CS4012 Topics in Functional Programming

Glenn Strong

Domain Specific Languages

- Use of libraries in progamming is ubiquitous.
- They can be a way of capturing styles of problem solving in some domain
- We can think of them as being like little programming languages
- Then there are Domain Specific Languages (DSLs)

This usually refers to a small, not general purpose language

- Captures (only) some specific problem domain
- For example: unix shell, make, SQL, Mathematica, VHDL, TEX, etc.
- Somone familiar with the domain should know the semantics already

- Programs are
  - concise, easy to write, easy to maintain
  - Easy to reason about
  - Something non-programmers can maintain!
- But
  - Language design is *hard*
  - People always want lots of features, good performance
  - We will end up with lots of languages
    - lots of lexers, parsers, type checkers, optimisers, interpreters, code generators, debuggers…

If we *embed* our DSL into Haskell we can have

- Powerful easy domain-specific expression
- Full Haskell expressiveness outside that domain

Two types of language embedding

- Shallow
  - Represent DSL programs as values in the host language (like, say, functions)
  - Provide a *fixed* semantics
  - A program in the DSL might consist of calls to library functions

- Deep
  - Represent DSL programs as values in the host language (still), but kept abstract
  - Use higher-order functions (combinators) to piece together programs
  - A program in the DSL might consist of construction of a value to describe the program that is fed to an interpreter

Be aware, the terminology is loose, the borders are a bit blurry.

Example: Geometric regions

- DARPA/ONR/Naval Surface Warfare Center project
- Geo-server tracks objects of interest
- Maintains notion of region
- Rules can trigger actions when objects and regions intersect

“Modular Domain Specific Languages and Tools”, Paul Hudak, 1998, IEEE Proceedings of Fifth International Conference on Software Reuse.

- Our primary abstraction in this domain is the *region*
- A *shallow* embedding captures the semantics directly

```haskell
data Vector a = Pt a a deriving (Eq, Show)
type Point = Vector Float

origin :: Point
origin = Pt 0 0

type Region = Point -> Bool
```

All we really need to know about a region is whether it contains a point:

```haskell
inRegion       :: Point -> Region -> Bool
p `inRegion` r  = r p
```

We can define some regions (given a few simple operations like `distance`):

```haskell
empty p = False

circle   :: Radius -> Region
circle r  = \p -> distance p < r*r

halfPlane     :: Point -> Point -> Region
halfPlane a b  = \p -> zcross (a-p) (b-a) > 0
                 where zcross (Pt x y) (Pt u v) = x*v - y*u
```

It gets interesting when we start taking advantage of the embedding of our region language in Haskell.

```haskell
outside :: Region -> Region
outside r = \p -> not (r p)

(/\)         :: Region -> Region -> Region
(r1 /\ r2) p  = r1 p && r2 p

(\/)    :: Region -> Region -> Region

intersect, union :: [Region] -> Region
```

```haskell
intersect         = foldr1 (/\)
union             = foldr1 (\/)
```

Primitives like `circle` place their regions at the origin; with a translation operation we can move them about:

```haskell
at       :: Region -> Point -> Region
r `at` p0 = \p -> r (p - p0)
```

Derived regions

```haskell
annulus      :: Radius -> Radius -> Region
annulus r1 r2 = outside (circle r1) /\ (circle r2)

convexPoly       :: [Point] -> Region
convexPoly (v:vs) = intersect (zipWith halfPlane ([v]++vs) (vs++[v]))
```

And so on.

We might want to know that, say, intersection is associative

$$(R_1 \cap R_2) \cap R_3 = R_1 \cap (R_2 \cap R_3)$$

That is, given our definition of intersection

```haskell
(r1 /\ r2) p  = r1 p && r2 p
```

we want to show

```haskell
(r1 /\ r2) /\ r3 = r1 /\ (r2 /\ r3)
```

By equational reasoning this is easy!

```haskell

  ((r1 /\ r2) /\ r3) p
= (r1 /\ r2) p && r3 p
= (r1 p && r2 p) && r3 p
= r1 p \&& (r2 p && r3 p)
= r1 p \&& (r2 /\ r3) p
= (r1 /\ (r2 /\ r3)) p
```

- Higher-order functions gave us a powerful embedding
- We can reason about "region problems" at the level of regions!
- We could even add rewrite rules to the compiler take advantage of properties we prove
- The properties could even become optimisations!

