EDAN40: Functional Programming
Functional Reactive Programming

Jacek Malec
Dept. of Computer Science, Lund University, Sweden

May 22nd, 2023

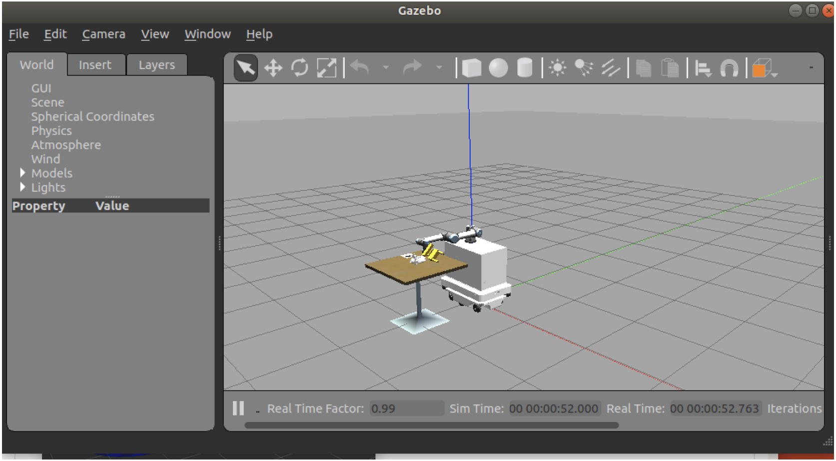
What is reactive programming

Lecture based on:

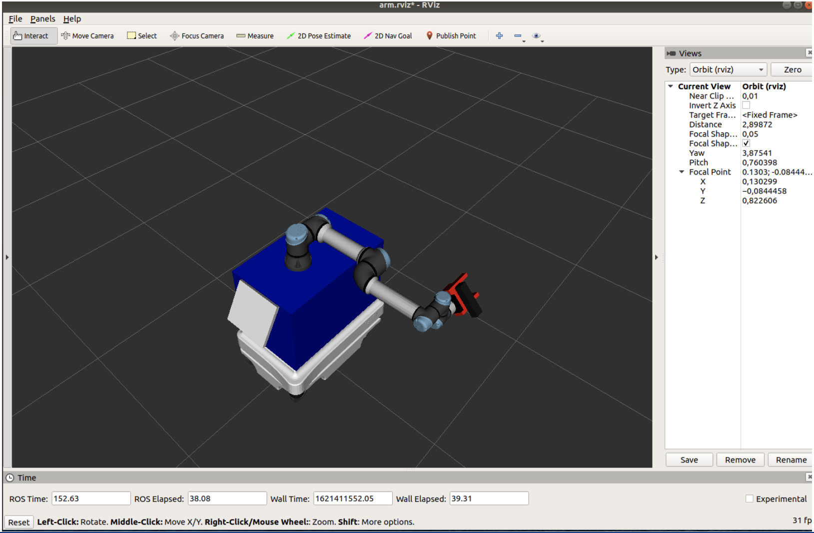
- http://www.haskell.org/haskellwiki/Functional_Reactive_Programming
- Edward Amsden: “A Survey of Functional Reactive Programming” (search for a video from September 2012)
- http://www.haskell.org/haskellwiki/Reactive-banana
- Conal Elliot’s (Yale, Yampa) slides on FRP
- https://github.com/gelisam/frp-zoo
- https://github.com/acowley/roshask.git

Heron

Heron + ROS


Gazebo
File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Atmosphere
Wind
Models
Lights
Property Value
Real Time Factor: 0.99
Sim Time: 00 00:00:52.000
Real Time: 00 00:00:52.763
Iterations

Heron + ROS


arm.rviz* - RViz
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
2D Nav Goal
Publish Point
Views
Type: Orbit (rviz)
Zero
Current View Orbit (rviz)
Near Clip ... 0,01
Invert Z Axis
Target Fra... <Fixed Frame>
Distance 2,89872
Focal Shap... 0,05
Focal Shap...
Yaw 3,87541
Pitch 0,760398
Focal Point 0.1303; -0.08444...
X 0,130299
Y -0,0844458
Z 0,822606
Save
Remove
Rename
Time
ROS Time: 152.63
ROS Elapsed: 38.08
Wall Time: 1621411552.05
Wall Elapsed: 39.31
Experimental
Reset
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
31 fp

