IrrHaskell User Manual

Ewen Cochran
School of Informatics
University of Edinburgh

2011

Types

Behavior

One of the fundamental types of functional reactive programming. Treats time as a continuous and increasing input to produce a stream of values.

BehaviorValues

A record of Behaviors which can be applied to a NodePtr object. Constructor is as follows:

Beh {position :: Behavior Position, rotation :: Behavior Rotation, visibility :: Behavior Visible}

Display

Defines the information needed to display text on the GUI of the game. Constructed as the triple - (String, String, Bool) - where the values correspond to:

(text to display, font, show background)

Event

A fundamental type of functional reactive programming. Treats time as an increasing input to produce a series of time-value pairs where the values are either *Just var* or *Nothing* depending on whether the Event has happened or not.

MeshPtr

A type representing the pointer to a mesh stored within C++. Use this to refer to a mesh within the game.

Model

Holds the details and information of what a model should contain. Contructed as a 4-tuple of (String, String, Position, Behavior Values) where the values correspond to:

(path to object mesh, path to object texture, initial position of the object, Behaviors to apply to the object)

NodePtr

A type representing the pointer to a node stored within C++. Use this to refer to a node within the game.

Position

Refers to a 3D coordinate within the game environment. Equivalent to (Int, Int, Int) where the values represent the x, y, and z coordinates of the object.

Rotation

Refers to a 3D coordinate within the game environment. Equivalent to (Int, Int, Int) where the values represent the x, y, and z coordinates the object should face.

Time

An equivalence of Haskell's POSIXTime (Data.Time.Clock.POSIX).

Visible

An equivalence of the Bool type in Haskell. Values are True or False depending on the visibility wanted.

Functions

```
(.|.) :: Event a -> Event a -> Event a
(.&.) :: Event () -> Event () -> Event ()
```

Similar to the Boolean operators (||) and (&&). If either Event reads a *Just* value then (.|.) will return a *Just* value. If both Events read a *Just* value then (.&.) will return a *Just* value.

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

The function $(=>>)$ receives data from an Event. This data is then used to produce an Event of a different type. $(->>)$ is a special case of $(=>>)$ in which the data is not received from the inital Event.

```
(>*), (<*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(==*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(>=*), (<=*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
(&&*), (||*) :: Ord a => Behavior a -> Behavior a -> Behavior Bool
```

Ordinal and logical operators for use with Behaviors over ordinal data types. Produces a Behavior Bool which can then be used to create Events.

```
behRecord :: BehaviorValues
```

The default record of Behaviors.

```
createExtrudedMesh :: [ (Int,Position) ] -> Position -> IO MeshPtr
```

Allows the creation of customised polygon objects in the game based on a list of vertices. Returns a MeshPtr which can be used to created a NodePtr for use within the game.

```
distanceBetweenBeh :: Behavior Position -> Behavior Position ->
    Behavior Float
```

Takes two Behaviors over Position and returns a Behavior Float where the value at each time value is the distance between the two positions at the same time.

```
eventOr :: [ Event a ] -> Event a
eventAnd :: [ Event () ] -> Event ()
```

Operate over a list Events to produce a single Event. eventOr adds *Just* to the output Event when one or more Events in the input have a *Just* value. eventAnd adds *Just* to the output Event when all of the Events in the input have a *Just* value.

```
fstB :: Behavior (a,b) -> Behavior a
sndB :: Behavior (a,b) -> Behavior b
```

Returns a Behavior over the first, or second, value of a Behavior over a tuple.

```
fstTriple :: (a,b,c) -> a
sndTriple :: (a,b,c) -> b
trdTriple :: (a,b,c) -> c
```

Similar to the fst and snd functions defined in the Haskell Prelude, except for use with triples. This included as triple are a common feature within the library.

```
key :: Event Char
lbp, sample :: Event ()
collision :: Behavior Float -> [ NodePtr ] -> [ NodePtr ] -> Event
    ()
```

Standard Events within the library. key and lbp denote key presses and left mouse button presses respectively. The collision event shows *Just* when objects from two groups pass within a distance threshold of each other.

```
lift0, constB :: a -> Behavior a
lift1 :: (a -> b) -> (Behavior a -> Behavior b)
```

Lifting functions to turn standard types into Behaviors over that type. The constB and lift0 functions take a static value and yield a Behavior over that value. lift1 applies a function, taking one value and returning another, to the values of a Behavior to produce a new Behavior. There are also functions called lift2, lift3, lift4, and lift5 available.