DSL's for the web

There is a mini-industry creating web frameworks for Haskell.

As an example (but also as a useful tool for the first project) we will look at some.

Probably the simplest framework is `scotty`

The Scotty web framework

Scotty is a “minimalistic” web framework for Haskell (inspired by Ruby’s Sinatra framework).

Easy to get going, and a good way to see many concepts other Haskell web frameworks use.

Great if you just want to get an application attached to a URL quickly!

Doesn’t enforce a model for templating or persistance (up to you to decide if this is a pro or a con!)

We will use the `Blaze` eDSL when we need to describe HTML.

Building a minimal Scotty app.

```
$ stack new scotty-example
$ cd scotty-example
```

Now we need to edit `scotty-example.cabal`.

Insert some dependencies:

```
executable scotty-example
  hs-source-dirs:      app
  main-is:             Main.hs
  ghc-options:         -threaded -rtsopts -with-rtsopts=-N
  build-depends:       base
                     , blaze-html
                     , scotty
```

Our source will live in `app/Main.hs`

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty

main = scotty 3000 $ do
  get "/" $ do
    html "Hello World!"
```

Running it

```
$ stack setup
$ stack build
$ stack exec scotty-example
```

and point a browser at `http://localhost:3000/`

It's short, but what does it all mean?

The first line:

```
{-# LANGUAGE OverloadedStrings #-}
```

turns on a Haskell language extension. Haskell has several string-like types, and this allows the compiler to select the right type for double-quoted literals.

Without overloaded strings this is what our example looks like:

```
import Web.Scotty
import Data.Text.Lazy

main = scotty 3000 $ do
  get (capture "/") $ do
    html $ pack "Hello World!"
```

Not terrible, but it's nice not to have to do those conversions manually!

Scotty gives us two monads:

- `ScottyM` which wraps the whole application (we can control the configuration of the app with this)
- `ActionM` which processes individual requests

We can add routes using:

```
addroute :: StdMethod -> RoutePattern -> ActionM () -> ScottyM ()
```

But it's usually easier to use one of a set of helper functions

```
get :: RoutePattern -> ActionM () -> ScottyM ()
post :: RoutePattern -> ActionM () -> ScottyM ()
matchAny :: RoutePattern -> ActionM () -> ScottyM ()
notFound :: ActionM () -> ScottyM ()
```

and so on…

Route patterns can be created from strings:

```
capture :: String -> RoutePattern
regex :: String -> RoutePattern
literal :: String -> RoutePattern
```

or in a structured way

```
function :: (Request -> Maybe [Param]) -> RoutePattern
```

When we use the `OverloadedStrings` language extension Haskell relies on an instance of the `IsString` class, which selects the `capture` function to do the conversion.

Once a route has been matched we provide one or more actions to modify the response. In our example we just replace the entire response body, which is what this function does:

```
html :: Text -> ActionM ()
```

The `Text` data type is more efficient (than `[Char]`) string representation, it's used throughout Scotty. Happily we can mostly just trust in the `OverLoadedStrings` when we need to create them.

Other options for setting the response body include

```
text :: Text -> ActionM ()
file :: FilePath -> ActionM ()
json :: ToJSON a => a -> ActionM ()
raw :: ByteString -> ActionM ()
```

Before setting the body we can manipulate the headers:

```
status :: Status -> ActionM ()
addHeader :: Text -> Text -> ActionM ()
setHeader :: Text -> Text -> ActionM ()
redirect :: Text -> ActionM a
```

Back to capturing routes - we can have several different routes captured (of course!)

```
main = scotty 3000 $ do
  get "/" $ do
    html "Hello World!"

  get "/greet" $ do
      html "Hello there"
```

Following the example of "Sinatra" we can also have "wildcards" in routes which are captured as parameters:

```
main = scotty 3000 $ do
  get "/" $ do
    html "Hello World!"

  get "/greet/:name" $ do
      name <- param "name"
      html $ mconcat [  "Hello there ",  name ]
```

Our server is currently not returning any *real* HTML yet!

