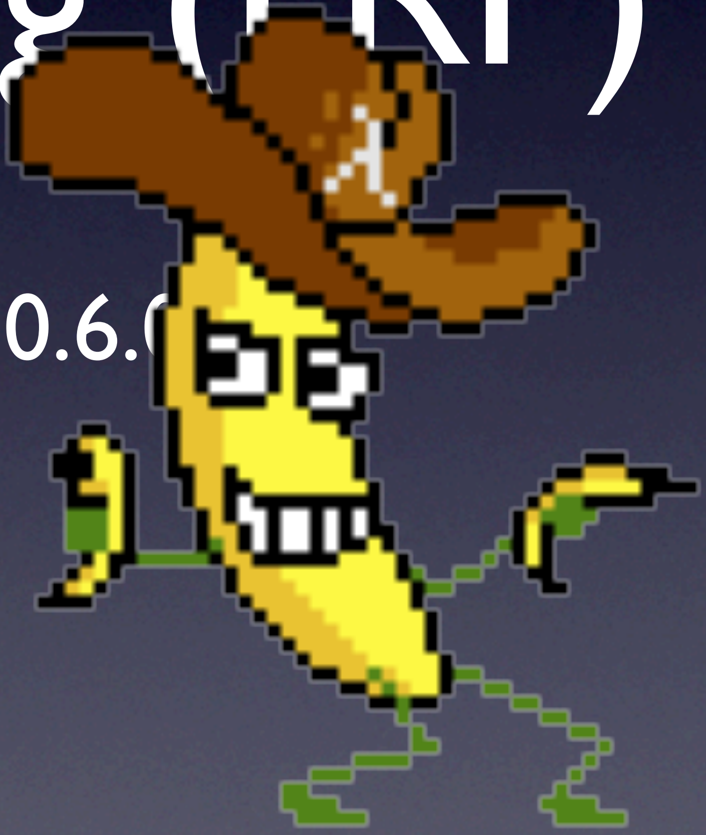
Functional Reactive Programming (FRP)

with reactive-banana-0.6.0.0

Heinrich Apfelmus

Functional Reactive Programming (FRP)

with reactive-banana-0.6.



Heinrich Apfelmus

Why?

Functional reactive programming is an elegant method for implementing interactive programs

- graphical user interfaces (GUI)
- animations
- digital music
- robotics

How?

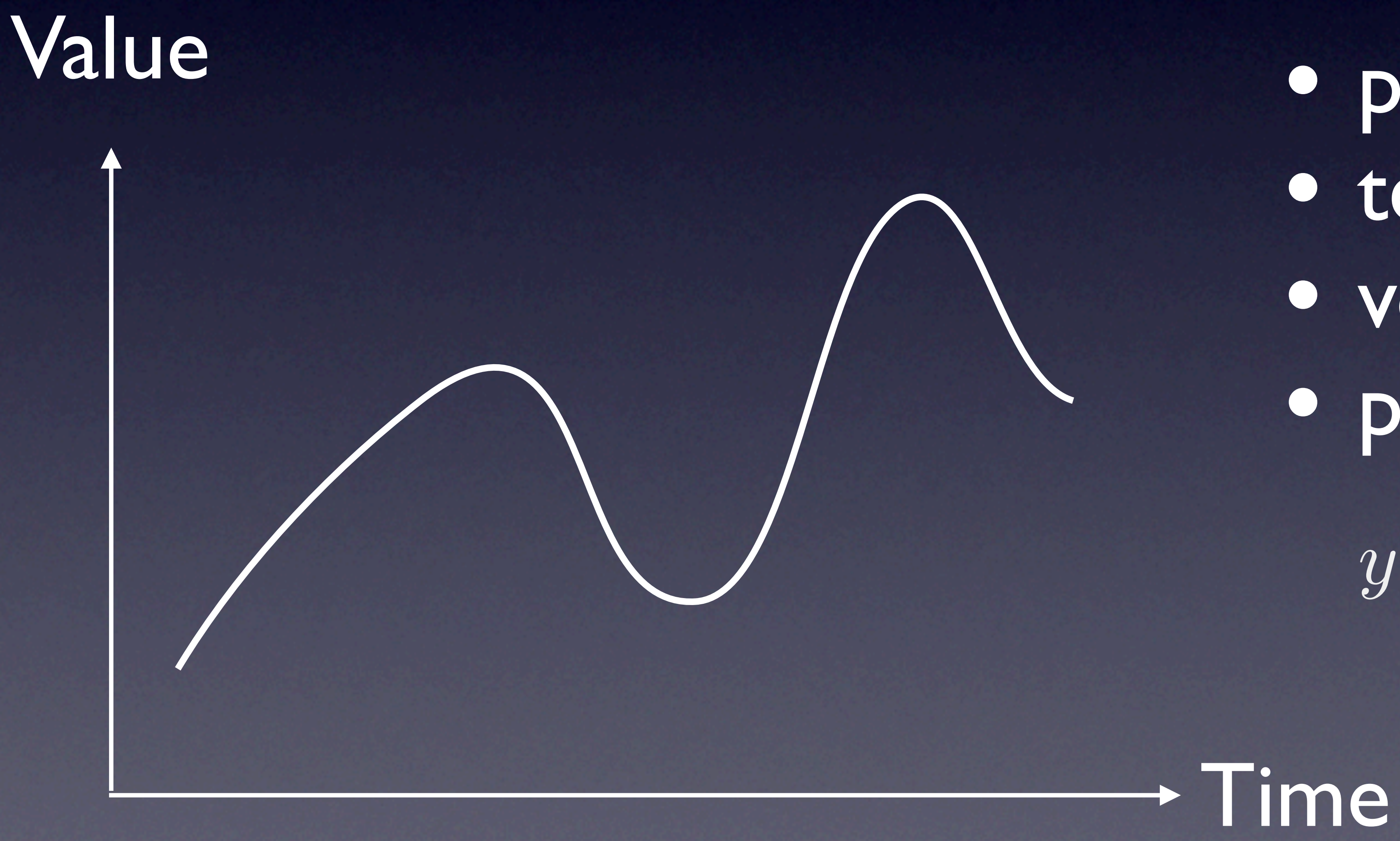
variation in time as first-class value

```
type Behavior a = Time → a
type Event a    = [(Time, a)]
```