These operate in the same way as lift1, except for functions of increasing numbers of parameters.

```
loadAmbientLight :: (Int,Int,Int,Int) -> IO ()
```

Places an ambient light into the game environment with the (r,g,b,alpha) value passed into the function.

```
loadCharacters :: [ Model ] -> IO [ NodePtr ]
loadMeshes :: [ (MeshPtr,String,Position,BehaviorValues) ] -> IO [
    NodePtr]
```

The loadCharacters function is used to load in a list of pre-made character models into the game. The loadMeshes function can be used to retrieve a list of NodePtrs based upon the MeshPtrs create by createExtrudedMesh.

```
loadStandardCamera, loadFPSCamera :: Position -> Position -> IO
    NodePtr
```

Creates either a standard, fixed position, camera or a controllable first-person shooter style camera. Returns a NodePtr for use and reference throughout the game.

```
maximumB, minimumB :: [ Behavior Int ] -> Int -> Behavior Int
```

Used to create a Behavior which, at every time sample, shows the maximum or minimum value of a list of Behaviors.

```
nodePosition :: NodePtr -> Behavior Position
nodeVisibility :: NodePtr -> Behavior Int
```

Creates a Behavior over the position, or visibility, of a given node. Note that nodeVisiblitity shows a 1 value if the node is visible and a 0 value if it is not.

```
pairB :: Behavior a -> Behavior b -> Behavior (a,b)
```

Takes two Behaviors and produces a Behavior over a the tuple of their values. Functions tripleB, quadB, and pentB are also available.

```
randomB :: Behavior Int
```

Produces a stream of random Integer values. These can then be used to trigger random Events.

```
reactimate ::
    Behavior Position -> Behavior Display -> [ (NodePtr,
       BehaviorValues) ] -> IO ()
```

The primary driver for the game engine. Pass in the Behavior over the camera's Position, a Behavior over the Display (used for the GUI text), and a list of NodePtr-BehaviorValues pairs (used to define every object in the game as well as their behavior).

```
setUp :: String -> Model -> IO NodePtr -> IO () -> IO NodePtr
```

A single function to define and initialise the game environment. Pass in the window caption, map model, the appropriate camera function, and a light function. The camera node is returned.

```
step :: a -> Event a -> Behavior a
stepAccum :: a -> Event (a -> a) -> Behavior a
```

These are based on the switch function. When the Event occurs, the step function changes the value of the Behavior to the value associated with the Event. In stepAccum, the functiona ssociated with the Event is applied to the last value of the Behavior to produce a new Behavior.

```
time :: Behavior Time
```

A Behavior where the value at each time step is the current time. This can be used when creating Behaviors which are dependent on the time.

```
undefinedBehavior :: Behavior a
```

A Behavior where, at every time sample, the value *undefined* is produced.

```
undefinedModel :: Model
```

A default Model value where each value is set as *undefined*. Useful for cases where no Model is required.

```
untilB, switch :: Behavior a -> Event b -> Behavior a
```

These are functions for linking Behaviors. Behaves as one Behavior until a defined Event occurs before switching to a new Behavior. The switch function will recursively process the Event to continually change Behavior.

```
when, threshold, while :: Behavior Bool -> Event ()
```

These functions produce Events based off of Behaviors over Booleans. The functions when and threshold add *Just* to the output Event when the Behavior changes from False to True, while adds *Just* to the output Event whilst the Behavior is True.