A **F**unctional **R**eactive **An**imation of a Lift using Fran

Simon Thompson
Computing Laboratory, University of Kent
Canterbury, Kent, CT2 7NF, UK
`S.J.Thompson@ukc.ac.uk`

May 1998

Abstract

This paper uses the **F**unctional **R**eactive **An**imation system, Fran, to give a simulation of a simple two floor lift (or elevator). We introduce those aspects of Fran relevant to the simulation, thus making the paper self-contained. We show how to extend the design to one for a lift with an arbitrary number of floors, and we conclude the paper with a discussion of how the Fran simulation can be verified in an informal temporal logic.

1 Introduction

This paper uses the **F**unctional **R**eactive **An**imation system – Fran – [EH97, PEL97] to give a simulation of a simple lift (or elevator). Fran is a substantial library extending the Haskell [PH97] functional programming language on Windows platforms. The work discussed here has been developed using the Hugs interpreter [Hug98]; compiled support is available using the Glasgow Haskell Compiler [Gla98]. The main architect of the Fran system is Conal Elliott of Microsoft Research, whose previous work has used C++ as a vehicle for similar ideas [E+94].

The functional approach of Fran is justified by the fact that the authoring medium for animations ought to "... give the author complete freedom of expression to say what an animation is, while invisibly handling details of discrete, sequential presentation. In other words [it] must be declarative ..."[PEL97]. This approach is familiar to the functional programmer; one can see it in influential work on 'functional graphics' [Hen82] some fifteen years ago as well as in more recent approaches to describing music in a declarative form [H+96], to name but two examples.

Fran provides two complementary modelling abstractions. First, `Behavior X` is the type of *time-dependent* values of type `X`. A time-dependent image is a graphical animation, for instance. On their own these behaviours are effectively static: once initiated they evolve autonomously. In order to react to internal or external events of various sorts, Fran provides the `Event` types, which can model, for instance, user input, timers, and a form of concurrency between components of an animation.

In this paper the Fran system is introduced in stages, and this is interleaved with a description of a version of the lift problem in Section 3.1 together with a top-down description of the lift simulation itself in Sections 3.3, 4.2, 5.2, 6.2 and 6.3. After completing the two floor case study, we examine in Section 7 how the system is extended to accommodate an arbitrary number of floors, and also give an overview of the animation of various graphical aspects of the system.

This is followed in Section 8 by a brief 'look under the bonnet' to see some of the primitives used to define the operators used in the case study; this is followed by an evaluation of Fran and how it might be developed. We also investigate in Section 10 how the temporal properties of the system can be described in a temporal logic framework, and how a verification of these properties might proceed.

The introduction to Fran given here is intended to make the paper self-contained; a more comprehensive introduction is available in the papers cited above and in animated form at [Ell97].

It is interesting to observe the positive benefits of embedding Fran in the declarative framework of the Haskell programming language. Beyond providing a natural home for a declarative modelling tool, the library is able to exploit features such as polymorphism and type classes. In writing this simulation we also have been able to exploit the power of the language in building general 'terminating' simulations (Section 9) and in writing general building-blocks for graphical interface components (Section 7.1).

It is also interesting to observe the beneficial effect of working in a typed environment: particularly when working with the libraries for `Events`, it was often possible to find the right component of the library by its type. Moreover, in nearly all cases, if a piece of code passed the type checker it was correct. It is all too easy to imagine what would happen in an untyped or weakly- typed language.

I am very grateful indeed to Conal Elliott who has answered my numerous questions, made suggestions about improvements to programming style and generally encouraged me in this enterprise. Erik Poll has made a number of very useful comments on the presentation of the material, and finally I would like to thank Howard Bowman, Helen Cameron and Peter King for their collaboration in our work on describing multimedia artifacts which has led to my looking at Fran.

2 Behaviours

This section looks at the way that continuously-evolving behaviours can be described in Fran.

2.1 Time-dependent values: `Behaviors`

Behaviours or time-dependent values of type `X` are given by the type `Behavior X`. These can be thought of as functions of type

```
Time -> X
```

where `Time` is the domain of real numbers.

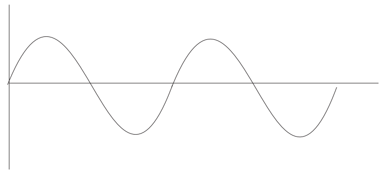
In fact, the representation of `Behavior` will be more complex for two reasons. First, the domain `Time` is more complicated than simply the reals in order to support a more effective implementation of event-detection algorithms. Further details are given in the paper [EH97] which provides a denotational semantics for Fran. Secondly, it can be more effective to work with representations of functions rather than with functions themselves. In both cases, however, the simplified model of behaviour suffices for our purposes here.

How are the `Behaviors` implemented? The system uses sampling to find values at various points, so that the animation produced is a sequence of distinct images generated at the sampling points. A discussion of the subtleties of some aspects of the implementation can be found in [EH97] as well as in the general literature on animation.

Among the types we shall use in our solution are

```
type RealB  = Behavior Double
```

and `ImageB`. An example value of type `RealB` is the `wiggle` illustrated here:



which is defined by

```
wiggle = sin (pi * time)
```

using the built in `time :: RealB` which can be thought of as the identity function.[1]

Primitive graphical objects include circles, rectangles, polylines, polygons and so forth. Pure graphical animations can be built from these and various library functions, some of which are discussed now.