```
$ curl http://localhost:3000/greet/Glenn
Hello there Glenn
```

Time to fix that!

Scotty doesn't have any HTML generation built in (lightweight, remember), but the `Blaze` library is widely used for creating HTML (and SVG, hint, hint!)

Some of the Blaze names conflict with Scotty names, so I've chosen to qualify the Blaze import

```
import qualified Text.Blaze.Html5 as H
import qualified Text.Blaze.Html5.Attributes as A
import qualified Text.Blaze.Html.Renderer.Text as R
```

(in practice you might like to do this the other way around, since you're more likely to be referring to Blaze names)

```
response :: Text -> Text
response n = do R.renderHtml $ do
                  H.h1 ( "Hello " >> H.toHtml n)
```

Blaze is fairly easy to get going with

```
longresponse :: Text -> Text
longresponse n = do
  R.renderHtml $ do
    H.head $ H.title "Welcome page"
    H.body $ do
      H.h1 "Welcome!"
      H.p ("Welcome to my Scotty app" >> H.toHtml n)
```

Attributes can be added to elements using the `!` operator:

```
myImage :: Html
myImage = img ! src "catPicture.jpg" ! alt "Awwww."
```

Functional Image Synthesis

- Conal Elliott's `Pan` was a language for functional image synthesis
- Ideas developed further in - `Fran` (Functional Reactive ANImation) - `Reactive` (A functional reactive programming framework) - `FieldTrip` (Functional Real-time 3D graphics) - Frameworks like `reactive-banana`
- Ideas lead to the invention of *Functional Reactive Programming* - Strongly influenced the development of languages like `Elm`

A quick overview of Pan

We will return to this next week to talk about the *reactive* side of things, and introduce animation.

What is an image?

```haskell
type Image = Point -> Colour
type Point = (Float, Float)
type Colour = (Float, Float, Float, Float)
```

Pan actually takes a more general view

```haskell
type Image a = Point -> a
```

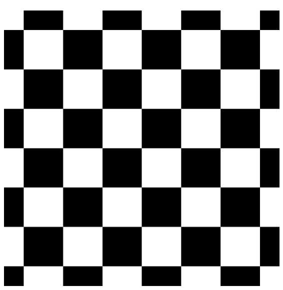
which allows us to create, say, masks as point sets:

```haskell
type Region = Image Bool
```

What does a Pan program look like?

```haskell
vstrip :: Region
vstrip (x,y) = abs x <= 0.5
```

```haskell
checker      :: Region
checker (x,y) = even (floor x + floor y
```



```haskell
altRings :: Region
altRings  = even . floor . dist0
dist0 (x, y ) = sqrt (x ^ x + y ^ y )
```

Combinators let us make complex images out of simpler ones

```
type ImageC = Image Colour
over :: ImageC -> ImageC -> ImageC
cond :: Image Bool -> Image a -> Image a -> Image a
```

Spatial transformations let us manipulate images

```
Transform :: Point -> Point -> Point

translate :: (Float, Float) -> Transform
scale :: (Float, Float) -> Transform
uscale :: Float -> Transform
rotate :: Float -> Transform

applyTrans :: Transform -> Image a -> Image a
```

What Pan actually does

What Pan actually does

- Pan programs generate Abstract Syntax Trees
- The AST is then given to a compiler which produces efficient C code
- The image fragments are fused along with the display function
- The result is algebraically simplified and optimised
- Finally C code is produced
- That code is compiled (by Visual C++, as it happens)

Reading: Functional Images chapter in "The Fun of Programming" (available on the web, More information)

Building a DSL for images

As an exercise let's develop some ideas in DSL's further

In his invited talk at the DSL summer school a couple of years ago Jeremy Gibbons presented a picture language (inspired by Brent Yorgey's `diagrams` package). What follows is partially based on that language.

The most basic element of a drawing is a simple *shape*. Leaving the actual implementation out for now:

```
data Shape = ...
empty, circle, square :: Shape
```

To talk about a shape's position or size we need a way to represent coordinates and vectors.

```
data Vector = Vector Double Double
type Point  = Vector
```

Shapes can be moved and deformed

```
data Transform = ...
identity :: Transform
translate :: Vector -> Transform
scale :: Vector -> Transform
rotate :: angle -> Transform
compose :: Transform -> Transform

(<+>) = compose
```

Now we have enough to draw something!