The key data types are Behavior and Event.
Behavior corresponds to a „value that varies in time“.
Event corresponds to „events that occurr at certains points in time“.
I‘m going to explain how to understand them. Of course, the real implementation is abstract.

Behavior

```
type Behavior a = Time → a
```



- position – animation
- text value – GUI
- volume – music
- physical quantity

$$y(t) = y_0 + v_0 t - g\frac{t^2}{2}$$

A Behavior associates a value to each point in time.

Behavior API

```
instance Functor Behavior
```

Functor

```
instance Applicative Behavior
```

Applicative

How to program with Behaviors?
The API for Behaviors is actually very simple: they are just applicative functors.

Behavior API

```
(<$>) :: (a -> b)
      -> Behavior a -> Behavior b
```

Functor

```
pure  :: a -> Behavior a
(<*>) :: Behavior (a -> b)
      -> Behavior a -> Behavior b
```

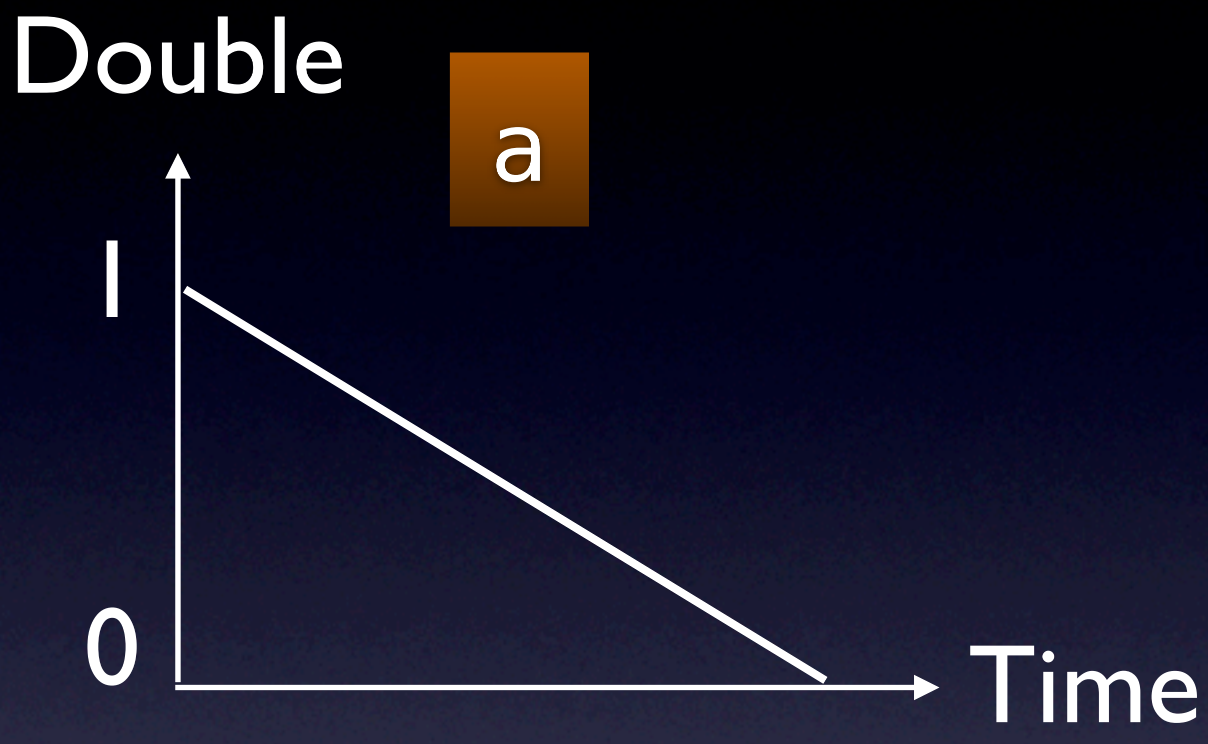
Applicative

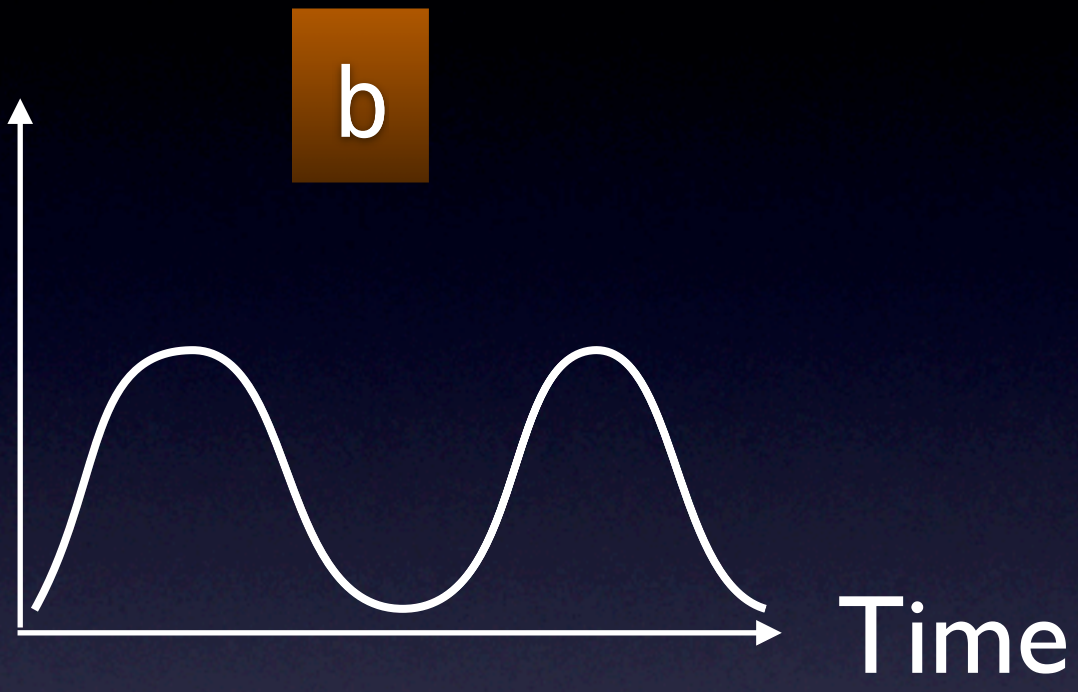
```
bf <*> bx =
   \time -> bf time $ bx time
```

at each
point in time

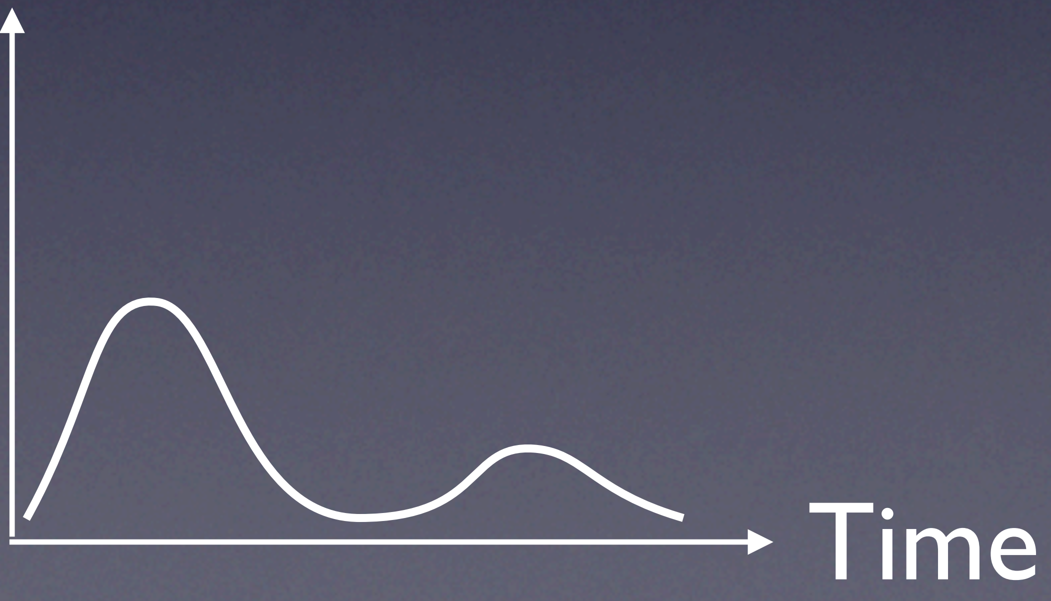
Reminder on the functions associated with Functor and Applicative classes.
The most important function is the <*> operator, which is called „apply“ and applies a time-varying function to a time-varying value, simply by applying them at each point in time.
The `pure` function constructs a value that stays constant in time.