`moveXY :: RealB -> RealB -> ImageB -> ImageB` [2] The first two arguments give the x and y coordinates of the position that the `ImageB` should take at each time. Note that it is not simply an image that is moved; it is an `ImageB` that can itself be moving, changing shape or colour and so on.

`over :: ImageB -> ImageB -> ImageB` This supports the super-imposition of the first image over the second, giving one form of concurrency — we discuss this further in Section 5.3.

`stretch :: RealB -> ImageB -> ImageB` This scales an `ImageB` according to a `RealB`, so allowing the size of animated objects to vary with time.

`withColor :: ColorB -> ImageB -> ImageB` The effect of this function is to change the colour of an `ImageB` according to the time-dependent colour supplied.

[1] Note here how the type class mechanism of Haskell supports overloading and in this case the use of `sin` and `pi` over `RealB` rather than `Double`. This overloading makes Fran programs substantially more readable.

[2] The double colon, `::`, should be read 'is of type', and the type here is a function taking three arguments. Functions in Haskell are in fact 'Curried' so that strictly `moveXY` takes a single argument and returns a function as result.

An example behaviour is given by

```
moveXY wiggle 0 pic1
`over`
moveXY 0 waggle pic2
```

in which `pic1` "wiggles" from left to right, `pic2` "waggles" (cosine) up and down and `` `over` `` is the infix form of the function `over`, and so superimposes the two `ImageB`s, which may themselves move, be composite, change colour and so forth.

Using these functions it is possible to build graphical animations of the non-reactive aspects of a lift simulation, such as graphics of a lift whose doors are opening, a lift whose doors are closing, a lift in motion and so forth, since these are built from blocks of colour of changing size and position.

2.2 Rate-based animation

Suppose we want a numerical quantity `f` to change with time as part of an animation. One way of doing this is to specify `f` as a function of time, as indeed we did with `wiggle` above. Fran provides an alternative, by which we specify the rate of change (or derivative) of the quantity, `f'` say, and using the `atRate` function `f` can be given as the integral of `f'`.

A simple example is given by position and velocity: a linear change in position is given by a constant velocity, for instance. This example reflects the general observation that it is often easier to describe the derivative of a function rather than the function itself, and we shall see an example of this in our lift simulation.

3 The case study: lift simulation

This paper addresses the first of the problems set for the 'Challenges for Executable Temporal Logics' workshop, June 1998 [Exe98], namely that of simulating the operation of a lift (or elevator). The problem as originally stated requires an arbitrary number of floors; in this paper we approach the problem by giving the full solution for the two floor case, and then by giving the top-level design for the general case. We have also stripped down the graphics to concentrate on the control aspects of the problem, which in the case of Fran are the aspects demanding the most effort to implement.

In our first attempt we control the operation of the lift using the mouse buttons. We show how to modify our solution to include a more general graphical control scheme in Section 7.1.

3.1 The two floor problem

The aim is to provide a graphical animation of the operation of a lift between two floors of a building. The lift can be called from the upper floor to request travel downwards; a call from the lower floor is taken to be a request to travel upwards. Because of this interpretation, there is no need to have buttons inside the lift itself, and input is taken from the mouse buttons: a left button click generates an 'up' request and the right a 'down'. In the solution presented, the lift is represented by a red blob, rather than any more complicated an animation.

There is a twist to the problem which makes its solution more complex than might first be envisaged. This is the fact that while the lift is travelling upwards there can be a further request to travel upwards, which can only be discharged by a journey back down and then back up again. In other words, some "memory" is required to solve even the case of a two-floor simulation, and shows that our simplification contains the essential elements of the original problem.

3.2 The `User` argument

Any animation which uses time information in a non-trivial way, or which interacts with the user will be defined as a function which takes a `User` argument and, on the basis of this argument, returns a `Behaviour` of some sort. The `User` argument consists of a timed stream of mouse button presses, key presses, mouse positions and other user data. It also gives the time at which the animation begins and other real-time information.

In order to exhibit such a user-driven animation in action we use the function

```
displayU :: (User -> ImageB) -> IO ()
```

which 'runs' an animation, by supplying it with the user input stream as its `User` argument, and produces primitive Haskell IO as a result.

3.3 Case study part 1: the top-level solution

We shall give the solution to the lift problem top-down, with the full code for the solution appearing in the Appendix. At the top level we define a function over a `User` argument, as explained in Section 3.2.

```
liftSim :: User -> ImageB
```

The simulation consists of a moving image of a lift,

```
liftSim u
  = moveXY xPos yPos image
```

where `xPos` and `image` are constants.[3] The definition of `yPos` and the auxiliary behaviours and events which determine it are local to the definition of the function and thus appear in a `where` clause. Note that the result depends upon the `User` argument `u` which will be used in a number of the definitions which follow.

As we explained in Section 2.2 it is often simpler to model phenomena by derivative rather than directly, and this we do here for the `yPos`:

```
yPos = lower + atRate dy u
```

where the velocity – `dy` – is piecewise constant and can take one of three values: zero, making the lift stationary; `upRate`, signifying that the lift is ascending and `downRate` for descent. We have to look at how events are modelled to see how `dy` is defined and this we do in the next section.

