Table of Contents

Three Exceptional Haskell Libraries

Benson Joeris
http://bjoeris.com

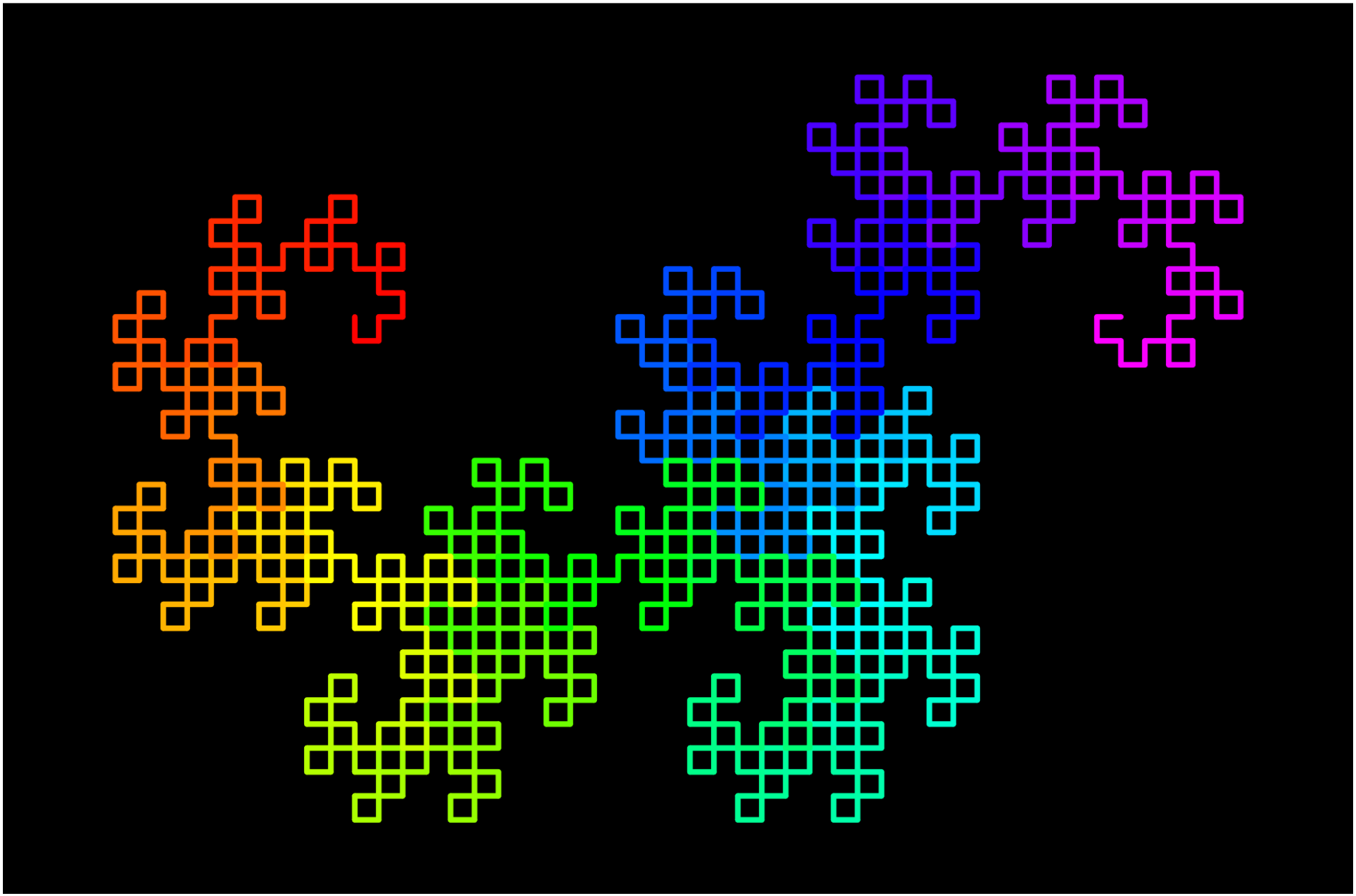
pluralsight
hardcore developer training

Overview

