Reasoning about functional reactive animations

Simon Thompson

TCS seminar

May 1999

Overview

Fran: functional reactive animations.

Introducing a subset of Fran . . .

. . . using the example of a lift animation.

Events and behaviours.

Temporal logic . . .

. . . and a translation/axiomatisation of Fran.

Verifying the lift simulation.

Wrinkles . . . and further work.

Introducing Fran

Conal Elliott – Microsoft Research, Redmond.

Behaviours in *continuous* time, plus . . .

. . . *discrete* events.

Behaviours give animations . . .

. . . events support reactivity.

High-level, declarative, reactive programming.

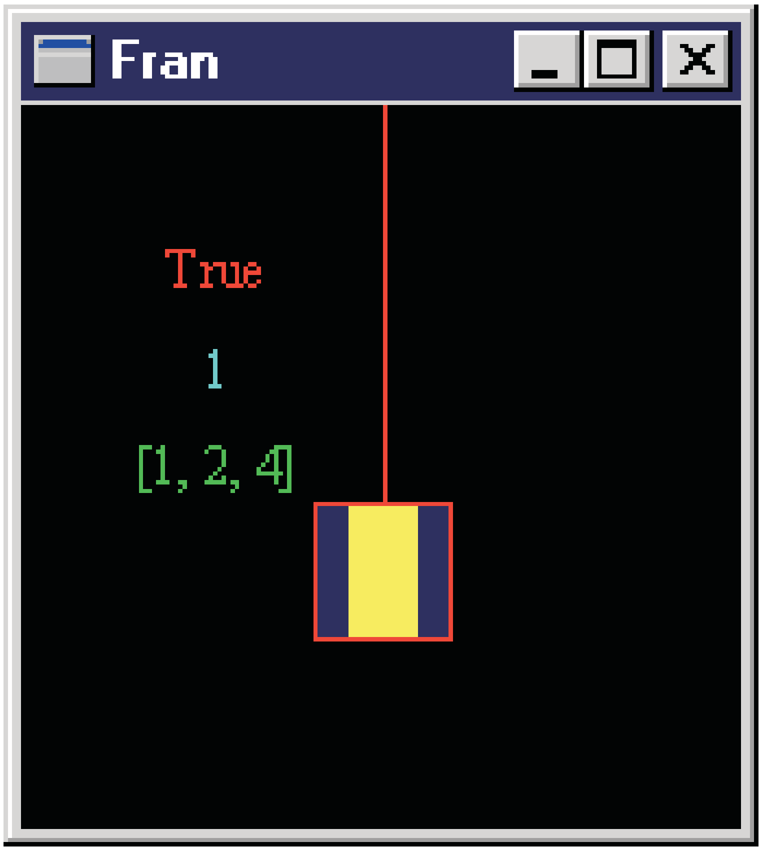
Contrast with temporal logic; verification?

Applications to multimedia: see Web page.

Lift controller

Traditional challenge problem in this field.

Functionality *plus* animation.



Two-floor lift simulation

Take a scenario where a lift moves between the `upper` and `lower` floors.

Calls to `goUp`/`goDown` are given by pressing the left/right mouse button.

A request to go up while the lift is going up has to be serviced by going down and then up again.

The animation consists of a moving static image . . . here a red blob.

Behaviours

Behaviours evolve in continuous time.

```
Time -> X
```

`X` can be a real number, a colour, an image, ...

Various combinators

```
moveXY    :: RealB -> RealB -> ImageB -> ImageB
over      :: ImageB -> ImageB -> ImageB
stretch   :: RealB -> ImageB -> ImageB
withColor :: ColorB -> ImageB -> ImageB
```

What is a typical Fran application like?

At top level an `ImageB` is produced and displayed.

Using `‘over‘` gives multiple components evolving concurrently . . .