Observe also that the `User` argument `u` is passed to the `atRate` function to provide the timing information – such as when the animation begins – needed by the integration.

[3]In a more complex simulation `image` would itself evolve, depending upon the values of certain parameters just as the position of the lift does. Its definition would follow the pattern of that for `yPos`.

4 Events

We have seen in Section 2 how certain simple time-dependent behaviours can be defined, but in order to define behaviours which respond to internal conditions or external events the model needs to be extended by the `Event` type.

An `Event X` is a sequence of timed occurrences, each of which is associated with a value of type `X`, so it is possible to think of `Event X` as a list of type

```
[(Time,X)]
```

sorted on its first components. Each of the elements, `(t,x)` say, is called an **event occurrence**, with the whole structure being the event.[4] (As was the case for behaviours, the implementation is somewhat more complicated than this, but for the purposes of this paper this will suffice.)

4.1 Handling events

The system provides substantial support for handling and modifying occurrences of events. In our model we only perform simple transformations on events.

Each event occurrence in a stream `str` of type `Event a` will have the form `(t,x)`, where `t::Time` and `x::a`. In every case we are interested in as a part of the case study, our aim will be to convert the value `x` to a fixed value `c::w`, thus converting the pair to `(t,c)`. The resulting stream will be written

```
str -=> c  :: Event w
```

The effect on a stream `[(t0,x0),(t1,x1),...,(tn,xn),...]` is therefore to produce the result

```
[(t0,c),(t1,c),...,(tn,c),...]
```

In the general case, the event occurrence produced by the event handler corresponding to `(t,x)` may depend upon `x`,[5], `t` and the remaining part of the `Event` (after the expired event occurrences are removed). Further details of the event handling mechanism `handleE` can be found in Section 8 and [PEL97].

Streams of event occurrences can be merged, time-wise, using the operation

```
.|. :: Event a -> Event a -> Event a
```

These two capabilities allow us to proceed further with our lift case study.

4.2 Case study part 2: top-level events

The movement of the lift is controlled by a number of `Events` generated internally

```
stop, goUp, goDown :: Event ()
```

which are intended to give the obvious values to the velocity of the lift, `dy`. `()` is the trivial type, whose only member is also denoted `()`; it is used in a situation where the value contained in the `Event` occurrence is of no significance and only

[4] This terminology appears to be in conflict with the more usual 'event' (for 'event occurrence') and '(event) stream' (for 'event'); we will use the Fran terminology in this account.

[5] In which case the operation is similar to the `map` function over lists.

the `Time` value is relevant, as is the case here. The `Events` will themselves be defined in Section 6.3.

The definition of `dy` uses `setRate :: Event Double` given by

```
setRate = stop   -=> 0
          .|.
          goUp   -=> upRate
          .|.
          goDown -=> -downRate
```

The effect here is to convert all occurrences of `stop` to occurrences with the value `0`; all occurrences of `goUp` to event occurrences with the value `upRate` and so on. These are merged, and so give an `Event Double` of the form

```
[ (2.3 , upRate) , (4.9 , 0) ,
  (8.6 , downRate) ...  ]
```

in which upRate is the value returned at the occurrence at time `2.3` and so forth. In order to define `dy` this timed stream of values needs to be converted to a behaviour of the form

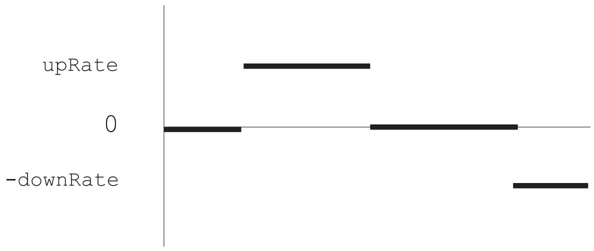
and this we investigate now.

4.3 Converting `Events` to `Behaviors`

Behaviours evolve continuously in time, while (occurrences of) events take place at discrete points in time. The power of the Fran model lies in the way in which these two types are linked. This section addresses one simple way in which `Events` can be converted to `Behaviors`; other more complex (and more fundamental) ways are examined elsewhere [PEL97]. We need to convert the event `setRate` to a behaviour. This is done by the Fran function

```
stepper :: a -> Event a -> Behavior a
```

which is parametrised by a starting value and an `Event`, and builds a piecewise constant `Behavior` from these inputs. With the starting value `0` and the event as above we have



so we define

```
dy = stepper 0 setRate
```

This shows the mechanism by which we convert streams of 'internal messages' – goUp and so forth – into a behaviour, and completes the top-level loop of the simulation. We now need to look at how these messages are themselves defined, but before that we investigate how to model system states.

5 States

5.1 Modelling states in Fran

As we mentioned in Section 3.1, the implementation of the lift will need to contain some element of memory to keep track of requests for travel that are still to be fulfilled. In an earlier version of the solution, a state monad [Wad95] was used to model the state, but this caused complications, particularly in conjunction with handling a `User` argument (which could itself be seen as giving rise to a monad).

There is a much more straightforward view of a state variable of type `X`, and that is as a `Behavior X` – an `X` value which varies with time. We thus get a declarative model of state in Fran.

