Lecture 8b. Design and DSLs

Functional Programming 2017/18

Alejandro Serrano

Goals

- Learn good practices for Haskell programming
  - Look at two libraries: `formatting` and `diagrams`
- Introduce the notion of *domain-specific language*

We use game-related examples

- Take note for your own game practical

Architectural and coding practices

Two different kinds of good practices

- *Architectural* practices describe how to arrange your types and functions to create a cohesive and understandable design
  - For example, "use classes for common abstractions"
  - The "macro" level of coding
- *Coding* practices describe patterns to write simpler and cleaner source
  - For example, "prefer guards over `if-then-else`"
  - The "micro" level of coding

This is not a black-or-white classification

Introduce one type per concept

Even if types are isomorphic, a separate one

- Improves readability and documents intention
- Prevents confusing one for the other
  - The compiler shouts if that is the case

1. Prevent "Boolean blindness"

```
data Status = Alive | Dead
data Level  = Finished | InProgress
-- instead of reusing Bool
```

2. Distinguish between points and vectors

```
data Point  = Point  Float Float
data Vector = Vector Float Float
-- Moves a point along a direction
translate :: Point -> Vector -> Point
```

Type classes declare common abstractions

- Types that have a position and can be moved

```
class Positioned a where
   getPosition :: a -> Point
   move        :: a -> Vector -> a
```

- Types that can be rendered to the screen

```
class Renderable a where
    render :: a -> Picture
```

- In general, *types that …*

ADTs may have multiple constructors

Declare closed sets of variants of a concept as constructors of a single data type

```
type Level = Int
data Enemy = Orc Level
           | Nazgul
           | Sauron
```

ADTs + type classes have a different flavor that OOP

- Do not try to import patterns from OOP into Haskell
- In particular, Haskell has not *inheritance*

Look for common abstractions

Haskell already comes with many common abstractions

- Equality with `Eq`, ordering with `Ord`, …

Monoids are a notable construction

- Remember, types with a binary associative operation with a neutral element
- In other words, you can combine two `A`s to get another `A`

Separate pure and impure parts

Pure functions deal only with values

- Always the same output for the same input
- The Haskell you have learnt until now

Impure functions communicate with the outside world

- Input and output, networking, interaction, …
- Marked in Haskell with the `IO` type constructor

Most common architecture

1. Impure part which obtains the input
2. Pure part which manipulates the data
3. Impure part which communicates the result

From previous lectures

- Do not use magic numbers to handle special conditions
  - Use your custom ADT, or use `Maybe` and `Either`
- Prefer pattern matching with guards over conditionals
- Favour higher-order functions over explicit recursion
- Write type signatures for every declaration

How to improve your style

- Compile your code with the maximum level of warnings
  - In the command line, use `ghc -Wall`
  - In your Cabal file, add to the stanza

```
executable your-project
    ...
    ghc-options:  -Wall
```

- Run HLint in your source files
  - HLint suggests improvements to your code

```
Found:
  and (map even xs)
Why not?
  all even xs
```

formatting and diagrams

Based on slides by Jurriaan Hage

printf in C

```
printf ("His name was %s, he earned %f.2 \
         at the age of %d", "John", 30500, 69);
```

- C is not picky about the types, but we are!
- `Text.Printf` provides a not type-safe `printf`

```
> import Text.Printf
> printf "%d plus %d makes %d\n" 2 3 5
2 plus 3 makes 5
> printf "%d plus %d makes %d\n" 2 3
2 plus 3 makes *** Exception: printf:
  argument list ended prematurely
> printf "%d plus %d makes %d\n" 2 3 "five"
2 plus 3 makes *** Exception: printf:
  bad formatting char 'd'
```

The solution: `formatting`

- Type safe string interpolation
  - strings have holes
  - values to fill the holes are passed in at some later time
  - formatted in the way indicated by the user
- Developed by Chris Done, based on Martijn van Steenbergen's `HoleyMonoid`
  - Available in Hackage

Run-time errors become type errors

```
> import Formatting
> let f = format (int % now " plus " % int
                        % now " makes " % int % now "\n")

> f 2 3 (2+3)
"2 plus 3 makes 5\n"

> f 2 3 "five"
<interactive>:17:7: error:
• No instance for (Data.String.IsString Integer)
  arising from the literal ‘"five"’
```

The primitives

To build a expression with holes we need to state

- Something is available *now*

```
now " plus "
```

- Something will become available *later*

```
later (fromString . show)
```

  - The argument to `later` is a function to process the hole

The combinator `(%)` sequences / composes two formatters

- Concatenates the texts as they become available

Customizable formatters

```
shown = later (fromString . show)
```

works for every type which is `Showable`

But other types might be formatted differently depending on the context:

- Integers can be formatted for different bases
  - Decimal, hexadecimal, binary
- Floats can be shown with different amount of precision, and optionally in scientific format

A huge variety is available in `Formatting.Formatters`

Formatters and functions

- Important distinction
  1. First you construct a formatter
  2. Then you apply `format` and the formatter becomes a function which expects arguments to fill the holes
  3. Once all are filled, a `Text` can be constructed
- Formatters can be used to construct other formatters

Formatter for Celsius

A `celsius` formatter can be written as follows

```
celsius :: Real a => Format a
celsius = fixed 1 % now "\x00b0" % now "C"
```