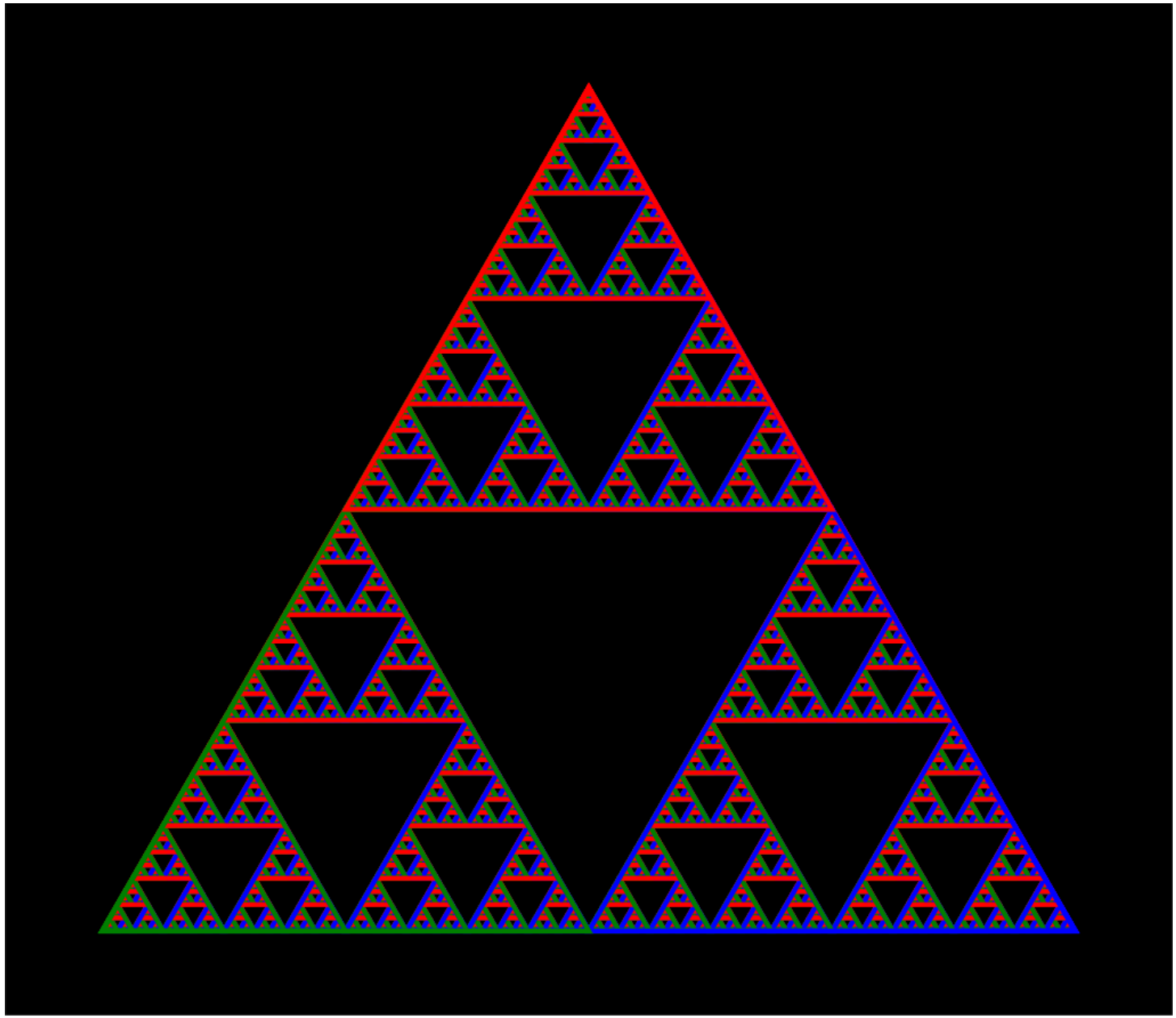
- Diagrams
  - Declarative drawing
- Lens
  - Access and manipulate nested data structures
- Functional Reactive Programmnig
  - Declarative event handling