5.2 Case study part 3: pending requests as 'variables'

Our model contains three 'Boolean variables'

```
upPending, downPending,
  pending :: Behavior Bool
```

which keep track of whether there is pending a request to travel up, down or in either direction. `pending` is the pointwise disjunction of `upPending` and `downPending`

```
pending = upPending ||* downPending
```

Here `||*` is the lifting of the Boolean disjunction operation `||` to `Behavior Bool`, other operations such as `==` are lifted to behaviours in a similar way below.

The state `upPending` is defined from an `Event` using `stepper` as above.

```
upPending = stepper False (setUp .|. resetUp)
```

The initial value of the variable is `False`; it is set to `True` by a request to travel upwards, and reset to `False` by the lift starting an upward journey, as signalled by `goUp`:

```
setUp   = upButton -=> True
resetUp = goUp     -=> False
```

The variable `downPending` is defined in a similar way.

5.3 Concurrency in the Fran model

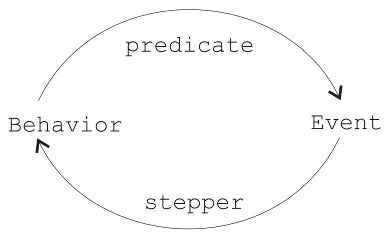
The model presented thus far appears to contain elements which evolve concurrently, in some sense at least. This section attempts to explain the nature of the concurrency in the system.

Animations – including sound as well as images – can be combined using the `‘over‘` function which places one animation on top of another, so we have a form of concurrency here.

Examining the implementation developed thus far, we appear to have `Behaviors` evolving in parallel in the lift simulation: the system contains state variables which are controlled by messages from other parts of the system, for instance. The concurrency here is completely implicit: various interdependent values – both `Behaviors` and `Events` – are defined in a single scope, and so can be thought of as evolving concurrently. This concurrency is clearly supported by the sample/display model which underlies Fran.

6 Predicates

Looking at the case study thus far, we have a model of the lift in which external stimuli are provided by mouse button presses. We need, however, to find a way of generating the ‘control’ messages, `goUp` and so forth, which are of type `Event ()` from the `Behaviors` in the system. The way in which we convert from behaviours to events gives the last piece of the Fran model used here.



6.1 The `predicate` function

In order to complete the simulation, the crucial pieces of information which we need are when we have arrived at the top or the bottom of lift shaft. We could keep an internal record of the time of departure and calculate offsets from that, but here we choose instead to check when we have arrived by means of a logical condition on behaviours. This `Behavior Bool` can be made to generate an event by means of the function

```
predicate :: BoolB -> User -> Event ()
```

which takes a Boolean behaviour and the `User` argument (for timing information) and returns an `Event`. We have to look for an appropriate `BoolB` with which to test having arrived at the top. Candidates include

`yPos ==* constantB upper` Given the sampling model, it is possible that the system will miss the point at which the condition is `True`. A fuller discussion of this issue can be found in [EH97], which makes it plain that implementations of `predicate` have changed with different releases of Fran.

`yPos >=* constantB upper` This condition will possibly be `True` over an interval of time, or in a more problematic way may become `True` arbitrarily often over a short period of time, giving rise to "event burst" as it were. This was indeed a problem in an early version of the solution.

`yPos >=* constantB upper &&* dy >* 0` Adding the condition that the lift is in motion – `dy >* 0` – gives this condition a *transitory* property: we shall see in Section 6.3 that the `Event` to which it gives rise will ensure that it becomes `False` immediately afterwards, thus avoiding the problems of the previous possibility.

We can now put together the final parts of the solution of the case study.

6.2 Case study part 4: conditions and predicates

The conditions of being at the top, and waiting stationary (at the top) are given by

```
atTop, stopped, waitingTop :: BoolB
atTop      = (yPos >=* upper)
stopped    = (dy ==* 0)
waitingTop = atTop &&* stopped
```

The `Event` of arriving at the top is defined by

```
arriveTop :: Event ()
arriveTop = predicate (atTop &&* dy >* 0) u
```

and similar definitions can be found for the bottom case.

6.3 Case study part 5: control `Events`

Recall that the top level of the simulation is driven by the 'control' `Events` `goUp`, `stop` and `goDown`. Now that we have a definition of the `Event` generated by arriving at the top (and bottom) we can define the control events.

We have seen earlier that button presses have an indirect effect on the operation of the lift by setting the pending variables; they can also have a direct effect by setting it into motion when it is stationary. We therefore define

```
upButton, downButton, eitherButton :: Event ()
upButton     = lbp u
downButton   = rbp u
eitherButton = upButton .|. downButton
```

so that the `eitherButton` event corresponds to the press of either button.

When should the lift go down; that is, when should there be occurrences of the `goDown` event? There are two cases.

- The lift can arrive at the top (`arriveTop`) when there is a request pending for travel in either direction (`pending`), or,

- either button can be pressed (`eitherButton`) while the lift is waiting at the top (`waitingTop`).