And this is how you can use it

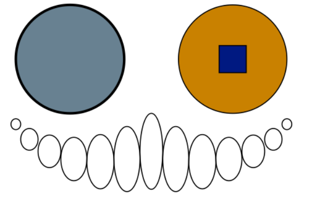
```
warmwhen :: Real a => a -> Text -> Text
warmwhen = format (now "It was " % celsius
                    % now " on " % text)

> warmwhen 22.9 "Oct 8"
"It was 22.9\176C on Oct 8"
```

About `diagrams`

Diagrams is a full-featured framework and embedded domain-specific language for creating declarative vector graphics and animations

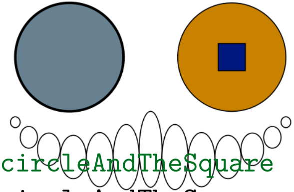
- Originally devised by Brent Yorgey
- Provides a fluent interface for drawing

Combinators by example



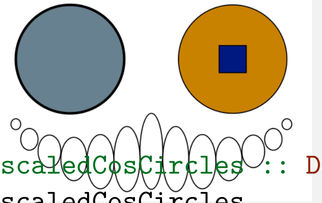
```
uglyFace :: Diagram B R2
uglyFace = circleAndTheSquare # center
           ===
           scaledCosCircles # center
```

Combinators by example



```
circleAndTheSquare :: Diagram B R2
circleAndTheSquare
  = theCircle ||| strut unitX ||| theSquare
  where
    theCircle = circle 1 # lw veryThick # fc gray
    theSquare = square 0.5 # fc navy
                `atop` circle 1 # fc darkgoldenrod
```

Combinators by example



```haskell
scaledCosCircles :: Diagram B R2
scaledCosCircles
  = foldr c mempty ([0.1,0.2..0.6] ++ [0.7,0.6..0.1])
  where
    c rad res = circle rad
                  # scaleX (1 - rad)
                  # translateY (0 - sin (pi * rad))
                 ||| res
```

Setting attributes

- Functions are used to set properties of diagrams

  ```
  fc gray (circle 1)
  ```

- `(#) :: a -> (a -> b) -> b` is inverse application
  - Fluency in action: diagram first, attribute later

  ```
  circle 1 # fc gray
  ```

- Sometimes we want to use the function

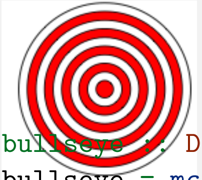
  ```
  map (fc gray) [circle 1, circle 2, circle 3]
  ```

The role of type classes

```haskell
fc :: (HasStyle a, ...) => Colour Double -> a -> a
```

- Type classes ensure that only a values that "have style" can be passed to `fc`
- By adding an instance for something that has style, we can apply `fc` and similarly for various other attributes

Another example



```haskell
bullseye :: Diagram B R2
bullseye = mconcat $ zipWith
  (\s c -> circle s # fc c # lw veryThin)
  [0.1, 0.2 .. 1.0] (cycle [red, white])
```

`mconcat` and `mempty` come from the `Monoid` class

More type classes

```
vcat :: ( Juxtaposable a, HasOrigin a
        , Monoid a, V a ~ R2 )
     => [a] -> a
```

- `Juxtaposable a` holds for all types of things that can be juxtaposed (put side by side, in any direction necessary)
  - `vcat'` allows you to introduce spacing
- `vcat` lines up diagrams vertically, based on the origin of the argument diagrams. Hence, we need `HasOrigin a`
- `V a ~ R2` implies we live in 2D

Lots of things undiscussed

- Envelopes, traces, paths, colour manipulation, alpha blending, text, texture
- 3D and animation
- In all, `diagrams` is a pretty serious library

Domain-specific languages

What is a DSL?

`formatting` and `diagrams` are examples of **domain-specific languages**, DSLs for short

- As opposed to *general purpose languages*, they are only useful for some programming tasks
  - Less powerful but easier to optimize
- The goal is to allow more people than just trained programmers to use the DSL
  - More intuitive and declarative

Other examples: SQL for databases, HTML for web pages

Walid Taha on DSLs

In a domain-specific language:

1. The domain is well-defined and central
2. The notation is clear
3. The informal meaning is clear
4. The formal meaning is clear and implemented

Without the latter, we have a *jargon*

What is an embedded DSL?

`formatting` and `diagrams` are DSLs **embedded** in Haskell

- The syntax is encoded inside that of a host language
- Advantages:
  - Escape hatch to the host language
  - Reuse existing libraries, compilers, IDEs
  - Easy to combine DSLs
- At the very least, useful as a prototype

What host language?

- Some provide extensibility as part of their design
  - Ruby, Python, Scheme / Racket
- Others are rich enough to encode DSLs with ease
  - Haskell, C++

What host language?

- Some provide extensibility as part of their design
  - Ruby, Python, Scheme / Racket
- Others are rich enough to encode DSLs with ease
  - Haskell, C++

Haskell as a host language

- Higher-order functions, parametric polymorphism and type classes go a long way
- The ability to declare custom operators also helps
- EDSLs are simply libraries with some kind of “fluency”
  - Types encode domain terms and invariants
  - Special operators to combine those values

Deep and shallow embedding

- *Shallow*: the code you write is interpreted in the normal way by the Haskell interpreter
  - `diagrams` is an example of those
- *Deep*: what you write implicitly constructs a DSL program that can be manipulated
  - The program can be validated or optimized before being emitted
  - `esqueleto` provides an embedded DSL to build SQL queries inside Haskell

Summary

- Embedded DSLs can provide elegant solutions to problems in a given domain
- Write your program as if you were developing a DSL
  - Use types which reflect the domain terms
  - Use type classes to abstract common concepts
- Strong type systems provide additional guarantees