ROS

- Robot Operating System
- middleware
- client-server
- publisher-subscriber (topics)
- streams (sometimes in real time)

ROS

- Robot Operating System
- middleware
- client-server
- publisher-subscriber (topics)
- streams (sometimes in real time)
- `roshask`

Basic Concepts of Reactivity

Reactivity $\equiv$ time-dependent responsiveness

1. behaviours (signals, fluents, streams) – functions of time
2. occurrences – elements in $Val \times Time$
3. events – sets of occurrences (lists in our case)

An interesting issue: continuous vs. discrete time

Basic Concepts of Reactivity

Reactivity $\equiv$ time-dependent responsiveness

1. behaviours (signals, fluents, streams) – functions of time
2. occurrences – elements in $Val \times Time$
3. events – sets of occurrences (lists in our case)

An interesting issue: continuous vs. discrete time

Approaches to reactivity:

- “embedding” (classical)
- signal-based
- n-ary FRP

Semantics vs. interpretation

Behaviours

```
newtype Behavior a =
    Behavior {
      at :: Time -> a
    }

myCityName :: Behavior String

myCityName ‘at‘ yesterday
```

Behaviours

```
type Behavior a = Time -> a     -- conceptually
type Time = Float

-- lifting many functions from a to Behavior a

timeTrans :: Behavior Time -> Behavior a
                                       -> Behavior a
timeTrans f ba t = ba (f t)

integral   :: Behavior Float -> Behavior Float
derivative :: Behavior Float -> Behavior Float
```

Events

```
type Event a = [(a, Time)]     -- conceptually
untilB :: Behavior a -> Event (Behavior a)
                                  -> Behavior a
switch :: Behavior a -> Event (Behavior a)
                                  -> Behavior a

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

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

Events, cntd

```
-- Snapshot events
snapshot_ :: Event a -> Behavior b -> Event b

-- Predicate event
when :: Behavior Bool -> Event ()

-- other
step   :: a -> Event a -> Behavior a
stepAccum :: a -> Event (a -> a) -> Behavior a
withElem_ :: Event a -> [b] -> Event b
```

A graphics library

`gloss` library

`paddle.hs`

Implementation issues

reactive banana library offers constructors:

```
filter :: (a -> Bool) -> Event a -> Event a
accumE :: a -> Event (a -> a) -> Event a
stepper :: a -> Event a -> Behavior a
apply :: Behavior (a -> b) -> Event a -> Event b

instance Functor Event
instance Functor Behavior
instance Applicative Behavior
instance Monoid (Event a)
```

Signal functions

Signal - primitive concept

SF - primitive type:

```
-- informally
SF a b = Signal a -> Signal b
```

and

```
-- informally again
Signal a = Time -> a
```

`Time` is considered to be real-valued.

Arrows

Arrow a b c represents a process that takes as input something of type b and outputs something of type c.

`arr` builds an arrow from a function:

```
arr :: (Arrow a) => (b -> c) -> a b c
```

Arrows are composed with `(>>>)`, while `first` and `second` create new arrows:

```
(>>>)  :: (Arrow a) => a b c -> a c d -> a b d
first  :: (Arrow a) => a b c -> a (b, d) (c, d)
second :: (Arrow a) => a b c -> a (d, b) (d, c)
```

Signal function primitives

Point-wise application:

```
arr :: (a -> b) -> SF a b
arr f = \s -> \t -> f (s t)
```

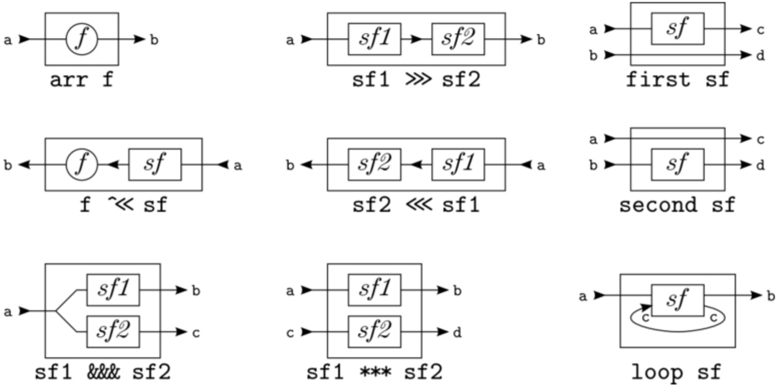
Signal composition:

```
(>>>) :: SF a b -> SF b c -> SF a c
sf1 >>> sf2 = \s -> \t -> (sf2 (sf1 s)) t = sf2 . sf1
```

Other compositions:

```
first :: SF a b -> SF (a, c) (b, c)
(&&&) :: SF a b -> SF a c -> SF a (b, c)
loop  :: SF (a, c) (b, c) -> SF a b
```

Signal function primitives



Signal functions, cont.

Doing something with it:

```
integral :: Fractional a => SF a a
```

is a *stateful primitive* (depends not only on t but maybe also on $[0, t]$).
The `integral` primitive computes the time integral of its input signal:

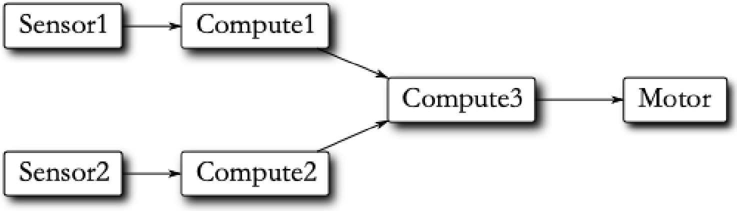
```
localTime :: SF a Time

localTime = const 1.0 >>> integral
```

Then we introduce events ... (for more see the AFRP papers).

Where to go from here?

- Arrows (a generalisation of monads)
  - Hughes@CTH (first paper on Arrows: 2000)
  - AFRP = Arrowized FRP (first paper on AFRP by Hudak et al.: 2002)
- Applicative functors (weaker than monads, no value passing)
- Various signal-functions-semantics implementations (see the survey paper)
- Actively developed libraries: Yampa (unary FRP a la Yale), reactive-banana (1.2.0.0 as of May 15th, 2018)
- Lots to do ...

So what about heron?




```
publish "t-cmd" (go $ interpolate fuse t1 t2)
```

roshask

Filtering a sensor value

1. identify threshold crossing
2. react to it

```
void handle_sensor(float val) {
  if(val > threshold) {
    act(val*0.1); } }
```

roshask

Filtering a sensor value

1. identify threshold crossing
2. react to it

```
void handle_sensor(float val) {
  if(val > threshold) {
    act(val*0.1); } }
```

When topics are first class objects:

```
subscribe "sense" >>=
  publish "cmd" . fmap act . filter (>threshold)
```

A couple of examples

A sliding window of given size, accumulating the values over `a`

```
slidingWindow :: (Monad m , Monoid a ) =>
                 Int -> Topic m a -> Topic m a
```

Averaging over n (10) values

```
avg :: Monad m => Topic m Float -> Topic m Float
avg = fmap (*0.1) . slidingWindow 10
```

How is it done?

A ROS topic is a step function yielding a value and the rest of the topic:

```
newtype T m a = T {unT :: m (a, T m a)}
```

where `m` is an additional type constructor.
Note that

```
instance Functor m => Functor (T m) where
  fmap f (T t) = T (fmap (f *** fmap f) t)
```

But be careful!

```
fmapT f (T t) = T (fmapm (f *** fmapT f) t)
```

`(***)` is coming from `Arrow` library.

roshask

```
telescope :: Node ()
telescope =
  advertise "video" $ (topicRate 60 (runTopicState images 0)

detectUFO :: Node ()
detectUFO =
  subscribe "video" >>= runHandler findPt >> return ()

main = runNode "NodeCompose" $ telescope >> detectUFO
```

The Heron example

```
class Sensor a where
  sensor :: Node (Topic IO a)
class Command a where
  command :: Topic IO a -> Node ()
class Controller a where
  controller :: a -> Node ()

instance Sensor Velocity where
  sensor = subscribe "odom"
    >>= return . fmap (_twist . _twist))
instance Command Velocity where
  command = publish "/mobile_base/commands/velocity"
instance (Sensor a,Command b) => Controller (a -> b) where
  controller f = sensor >>= command . fmap f
```

ROSY example

```
move :: Velocity
move = Velocity 0.5 0
main = simulate move

accelerate :: Velocity -> Velocity
accelerate (Velocity vl va) = Velocity (vl+0.5) va

play :: Bumper -> Maybe Sound
play (Bumper _ Pressed) = Just ErrorSound
play (Bumper _ Released) = Nothing
accelerateAndPlay = (accelerate,play)
```