In both cases there is a condition on the event occurrences. Using the function

```
whenE :: Event a -> BoolB -> Event a
```

the expression

```
ev `whenE` cond
```

selects from `ev` precisely those occurrences at which the condition `cond` is `True`.[6] The definitions of `goDown` and `stop` are then

```
goDown = arriveTop    `whenE` pending
         .|.
         eitherButton `whenE` waitingTop

stop   = (arriveTop .|. arriveBottom)
 `whenE` notB pending
```

It is not hard to see that the lift will stop in the situation of arriving either at the top or the bottom when no request is pending. `goUp` is defined by analogy with `goDown`.

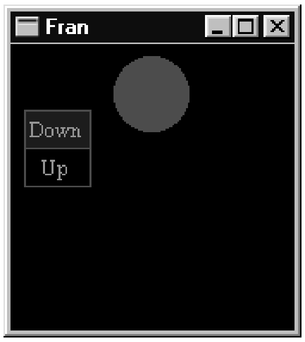
This completes the definition of the Fran lift simulation for a two floor lift. The full code is contained in the Appendix.

7 Extending the implementation

This section outlines the way in which the implementation can be given an graphical interface and how it can be extended to implement an n-floor lift.

7.1 Graphical interface

The lift is controlled by the mouse buttons in the solution presented thus far. We can modify this to include two on-screen buttons, as in the illustration



which shows the lift in motion with a pending request for travel downwards. This modification is made with minimal changes to the code presented thus far. As well as having to modify the code already written, we have to add to the code a block of buttons and for this we use a function `buttonBlock` of type

[6] `whenE` acts rather like the standard function `filter` over lists.

```
Geom ->
[(String,Event (),a)] ->
(User -> Event ()) ->
User ->
(ImageB,Event a)
```

where the arguments consist of

- The geometry of the buttons.
- Information for each button: its textual form, the event which resets it and the value identifying it in the `Event` for the button block.
- The event which presses the button.
- The `User` input.

The result consists of the image of the block paired with the `Event` which it generates.

This function is not part of the Fran library, and it is noteworthy that it can be implemented from scratch in about a hundred lines of code. This is a benefit of embedding the Fran library in a general purpose, higher-order language like Haskell.

Taking input from on-screen buttons is obviously a way of animating the input for the n-floor problem, to which we turn now.

7.2 The n-floor problem

The top-level design of the n-floor lift is similar to the two floor design given here.

- As in the two floor design, the lift is modelled using its velocity, `dy`. This will be controlled using the `stepper` function as here.
- The state of the system is more complex than in the two floor case. It can be modelled by two state variables which will be lists:
  - the first represents the requests pending from inside the lift, and
  - the second represents the requests pending from the floors of the building.

In the two floor solution, the state variables such as `upPending` have only two values. This means that when they are reset the new value is independent of the current value of the state variable. This will not be the case for the state variables here, and so we need to define a function

```
stateStepper :: (a -> b -> a) -> a -> Event b -> Behavior a
```

to manage the states. In the result of

```
stateStepper f x0 ev
```

the first value for the behaviour will be `x0`; subsequent values will be generated from the values in `ev` – `y0`, `y1`, ... say – thus: `f x0 y0` (call this `x1`), `f x1 y1` and so forth, with the `n+2`nd value being `f xn yn` where `xn` is the `n+1`st value of the variable and `yn` the `n+1`st value in `ev`.[7]

[7] The function `stateStepper` is analogous to the `scanl` of the Haskell prelude.

8 A glimpse under the bonnet

Up to this point we have used a high-level and proper subset of the facilities provided by Fran. Behaviours which evolve have been defined using `stepper`, which turns an `Event a` – that is a timed sequence of values from type `a` – into a piecewise-constant `Behavior a`. We have also handled event occurrences in a simple-minded way, using the `-=>` operator to transform occurrences `(t,v)` of the event `ev` into occurrences `(t,c)` of the event `ev -=> c`.

How in general can the occurrence of an event cause a change in behaviour; how in general are events handled? In this section we describe the two operations `handleE` and `untilB` which provide a primitive interface to behaviors and events, and we see as examples how to define `stepper` and `ev -=> c` from these primitives.[8]

8.1 Handling `Events`

Recall from Section 4 that an `Event a` is a sequence of timed occurrences, each of which has associated with it a value of type `a`. In general, three values characterise an event occurrence:

- the time of the occurrence;
- the value of the occurrence, and
- the remainder of the Event, after removing the occurrences up to and including the occurrence in question.

To handle an event occurrence, we transform these three values to a value, of type `b`, say. This is accomplished by a function of type

```
Time -> a -> Event a -> b
```

and we can apply such a function to each occurrence in a stream of occurrences to give a stream of occurrences of type `b`, that is an `Event b`. That is the effect of the general event handler,

```
handleE :: Event a -> (Time -> a -> Event a -> b) -> Event b
```

As a special case of `handleE` in which the handler function is constant we define `-=>` thus:

```
(-=>) :: Event a -> b -> Event b
ev -=> c = ev `handleE` (\_ _ _ -> c)
```

Also of interest is the `Event`-equivalent of `map`,

```
(==>) :: Event a -> (a -> b) -> Event b
ev ==> f = ev `handleE` (\_ v _ -> f v)
```

which when applied to the timed stream `[(t0,x0),(t1,x1),...,(tn,xn),...` and the function `f` gives the result `[(t0,f x0),(t1,f x1),...,(tn,f xn),...`.

