] TED-Ed. 2012. High-tech art (with a sense of humor) - Aparna Rao. https://youtu.be/kJLDl2uDNaA?si=QVFeVN7Tn0ng-UJ7. Accessed: 2024-05-22.

Received 24-MAY-2024; accepted 2024-07-02