Behavior API





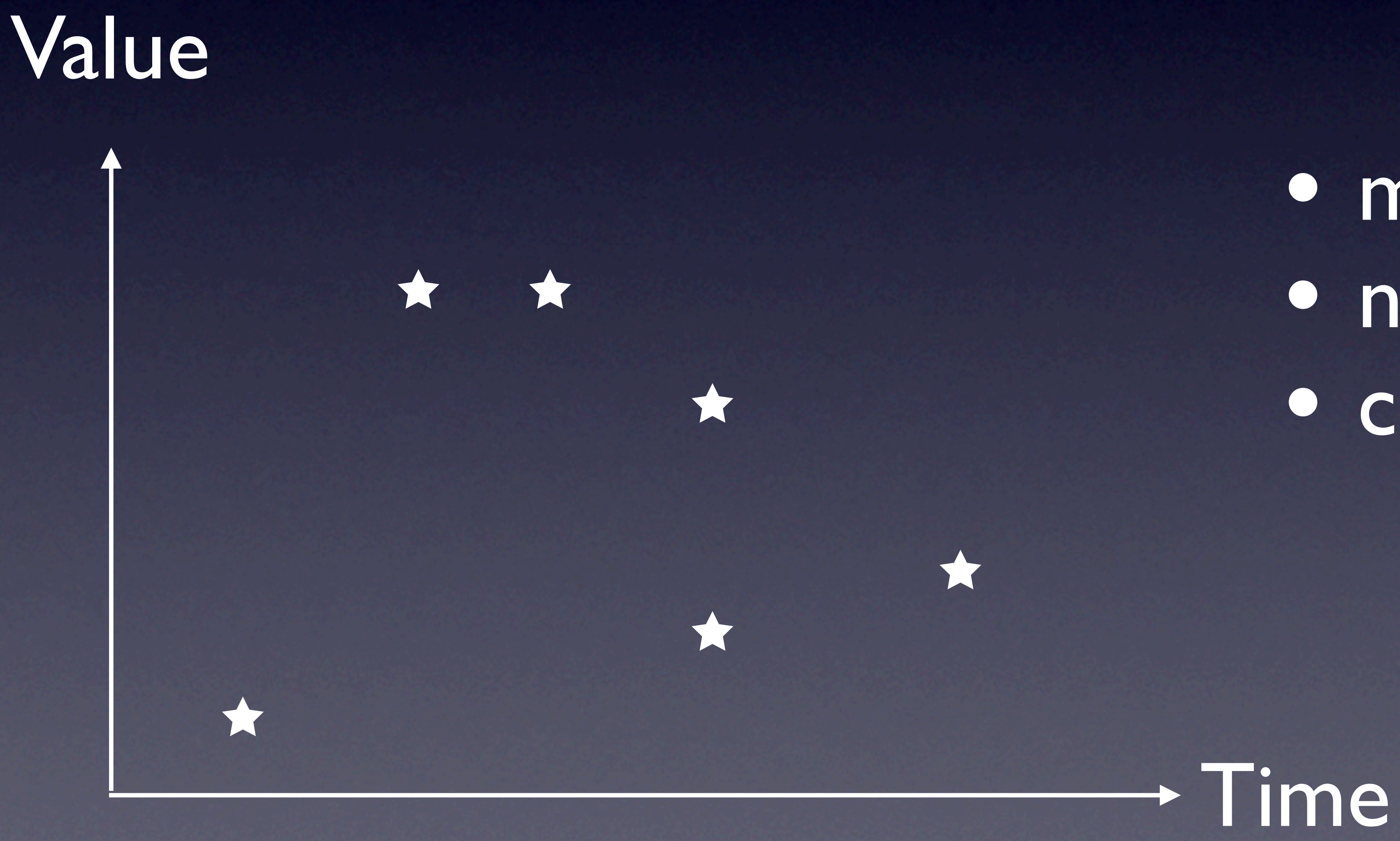
```
(*) <$> a <*> b
```



Example task: attenuate an oscillation.

Event

```
type Event a = [(Time,a)]
```



- mous clicks – GUI
- notes – music
- collision – physics

An Event is a collection of values that „occur“ at particular points in time.