[8] This section is included for completeness of exposition, but obviously owes much to [PEL97].

8.2 Modifying `Behaviors`

How can one sequence behaviours, so that one follows another? A simple solution is provided by

```
seqB :: Behavior bv => bv -> Event () -> bv -> bv
```

so that `seqB bh1 ev bh2` behaves as `bh1` until the first occurrence of `ev`, and after that behaves as `bh2`. One can also lift this sequencing to operate over behaviours dependent upon a `User` argument to give

```
seqUB :: Behavior bv => (User -> bv) -> (User -> Event ()) ->
                        (User -> bv) -> (User -> bv)
```

and so forth. In fact, Fran provides a different primitive, `untilB`, which generalises these. It is, however, instructive, to see how `seqB` is defined from `untilB`. The function `untilB` has the type

```
Behavior bv => bv -> Event bv -> bv
```

and the effect of ``bh `untilB` ev`` is to behave as `bh` until the first occurrence of the event `ev`; after that the behaviour is whatever value is returned by that first event occurrence, which does indeed return a value of type `bv`. We can then see that

```
seqB bh1 ev bh2 = bh1 `untilB` (ev -=> bh2)
```

so that `seqB` separates the event and the continuation behaviour which are combined in the second argument to `untilB`. In a similar way, we define

```
seqUB ub1 uev ub2
  = \u -> ub1 u `untilB` (nextUser_ uev u ==> ub2)
```

where the expression `nextUser_ uev u` of type `Event User` gives a stream of `Users` each aged to contain only those occurrences which follow the occurrence in question. The effect of `nextUser_ uev u ==> ub2` is thus to pass the aged `User` to the `User`-lifted behaviour `ub2`, and so to continue the computation as required.

8.3 Using `handleE` and `untilB` to define the `stepper`

The `stepper` produces piecewise constant behaviour from a stream of values of type `a` and a starting value of that type.

```
stepper :: a -> Event a -> Behavior a
```

In what follows we build the result `stepper start ev` in stages. Initially, the behaviour will be the constant behaviour with value `start`, that is

```
constantB start
```

If we think of `ev` as an infinite list, we then want recursively to call `stepper` on `(head ev)` and `(tail ev)`. We cannot do this directly, but we can *indirectly* using `handleE`. The function

```
withRestE :: Event a -> Event (a,Event a)
withRestE ev = ev `handleE` (\_ head tail -> (head,tail))
```

returns the stream of 'head,tail' pairs from the event occurrences in `ev`. To apply `stepper` to each of these pairs, we 'map' it along `withRestE ev` using the `==>` operator, as follows

```
withRestE ev ==> uncurry stepper
```

where note that we have to `uncurry` the `stepper` function to accept its argument as a pair rather than ans two separate arguments. Putting all the parts together, we have

```
stepper start ev
  = (constantB start) `untilB` (withRestE ev ==> uncurry stepper)
```

This example shows how the exception handling mechanism of `hamdleE` gives a flexible way of dealing with `Events` – as timed streams of event occurrences – and also how the primitive `untilB` turns a stream of `Behaviors` into a single behaviour.

As we have seen, using the primitives `handleE` and `untilB` together with various of their derivatives and a number of other utility programs such as `nextUser_` it is possible to define a variety of powerful programs; in the section which follows we reflect on other aspects of using Fran.

9 Reflection

The solution presented in this paper is the result of a number of iterations which reflect a growing understanding of the capabilities of the Fran library as well as different approaches to the design of the simulation itself.

For instance, in an earlier design the components of the solution were represented as terminating behaviours, which were sequenced together to form the overall simulation – a 'monadic' approach. Embedding Fran in a functional language with general-purpose capabilities makes modelling these terminating behaviours as pairs

```
( bv  , Time -> User -> Event () )
```

a straightforward matter: the second components of these pairs are used to signal the termination of the behaviour in the first component. Such behaviours are sequenced using the analogue of `seqUB` from the previous section. This 'monadic' approach can be used in situations in which behaviours are built from components which are not piecewise continuous, for example.

The solution we have presented here relies heavily on recursion. For example, `goUp` depends upon `pending` and `pending` depends upon `upPending` which in turn depends upon `goUp`. The implementation of Fran uses lazy evaluation heavily, so the recursive definition of structures is possible (we have done it here), but some recursions can lead to non-termination, and others to the system locking up. This reflects a general phenomenon for which evidence is apparent in earlier approaches to functional I/O (see, for example, [Tho90]) and which led to the definition of sets of (monadic) combinators to handle I/O in a more disciplined manner [Tho90, PH97].

10 Towards a verified lift implementation

This section sketches work in progress – namely giving a logical rendering of aspects of Fran – by means of a discussion of the verification of the lift case study.

10.1 Foundations

The Fran model addresses both continuous behaviour – by means of `Behaviours` – and also discrete behaviour, through its `Event` types. Moreover, the discrete and continuous behaviours will in general be mutually dependent, as illustrated by the figure in Section 6. How can we describe the system we have built here, and systems implemented in Fran in general?