Table of Contents

Diagrams



Diagrams



Diagrams

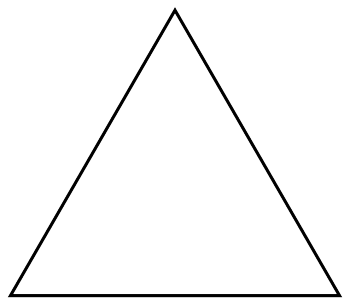
- Declarative
  - *What* to draw, rather than *how* to draw it
- Compositional
- Multiple Backends
  - Files: SVG, Bitmap, PDF, Postscript
  - Display: GTK, OpenGL

Diagrams

```
cabal install diagrams
```

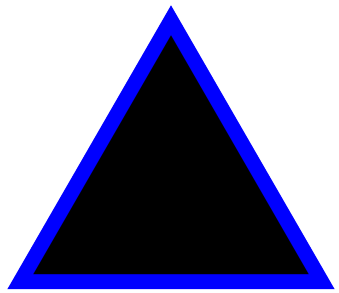
Diagrams

```haskell
{-# LANGUAGE NoMonomorphismRestriction #-}

import Diagrams.Prelude
import Diagrams.Backend.SVG.CmdLine

main = mainWith (triangle 1 :: Diagram B R2)
```



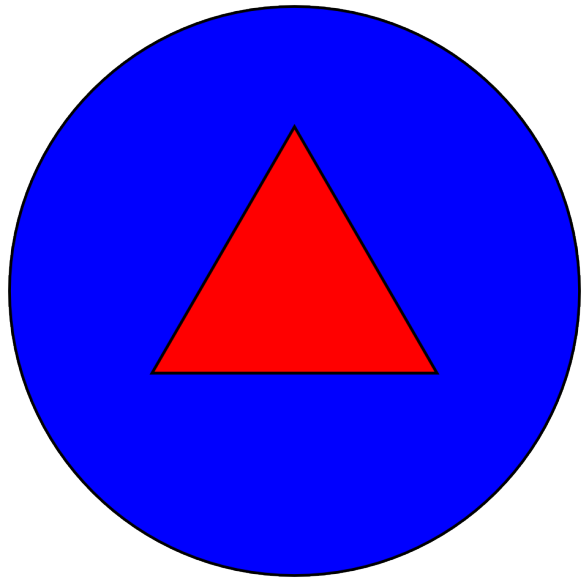
Diagrams

```
{-# LANGUAGE NoMonomorphismRestriction #-}

import Diagrams.Prelude
import Diagrams.Backend.SVG.CmdLine

example :: Diagram B R2
example = triangle 1  #  fc black # lc blue # lw 0.1

main = mainWith example
```



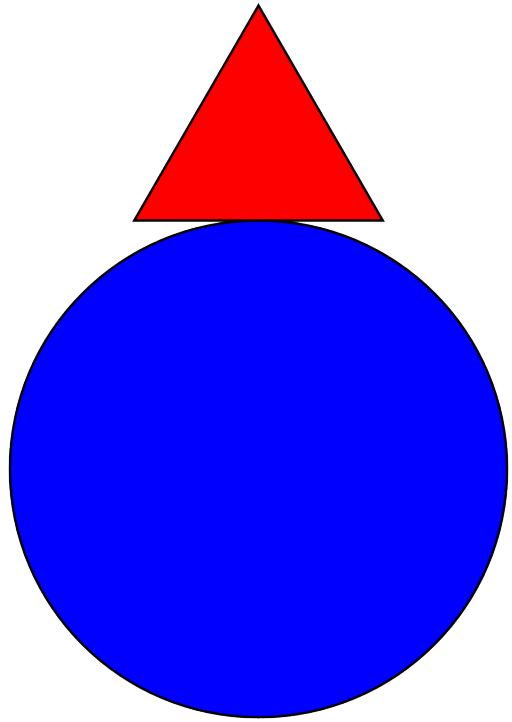
Diagrams

```
example :: Diagram B R2
example = triangle 1 # fc red
          <>  circle 1 # fc blue
```