You can also see that event occurrences may happen simultaneously, at least in reactive-banana-0.6.

Event API

```
instance Functor Event
```

```
never     :: Event a
unionWith :: (a -> a -> a)
 -> Event a -> Event a -> Event a

filterE   :: (a -> Bool)
          -> Event a -> Event a

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

Functor

List

[]

zipWith

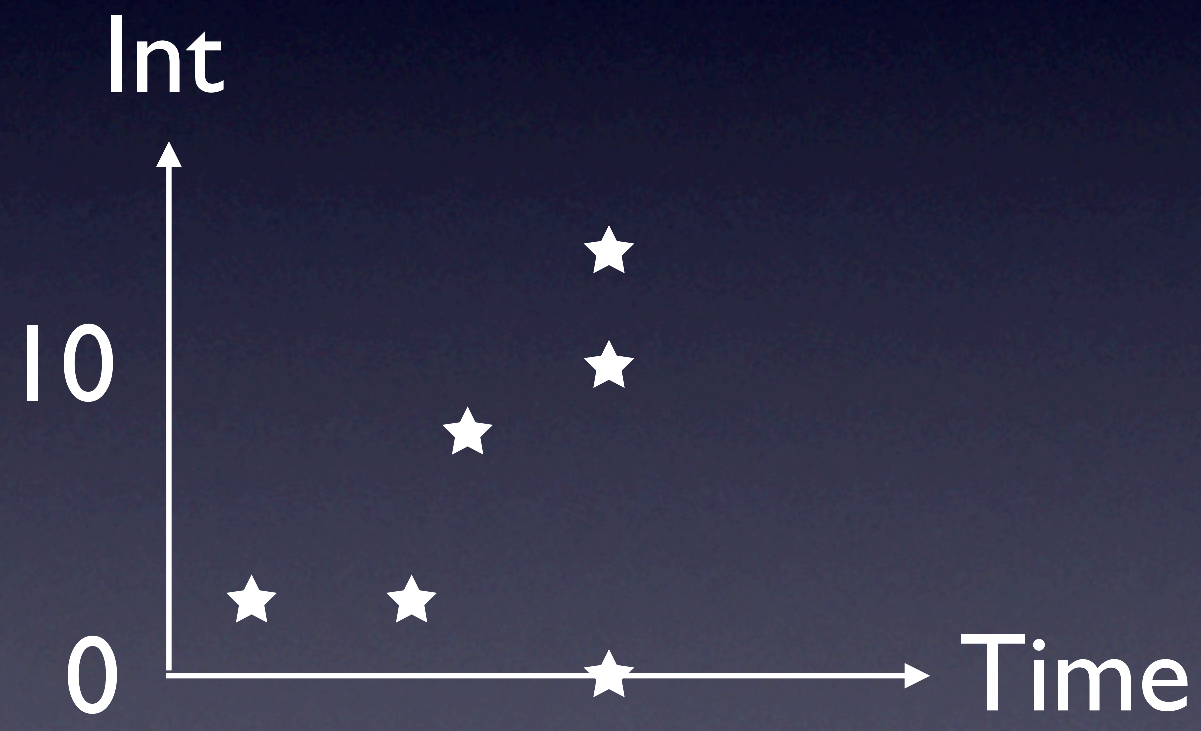
filter

scanl

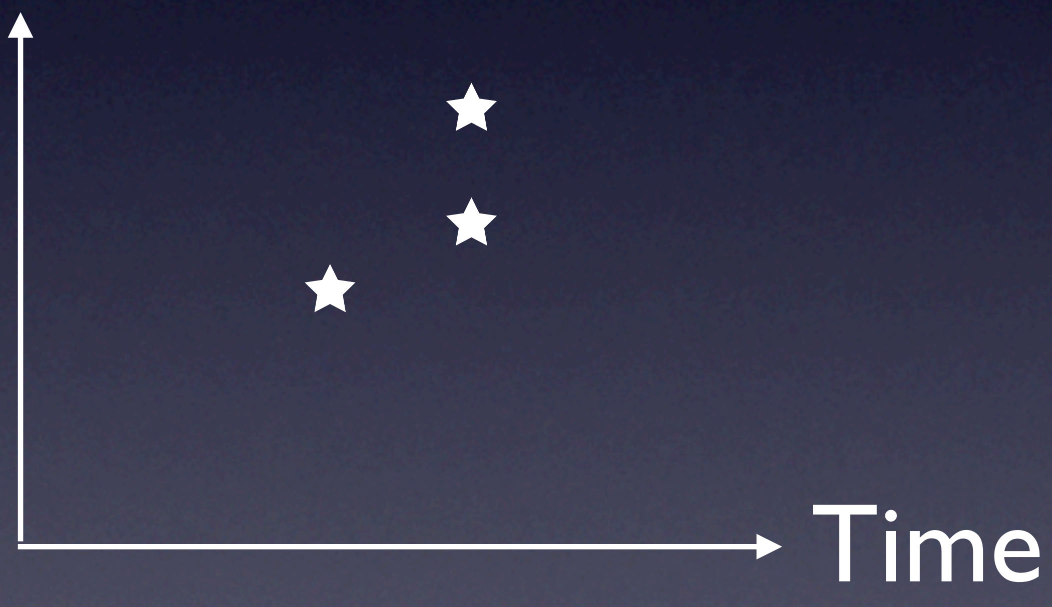
How to program with Events?
The API for Events is a bit more elaborate, but is closely related to operations on lists.

Event API