An obvious candidate is a temporal logic [Eme90], either in a non-timed or real- time form. A variant of interval temporal logic which addresses continuous variation is the duration calculus [Zho94], and the mean-value duration calculus [ZX94] also includes discrete events. In general it remains to be seen whether this is the form of calculus most suited to reasoning about Fran systems, or whether a calculus based on functions and lists of event occurrences or indeed some other foundation is the most suitable. Certainly the function-based approach would be closer to the denotational semantics of Fran [EH97].

10.2 Towards a temporal logic for Fran

We assume here a standard dense-time temporal logic with the usual connectives of predicate logic and the modalities $\Box$, $\Diamond$ and $\mathcal{U}$ (for 'Until'), as well as adding the modality $\triangle$ for 'valid on some open interval beginning at the current time point'.[9] All these modalities can be indexed with real-number bounds to give a timed variant of the logic.

How are Fran-defined values to be rendered in the logic? We shall take the view that `Bool`ean behaviours are atomic temporal formulas[10] and we will also read `Events` as propositions which are true exactly at their occurrence points.

10.3 Verification conditions

The top level requirement of the lift can be stated as

$$\Box(\texttt{upButton} \Rightarrow \Diamond\texttt{goUp}) \wedge$$
$$\Box(\texttt{downButton} \Rightarrow \Diamond\texttt{goDown}) \qquad (\texttt{P0})$$

that is 'a request to travel is eventually discharged'. By indexing the $\Diamond$ modalities one also can put a constraint on the time taken for a request to be satisfied.

How can we argue for the validity of this requirement? As a first step we can state an invariant which is a direct consequence of the definition of `stepper` and `(i)`:[11]

[9] Because the interval is open, the current time point will not be included in it.

[10] This neglects the issues of termination; see [Tho95] for a general discussion of the axiomatisation of a functional programming language.

[11] We refer to the program text in the Appendix by means of the labels `(i)`, `(ii)` and so on.

```
□(dy ==* 0 ∨ dy ==* upRate ∨
   dy ==* -DownRate)                                      (P1)
```

with `upRate > 0` and `-DownRate < 0`, so

```
□(dy ==* 0 ∨ dy >* 0 ∨ dy <* 0)                           (P2)
```

We can also state an invariant on the position of the lift:

```
□(lower <=* yPos ∧ yPos <=* upper)                        (P3)
```

which is true initially since `yPos` starts at `lower`. How can we show that it is valid? Consider a general time point up to which the constraint holds. By `(P2)` and `(ii)` the `yPos` is either static, thus preserving the invariant, or changing.

Suppose that `dy >* 0`; now `yPos` is increasing and `dy` stays constant, with the consequence that

```
(lower <=* yPos ∧ yPos <=* upper ∧ dy >* 0)
  𝒰 arriveTop                                             (P4)
```

by `(iii)`.

We now need to verify that arriving at the top has the effect of stopping the lift. How can this be stated? By the definitions of `goDown` and `stop` (at `(iv)`) one of these two will happen at the arrival point, so setting the rate to `0` or `-downRate`. We have to be careful about a statement of when the rate is set[12] – it is here that we use the $\triangle$ operator which states that the rate is non-positive only after the time point in question:

```
□(arriveTop ⇒ △(dy <=* 0))                                (P5)
```

This is enough to ensure that the lift then descends, and so fulfills the invariant `(P3)` in the case that `dy >* 0`. A symmetrical argument ensures the result in the negative case, and so shows that `(P3)` is always the case.[13]

A similar argument will show the validity of

```
□(waitingBottom ∨ waitingTop ∨
  ◇arriveBottom ∨ ◇arriveTop)                             (P6)
```

Now, we can justify the validity of `(P0)`. We show that

```
□(upButton ⇒ ◇goUp)
```

with an argument by cases according to which of the cases in the disjunction of `(P6)` holds at the current point. The other conjunct of `(P0)` follows in a similar way. The cases are as follows.

- In the case that `waitingBottom` holds at the current point, then by the definition of `goUp`, this event holds immediately.
- Suppose that `waitingTop` holds. Motion of the lift downwards will begin by the definition of `goDown`, and by the downward analogue of `(P4)` we can infer that `◇arriveBottom`. This is the case that we consider next.

[12] A naive statement has the gradient being both positive and non-positive simultaneously.

[13] The argument here should be formalised as an induction of some sort; the duration calculus suggests such principles for piecewise-constant functions like those defined by `stepper`.

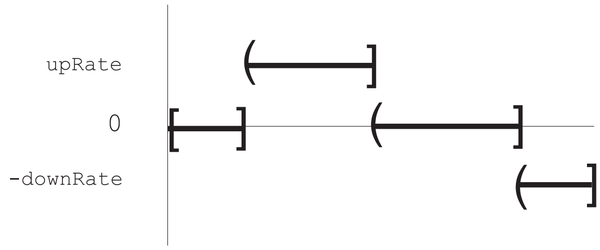
- The effect of the `upButton` will have been to set `upPending` to `True` – by `(v)` and properties of the `stepper` function – and so on arrival there will be a `pending` request, so that `goUp` will be generated by the definition of `goUp` in `(iv)`.

- Finally, we consider the case that $\diamond$`arriveTop`. The `upButton` will set `upPending` to `True`. On arrival at the top this means that there is a pending request, and so downward motion will begin. We can then argue as in the previous case.