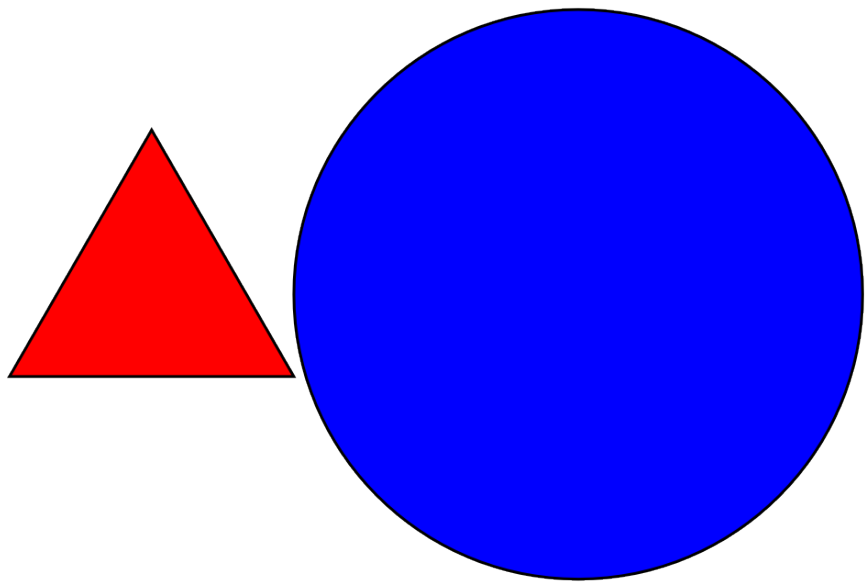
Diagrams

```
example :: Diagram B R2
example = triangle 1 # fc red
          ===  circle 1 # fc blue
```



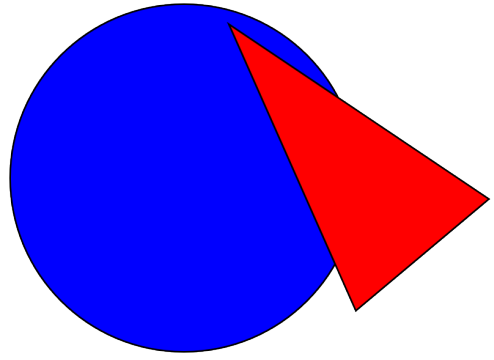
Diagrams

```
example :: Diagram B R2
example = triangle 1 # fc red
          |||  circle 1 # fc blue
```