. . . the components can depend on other behaviours which themselves evolve concurrently.

```
liftSim :: User -> ImageB

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

The `User` argument supplies interaction and timing information.

Controlling the vertical position

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

`atRate` gives `yPos` as the *integral* of the velocity, `dy`.

Modelling advantage: `dy` is piecewise constant.

`dy` is controlled by (external) interactions: look at the `Event` model.

Events

An `Event X` is a sequence of timed *occurrences*, each of which is associated with a value of type `X`:

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

Events can be *handled*, producing effects depending upon the timing, the value (and the residual event). Events can also be *merged*.

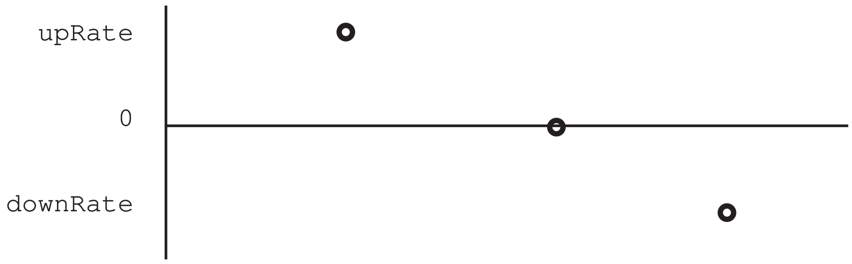
Simple handling only: convert values to a constant and merge . . .

```
setRate :: Event Double

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

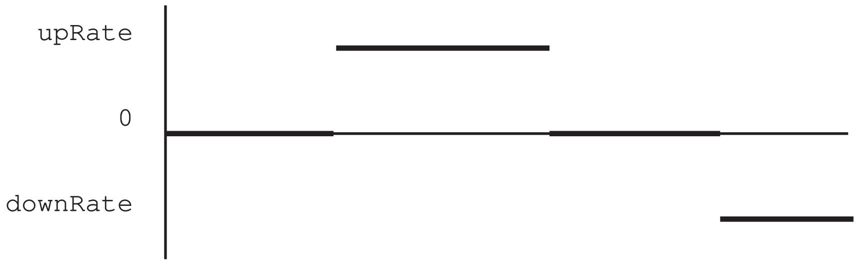
Events into behaviours

`setRate:`



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

`dy = stepper 0 setRate:`



In general,

`switcher :: GBehavior bv => bv -> Event bv -> bv`

The crucial movement events

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

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

`pending` is a Boolean behaviour ($\equiv$ state variable) recording that motion (up or down) is pending.

`arriveTop` is an internal event, generated when the lift arrives at the upper level.

`` `whenE` `` selects the event occurrences which take place at times when a Boolean behaviour is `True`.

Behaviours into events: `predicate`

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

The event occurs at (start points) of times when the `BoolB` – a time dependent predicate – is `True`.

Arriving at the top?

```
yPos ==* constantB upper
yPos >=* constantB upper
```

```
arriveTop
  = predicate (yPos >=* constantB upper &&*
               dy >* 0) u
```