The arguments in this section can be bounded using the parameters of the lift simulation; for instance, a journey will take `(upper-lower)/upRate` time units to complete.

10.4 Commentary

The argument given in Section 10.3 is informal, but can be formalised. The main novelty is the way in which `predicate` is handled, giving an `Event` when the lift arrives at the top or bottom. The event occurrences in turn affect behaviours through the `stepper`, and in formalising this we have used the $\triangle$ operator, which implicitly gives the stepper graph the following form



where a '(' denotes an open end of an interval and a ']' a closed end, since, as was explained earlier, we only expect the change in the stepped value to hold after the step point.

It remains to be seen how verification for the general model will proceed.

11 Conclusion

In this paper we have shown that Fran can be used to give a simulation of a lift, and we have argued that it is well suited to the task both because of its declarative model of the system and also because it is embedded in a general-purpose functional programming language, namely Haskell. This allows extensions of the library to be constructed with relatively small effort, and also allows those extensions to use features of Haskell – such as higher-order functions, polymorphism and type classes (for overloading) – in an essential way.

We have also sketched a verification of the code using a temporal logic; we expect to extend this work with a more thoroughgoing investigation of the logical foundations of the Fran model.

References

[E+94] Conal Elliott et al. TBAG: A high-level framework for interactive, animated 3D graphics applications. In *Proceedings of SIGGRAPH '94*. ACM Press, 1994.

[EH97] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP97)*. ACM Press, 1997.

[Ell97] Conal Elliott. Composing Reactive Animations. Available from `www.research.microsoft.com/~conal/fran`, 1997.

[Eme90] E Allen Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier/MIT Press, 1990.

[Exe98] Challenges for Executable Temporal Logics. Further details at `www.cs.waikato.ac.nz/~dsmith/CHALLENGES/`, 1998.

[Gla98] The Glasgow Haskell Compiler. Available from `www.dcs.glasgow.ac.uk/fp/software/ghc/`, 1998.

[H+96] Paul Hudak et al. Haskore music notation – An algebra of music. *Journal of Functional Programming*, 6, 1996.

[Hen82] Peter Henderson. Functional Geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*. ACM Press, 1982.

[Hug98] The Hugs System, Version 1.4. Available from `www.haskell.org/hugs14`, 1998.

[PEL97] John Peterson, Conal Elliott, and Gary Shu Ling. Fran Users' Manual. Available from `www.haskell.org/fran`, 1997.

[PH97] John Peterson and Kevin Hammond, editors. *Report on the Programming Language Haskell, Version 1.4*. `www.haskell.org/report`, 1997.

[Tho90] Simon Thompson. Interactive functional programs: a method and a formal semantics. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 249–285. Addison-Wesley, 1990.

[Tho95] Simon Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, 7, 1995.

[Wad95] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*. Springer-Verlag, Lecture Notes in Computer Science, 925, 1995.

[Zho94] Zhou Chaochen. Duration Calculi: An Overview. In Dines Bjørner et al., editors, *Formal methods in programming and their applications*. Springer-Verlag, Lecture Notes in Computer Science, 735, 1994.

[ZX94] Zhou Chaochen and Li Xiaoshan. A Mean Value Calculus of Durations. In A.W̃. Roscoe, editor, *A Classical Mind*. Prentice-Hall, 1994.

Appendix: The complete program for the two floor lift

```
liftSim :: User -> ImageB

liftSim u

  = moveXY xPos yPos image

    where

    image :: ImageB

    image = stretch 0.3 circle

    xPos,yPos,dy :: RealB

    xPos = constantB 0
    yPos = lower + atRate dy u                                (ii)
    dy   = stepper 0 setRate                                   (i)

    setRate :: Event Double

    setRate = stop      -=> 0
              .|.
              goUp      -=> upRate
              .|.
              goDown    -=> -downRate

    atBottom, atTop, stopped, waitingBottom, waitingTop :: BoolB

    atBottom = yPos <=* lower
    atTop    = yPos >=* upper
    stopped  = (dy ==* 0)

    waitingBottom = atBottom &&* stopped
    waitingTop    = atTop    &&* stopped

    arriveBottom, arriveTop :: Event ()

    arriveBottom = predicate (atBottom &&* dy <* 0) u
    arriveTop    = predicate (atTop    &&* dy >* 0) u        (iii)
```

```
upButton, downButton, eitherButton :: Event ()

upButton     = lbp u
downButton   = rbp u
eitherButton = upButton .|. downButton

upPending, downPending, pending :: Behavior Bool

pending     = upPending ||* downPending
                                                                   (v)
upPending   = stepper False (setUp   .|. resetUp)
downPending = stepper False (setDown .|. resetDown)

setUp, setDown :: Event Bool

setUp   = upButton   -=> True
setDown = downButton -=> True

resetUp, resetDown :: Event Bool

resetUp   = goUp   -=> False
resetDown = goDown -=> False

goDown, goUp, stop :: Event ()                                    (iv)

goDown     = arriveTop    `whenE` pending
             .|.
             eitherButton `whenE` waitingTop

goUp       = arriveBottom `whenE` pending
             .|.
             eitherButton `whenE` waitingBottom

stop       = (arriveTop .|. arriveBottom) `whenE` notB pending
```