```
x
```



```
filterE (> 5) x
```

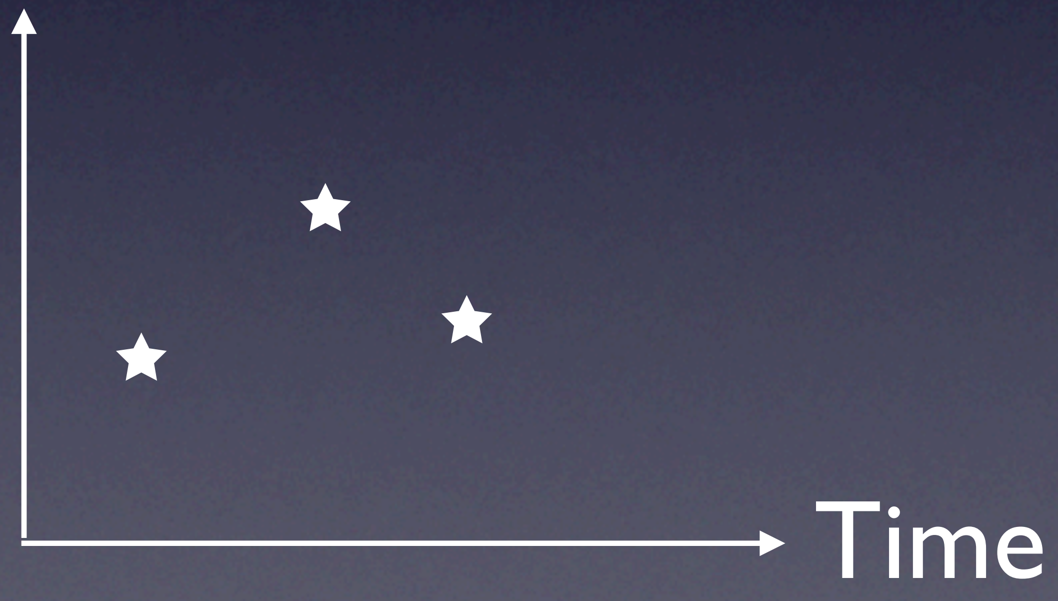


Example: filterE

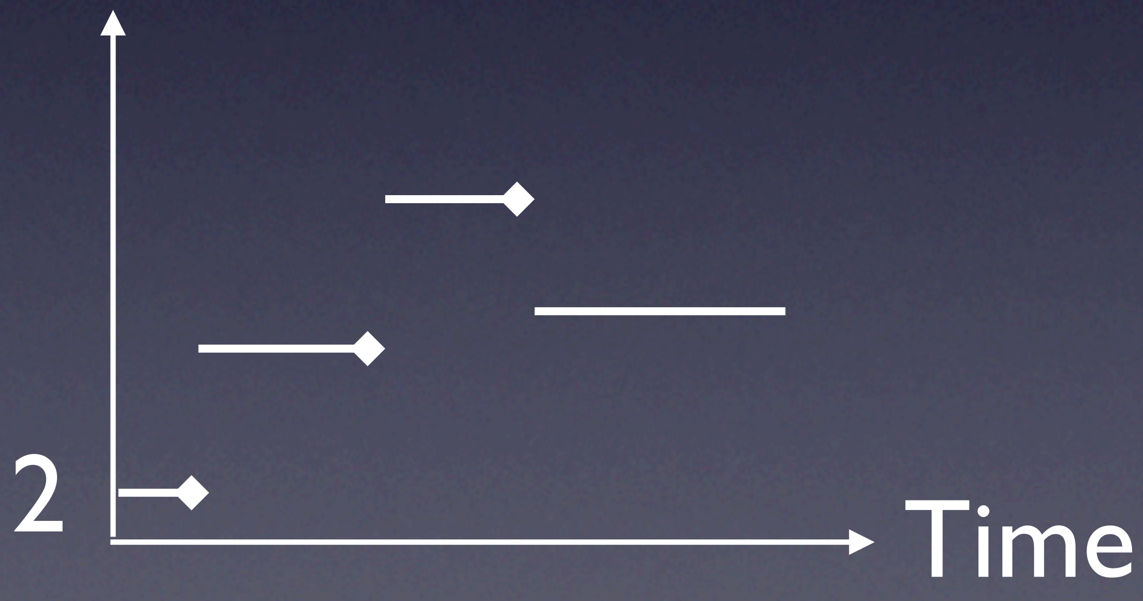
Event & Behavior API

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