State variables: recording pending journeys

The state is recorded as a collection of behaviours.

```
pending, upPending, downPending :: BoolB

pending = upPending ||* downPending

upPending = stepper False (setUp .|. resetUp)

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

Contrast this with a monadic approach.

The rest of Fran . . .

This example has covered some of the essentials of Fran . . .

. . . events and behaviours,

. . . moving between the two,

. . . concurrency,

. . . state variables.

It contains other more fundamental mechanisms which are used to implement some of the idioms seen here.

Verification?

Need first a clear view of the *implementation*, and thus its semantics

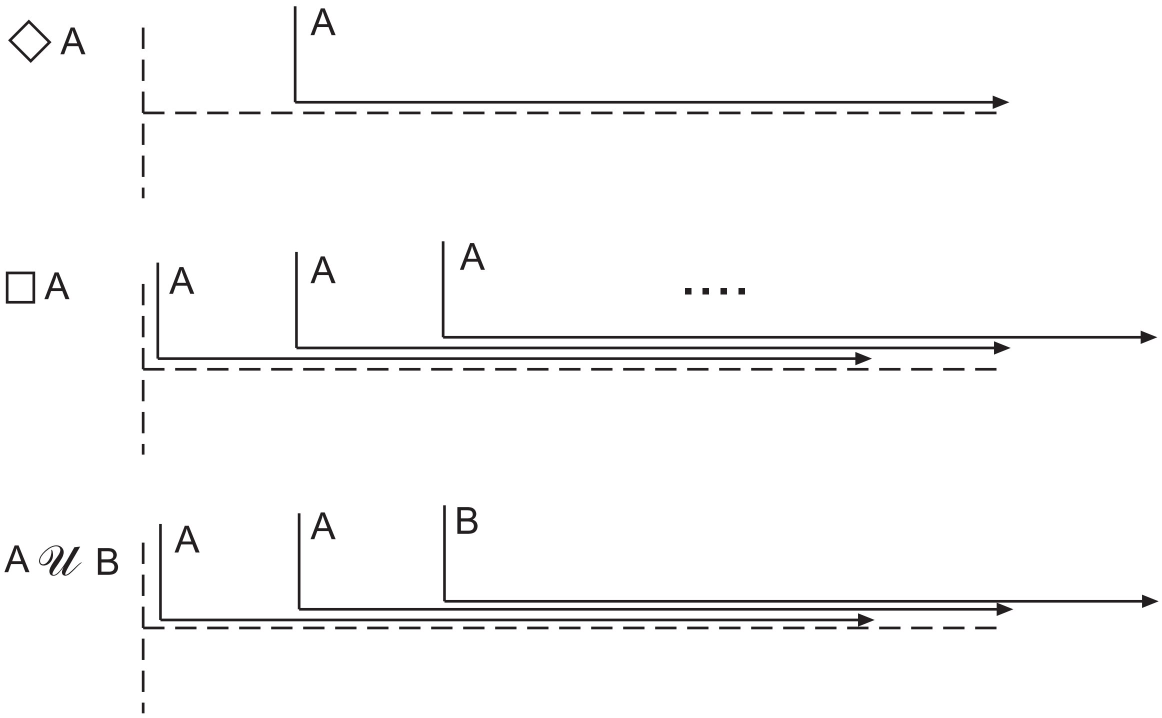
Naive approach works by *sampling* with finite-length intervals.

The current implementation works by *optimising* this in a number of ways.

Model an *idealised* Fran, where the finiteness of the sampling is elided.

Temporal logic

Formulas are interpreted over (semi-infinite) intervals of time.



Propositional connectives are interpreted pointwise.

Fran into temporal logic

Behaviours become *variables*.

Boolean behaviours are coerced into *formulas*.

What about events? In this model take events also into *primitive formulas*, true at exactly those points at which the events occur.

Interpreting Fran constructs

‘Event merge’ operation, `.|.`, is seen as logical disjunction.

The function `whenE` is interpreted as logical conjunction.

$$
\begin{aligned}
\texttt{goDown} \quad = \; & (\texttt{arriveTop} \wedge \texttt{pending}) \\
& \vee \\
& (\texttt{eitherButton} \wedge \texttt{waitingTop})
\end{aligned}
$$

Therefore, `.|.` and `whenE` distribute over each other.

Under certain circumstances `predicate` is the identity. Occurs when the conditions are transitory.

```
arriveTop = (at Top &&* dy >*y)
```

Identify the non-lifted and lifted forms of connectives and other operators, in line with standard temporal logic.

Stepper behaviours

A behaviour given by a stepper can be described using *weak* ‘until’.

```
upPending :: Behavior Bool

upPending = stepper False (setUp .|. resetUp)

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

$\Box$ (upButton $\Rightarrow$ (upPending $\mathcal{U}^0$ goUp))
$\Box$ (goUp $\Rightarrow$ (notB upPending $\mathcal{U}^0$ upButton))

The only remaining problem is `atRate` ...

Axioms for integrals

The important property in the example here is that 'the integral of a function bounded away from zero grows arbitrarily large'.

In the context of the example, relate `yPos` and `dy`:

$$\Box\ \texttt{(dy >=* k)} \Rightarrow \Diamond\ \texttt{(yPos >=* v)} \qquad \texttt{(int.1)}$$

$$\Box\ \texttt{(dy <=* -k)} \Rightarrow \Diamond\ \texttt{(yPos <=* v)} \qquad \texttt{(int.2)}$$

for all `v` and positive `k`.

Can be strengthened slightly to

$$\Box\ \texttt{(dy >=* k)} \Rightarrow \Diamond\ \texttt{((yPos >=* v)} \wedge \texttt{(dy >* 0))} \qquad \texttt{(dy.4)}$$

Time-bounded axioms

If the velocity is also bounded above, then can give a *lower* bound on the time taken to grow to a particular size:

For all positive `k`, `m` and all `v` there exists a positive `t` so that

$$\Box\ (\mathtt{m >=* dy >=* k}) \Rightarrow \Diamond_{\geq \mathtt{t}}\ (\mathtt{yPos >=* v}) \qquad (\mathtt{int.3})$$

$$\Box\ (\mathtt{-m <=* dy <=* -k}) \Rightarrow \Diamond_{\geq \mathtt{t}}\ (\mathtt{yPos <=* v}) \qquad (\mathtt{int.3})$$

The bounds are needed in proving certain properties by inductive arguments.