```
type Drawing = [(Transform,Shape)]

example = [ (scale (point 0.5 0.5) <+> translate (point 1.2 0.4), circle) ]
```

What might an interpretation function look like for a drawing?

One possibility is to ask if a point on the plane lies inside our drawing.

```
inside :: Point -> Drawing -> Bool
```

A shallow embedding

we haven't talked about an actual implementation yet, just the API.

Shallow embeddings are often easier when you can get away with them, but they are often harder to extend and compose.

A shallow embedding could look a lot like the region language we saw before.

```
type Shape = Point -> Bool

inside:: Point -> Drawing -> Bool
```

For shapes:

```
circle = \(Vector x y) -> x ^ 2 + y ^ 2 <= 1
```

Transformations apply themselves to points, for example:

```
translate (Vector tx ty) = \(Vector px py) = Vector (px - tx) (py - ty)
```

(aside: why is this subtracting? We are applying the *inverse* of the transformation, because we are translating the point we are asking about)

Our interface

```

inside1 :: Point -> (Transform,Shape) -> Bool
inside1 p (t,s) = s . t p

inside :: Point -> Drawing -> Bool
inside p d = or $ map (inside1 p) d
```

A deep embedding

Deep embeddings are often more complex, but it is easier to do things like add new interpretations, or to add optimisations.

In a deep embedding the types hold values:

```
data Vector = Vector Double Double
type Point  = Vector
```

```
data Shape = Empty
           | Circle
           | Square
             deriving Show
```

```
empty = Empty
square = Square
circle = Circle
```

That was easy!

```
data Transform = Identity
           | Translate Vector
           | Scale Vector
           | Compose Transform Transform
           | Rotate Matrix
             deriving Show

data Matrix = Matrix Vector Vector
```

Some example transformation constructions:

```
translate = Translate

rotate angle = Rotate $ matrix (cos angle) (-sin angle) (sin angle) (cos angle)
```

All the heavy lifting is done in the interpretation functions:

```
transform :: Transform -> Point -> Point
transform (Translate (Vector tx ty)) (Vector px py)  = Vector (px - tx) (py - ty)
transform (Rotate m)                 p = (invert m) `mult` p

invert :: Matrix -> Matrix
mult :: Matrix -> Vector -> Vector
```

```
inside :: Point -> Drawing -> Bool
inside p d = or $ map (inside1 p) d

inside1 :: Point -> (Transform, Shape) -> Bool
inside1 p (t,s) = insides (transform t p) s

insides :: Point -> Shape -> Bool
p `insides` Empty = False
p `insides` Circle = distance p <= 1
p `insides` Square = maxnorm  p <= 1
```

Building a DSL for animation

Our next step is to incorporate *animation*

We want to model how a drawing might change over time.

Signal functions

Do do this we will dip into the world of *Functional Reactive Programming* (FRP).

This is an approach that tries to integrate (notionally) *continuous* functions, with *time flow*, and *events* into Functional Programming.

Application domains for FRP include

- Animation
- Robotics
- Computer vision
- UI programming
- Simulation

The original paper for this is "Functional Reactive Animation" by Conal Elliott and Paul Hudak.

For our animation we will only need the notion of time-varying functions (but we'll leave out the notion of FRP "events" for now)

A signal is a function that emits values over time

```
type Time        = Double
newtype Signal a = Signal {at :: Time -> a}
```

Here's a simple signal function. It represents a function which always produces the same value no matter when you inspect it.

```
constant :: a -> Signal a
constant x = Signal (const x)
```

This next signal varies - at time `t` this signal will produce the value `t`.

```
timeS :: Signal Time
timeS = Signal id
```

We can transform the values in a stream with a function of this type

```
mapS   :: (a -> b) -> Signal a -> Signal b
```

In fact, this is exactly the type (and the meaning) that we would need for an instance of Functor. So we'll go ahead and do it that way instead!

```
instance Functor Signal where
    fmap = mapS
```

It would also be handy if we could lift function application into our Signal type.

If we use a function of this type:

```
applyS :: Signal (a -> b) -> Signal a -> Signal b
```

then we have the right tools to make our signals Applicative Functors as well!