```
x
```



```
stepper 2 x
```



Of course, the most interesting part about the API concerns the interaction between Behavior and Event.
The `stepper` function turns an Event into a Behavior by remembering the value. The result is a step function, hence the name.

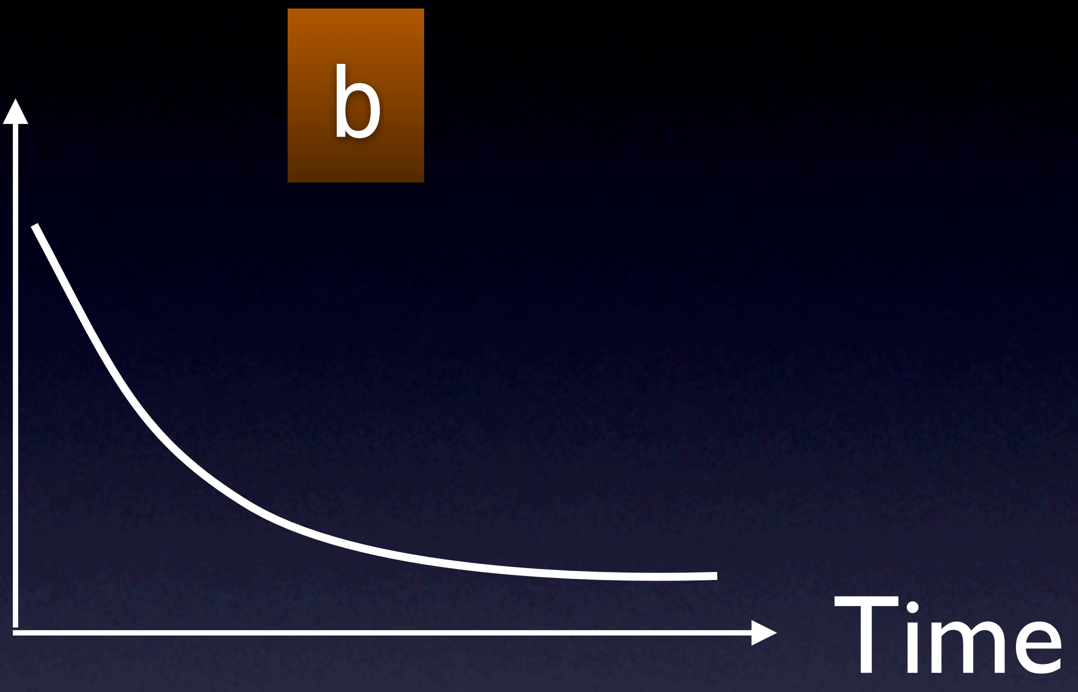
Event & Behavior API

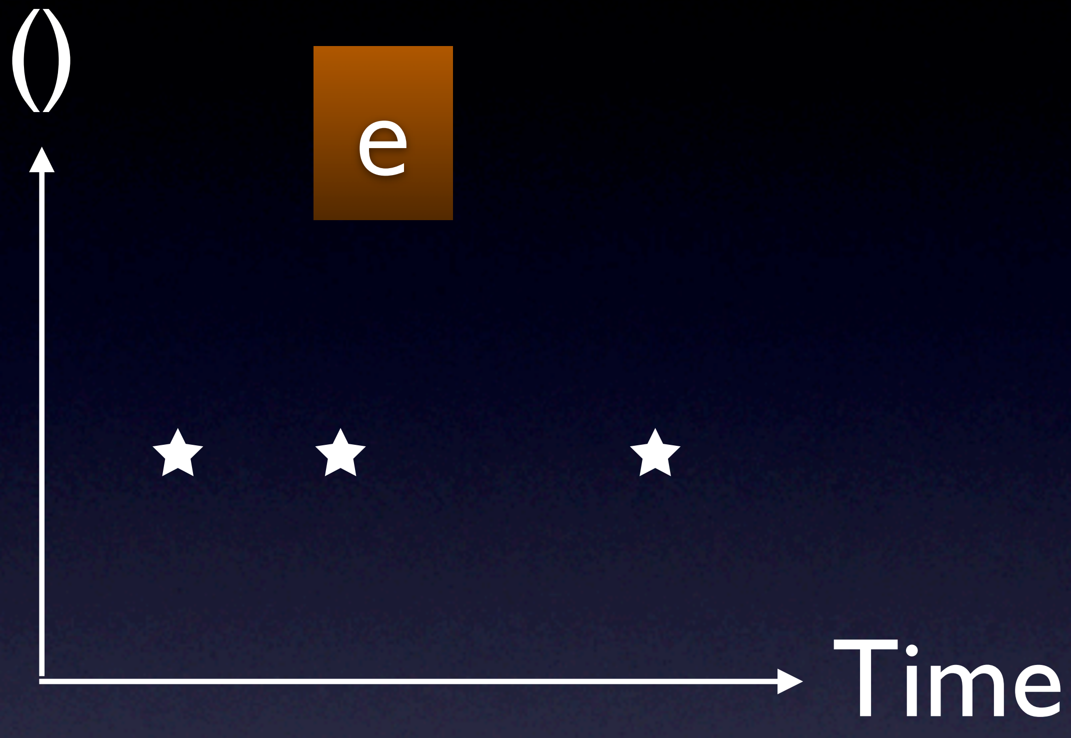
```
(<@>) :: Behavior (a -> b)
      -> Event a -> Event b