Invariants

Certain *invariant* properties hold throughout an execution.

$$\Box\ (\texttt{lower <=* yPos <=* upper}) \qquad \texttt{(I1)}$$

$$\Box\ (\texttt{dy ==* upRate} \lor \texttt{dy ==* downRate} \lor \texttt{dy ==* 0}) \qquad \texttt{(I2)}$$

$$\Box\ (\texttt{dy ==* 0} \Rightarrow (\texttt{y ==* upper} \lor \texttt{y ==* lower})) \qquad \texttt{(I3)}$$

How are they proved?

Proving invariant properties

From

$\Box\ ((\mathtt{A} \wedge \mathtt{P}) \Rightarrow \mathtt{P}\ \mathcal{U}_{\geq k}\ \mathtt{A})$

where `k` is a positive constant, we can infer the conclusion

$\Box\ ((\mathtt{A} \wedge \mathtt{P}) \Rightarrow \Box\ \mathtt{P})$

The bound is required so that the infinite sequence of intervals over which `P` holds exhausts an infinite amount of time.

(No Zeno sequences, in other words.)

Proving the boundedness invariant

To prove

$$\Box\ (\texttt{lower <=* yPos <=* upper}) \qquad \text{(I1)}$$

use

$$\Box\ (\texttt{goUp} \wedge (\texttt{lower <=* yPos <=* upper}) \Rightarrow ((\texttt{lower <=* yPos <=* upper})\ \mathcal{U}\ (\texttt{stop} \vee \texttt{goDown})))$$

$$\Box\ (\texttt{stop} \wedge (\texttt{lower <=* yPos <=* upper}) \Rightarrow ((\texttt{lower <=* yPos <=* upper})\ \mathcal{U}^0\ (\texttt{goDown} \vee \texttt{goUp})))$$

which are themselves proved using the characterisation of `dy`:

$$\Box\ (\texttt{goUp} \Rightarrow (\texttt{dy ==* upRate}\ \mathcal{U}^0\ (\texttt{stop} \vee \texttt{goDown})))$$
$$\Box\ (\texttt{stop} \Rightarrow (\texttt{dy ==* 0}\ \mathcal{U}^0\ (\texttt{goDown} \vee \texttt{goUp})))$$

and the axiomatisation of the integral definition of `yPos` from `dy`.

The correctness property

‘Every journey requested eventually happens.’

$\Box(\texttt{upButton} \Rightarrow \Diamond(\texttt{atBottom} \wedge \Diamond\texttt{arriveTop}))$
$\Box(\texttt{downButton} \Rightarrow \Diamond(\texttt{atTop} \wedge \Diamond\texttt{arriveBottom}))$

Proof: using the invariant

$$\begin{aligned}\Box\ (&\texttt{dy ==* upRate} \vee \\ &\texttt{dy ==* downRate} \vee \\ &\texttt{waitingBottom} \vee \\ &\texttt{waitingTop})\end{aligned}$$

which is consequence of the earlier invariants, do a case split at the point at which the button is pressed. An analysis of each shows that the lift will then get to the bottom and subsequently arrive at the top.

A wrinkle in idealised Fran

On arriving at the top, time `t`, the velocity is *immediately* set to `0` or `downRate`.

At what time does the velocity change?

At time `t`?

... *Inconsistent: implies* `upRate <= 0`.

At some time soon after `t`?

... *Not immediate: finite sampling; lose idealised model.*

At all times soon after `t`?

... *No first point – when does* `(dy <=* 0)` *happen?*

... ?

An alternative

. . . at $\mathtt{t}+\delta\mathtt{t}$ where $\delta\mathtt{t}$ is a positive *infinitesimal*?

The last solution has the attraction that it is an ideal model in which a finite sampling rate becomes infinite.

It models infinitely rapid response without affecting causality: $\mathtt{t}+\delta\mathtt{t}$ is after `t`, even if by an infinitesimal amount.

To investigate further . . .

Conclusion

Fran is a powerful tool for building declarative reactive animations.

Parts of it can be given an intuitive model by translation into a temporal logic. The full system presents a substantial challenge.

Formalise the logic fully . . . including the wrinkles.