```
instance Applicative Signal where
  pure = constant
  (<*>) = applyS
```

Implementations

```
instance Applicative Signal where
  pure x = Signal $ const x
  fs <*> xs = Signal $ \t -> (fs `at` t)  (xs `at` t)
```

Implementations

```
instance Functor Signal where
  fmap f xs = pure f <*> xs
```

Making use of our library

```
sinS :: Double -> Signal Double
sinS freq = mapT (freq*) $ fmap sin timeS


scale :: Num a =>  Signal a -> Signal a
scale = fmap ((30*) . (1+))

-- Discretize a signal
discretize :: Signal Double -> Signal Int
discretize = fmap round
```

```
-- convert to "analog"
toBars :: Signal Int -> Signal String
toBars = fmap (`replicate` '#')

displayLength = 500
-- display the signal at a number of points
display :: Signal String -> IO ()
display ss = forM_ [0..displayLength] $ \x ->
putStrLn (sample ss x)
```

```
magic :: Signal Double -> IO ()
magic = display . toBars . discretize . scale

main :: IO ()
main = magic $ sinS 0.1
```

Animation

We can use this to make some animations with our drawing.

Some preliminaries to allow us to draw things…

ASCII-art screen addressing module:

```
module Ansi where

cls :: IO ()
goto :: Int -> Int -> IO ()
```

Uses ANSI escape codes, should work with most “standard” terminal emulators.

Using this, a simplistic rendering module draws elements into a window. We need to provide a mapping from the *model* coordinates to the *screen* coordinates;

```haskell
data Window = Window Point Point (Int,Int)
makeWindow :: Point -> Point -> (Int,Int) -> Window
defaultWindow :: Window
pixels :: Window -> [[Point]]
render :: Window -> Drawing -> IO ()
```

```haskell
render :: Window -> Drawing -> IO ()
render win sh =  sequence_ $ map pix locations
  where
    pix (p,(x,y)) | p `inside` sh = goto x y  >> putStr "*"
                  | otherwise     = return ()

    locations :: [ (Point, (Int,Int) ) ]
    locations = concat $ zipWith zip (pixels win) (coords win)
```

And an IO function to render frames one after the other:

```haskell
animate :: Window -> Time -> Time -> Signal Drawing -> IO ()
```

(Loosely, "Sample this signal between these two intervals and draw what you find into a window")

What do Drawing Signals look like?

they can be very simple.

```haskell
staticBall :: Signal Drawing
staticBall = pure ball
     where ball = [(scale (point 0.5 0.5) <+> translate (point 1.2 0.4), circle)]
```

More interesting is a circle that is at different places at different times.

This signal could represent the "Y" coordinates of a ball that's bouncing up and down:

```haskell
bounceY :: Signal Double
bounceY = fmap (sin . (*3)) timeS
```

Then turn this into a Point Signal:

```haskell
posS :: Signal Point
posS = pure point <*> pure 0.0 <*> bounceY
```

Build this up into a singal of transformers:

```haskell
ts :: Signal Transform
ts = fmap translate posS
```

Finally, apply those transformations to a simple shape:

```haskell
addT :: (Transform, Shape) -> Transform -> (Transform, Shape)
addT (ts,s) t = (ts <+> t, s)

movingBall :: Signal Drawing
```

```
movingBall = fmap (:[]) $ (fmap (addT ball) ts
    where ball = (scale (point 0.3 0.3), circle)
```

Another example:

```
rotatingSquare :: Signal Drawing
rotatingSquare = fmap (:[]) $ fmap (addT sq) rssquare)] -- mapS (:[]) $ mapS sq rs
     where
           rs :: Signal Transform
           rs = fmap rotate timeS

           sq :: (Transform, Shape)
           sq = ( scale (point 0.5 0.5) <+> translate (point 1.2 0.4) , square)
```

```
bouncingBall :: Signal Drawing
bouncingBall = fmap (:[]) $ fmap (preaddT ball) ( fmap translate pos )
       where bounceY = fmap (sin . (3*)) timeS
             bounceX = fmap (sin . (2*)) timeS
             pos = pure point <*> bounceX <*> bounceY
             ball = ( scale (point 0.3 0.3), circle )
```