(<@)  :: Behavior b
      -> Event a -> Event b
```

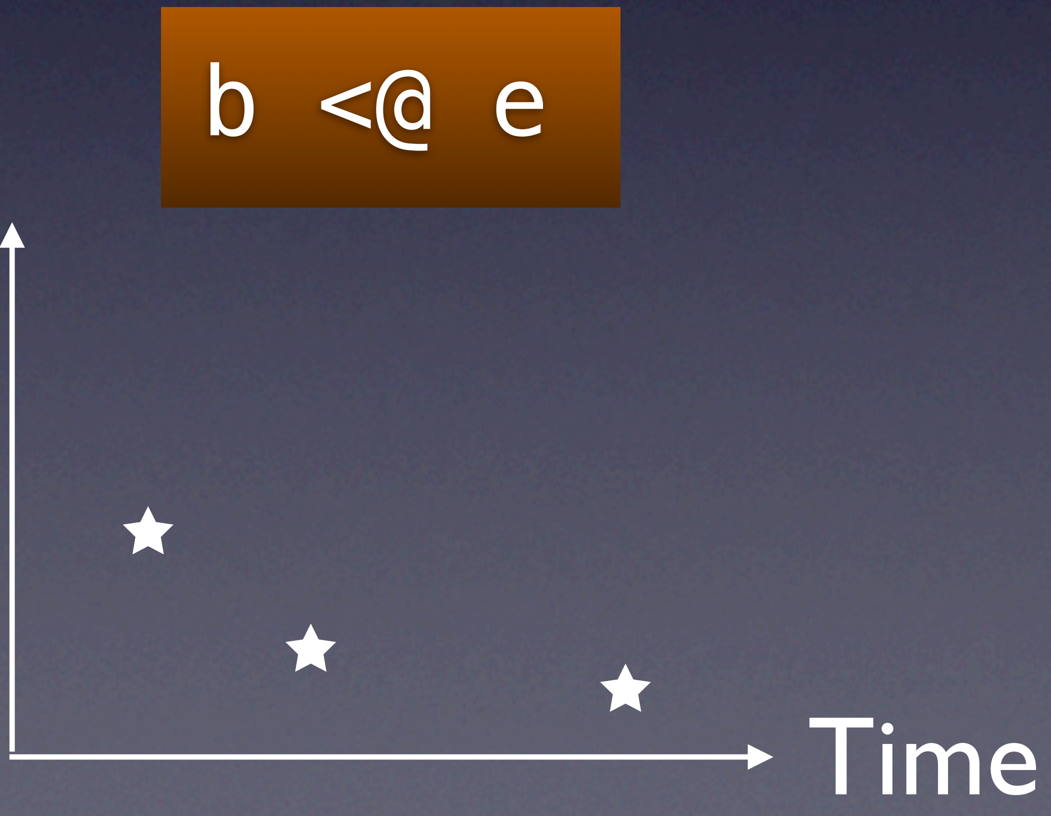
„apply“

The <@> operator is called „apply“ and applies a time-varying function to event occurrences.
Its little brother <@ tags an Event with values from the Behavior. It is analogous to the <$ operator from Data.Functor.

Event & Behavior API







Visualization of the <@ operator.

Frameworks (GUI, ...)

`data NetworkDescription t a`

`fromAddHandler`	import Event
`fromPoll`	import Behavior
`reactimate`	export Event
`changes`	get Event from Behavior

The API discussed so far allows you to combine existing Events and Behaviors into new ones, but it doesn't tell you how to get them in the first place. For this, you have to bind to external frameworks like wxHaskell. The NetworkDescription monad from the module Reactive.Banana.Frameworks allows you to do this. It's not a very interesting monad, it's just a device for bookkeeping and I recommend that you think of it as some sort of syntactic sugar.