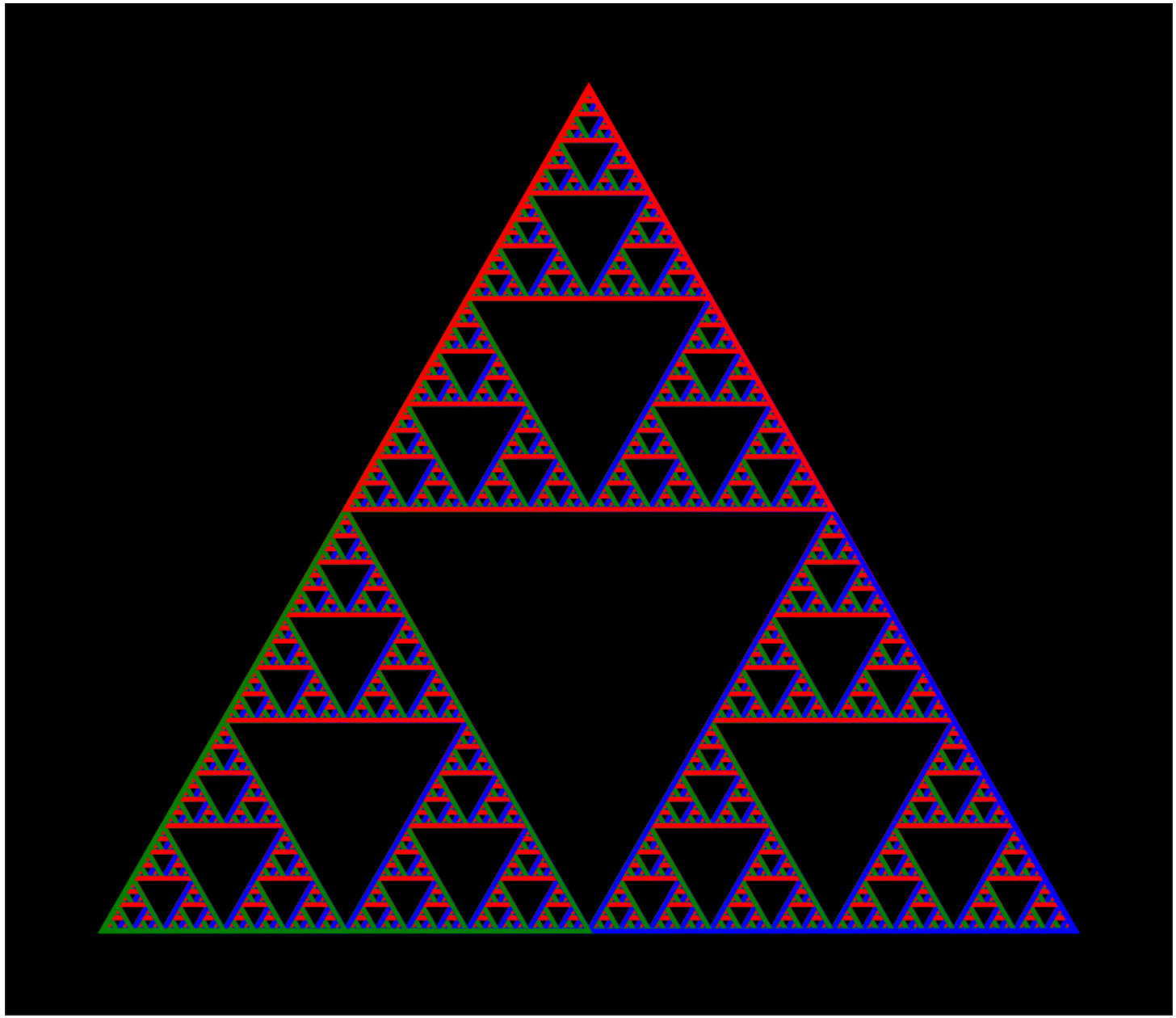


Diagrams

```
example :: Diagram B R2
example = triangle 1 # fc red
                     # scaleY 2
                     # rotateBy (1/9)
                     # translateX 1
          <> circle 1 # fc blue
```



Diagrams



Diagrams

```
sierpenski :: Int -> Diagram B R2
sierpenski n
  | n == 0 = mempty
  | otherwise =
      piece red  ===
          (centerX (piece green  |||  piece blue))
      where
       piece color = triangle 1 # lc color # alignT
                       <> sierpenski (n-1)
                          # scale 0.5 # alignT

drawing :: Diagram B R2
drawing = sierpenski 7 # center # pad 1.2
                       # bg black # lw 0.01
```

Table of Contents

Lens

- Access and manipulate nested data
  - Manipulating *immutable* nested data is tricky

Lens: The Problem

```
data Player = Player
  { playerName      :: String
  , playerSalary    :: Int
  , playerStats     :: PlayerStats }
data PlayerStats = PlayerStats
  { goals           :: Int
  , gamesPlayed     :: Int }
```

Lens: The Problem

- Nested query

```
playerGoals :: Player -> Int
playerGoals player =
  goals (playerStats player)
```

Lens: The Problem

- Simple update

```
increasePlayerSalary :: Player -> Int -> Player
increasePlayerSalary player raise =
  player { playerSalary =
             playerSalary player + raise }
```

Lens: The Problem

- Nested update

```
incrementPlayerGoals :: Player -> Player
incrementPlayerGoals player = player
  { playerStats = (playerStats player)
      { goals = goals (playerStats player) + 1}}
```

Lens: The Problem

- Nested update

```
void incrementPlayerGoals(Player player) {
    player.playerGoals += 1;
}
```

Lens: The Problem

```
data Player = Player
  { playerName   :: String
  , playerSalary :: Int
  , playerStats  :: PlayerStats }
data PlayerStats = PlayerStats
  { goals        :: Int
  , gamesPlayed  :: Int }
data Team = Team
  { teamName     :: String
  , teamPlayers  :: [Player]}
```

```
incrAllPlayersGamesPlayed :: Team -> Team
incrAllPlayersGamesPlayed = ?
```

Lens: Installing

```
cabal install lens
```

```
import Control.Lens
```

Lens: Using Lenses

- Query

```
getPlayerName :: Player -> String
getPlayerName player =
  player  ^.  playerName
```

```
playerName :: Lens' Player String
```

```
(^.) :: s -> Lens' s a -> a
```

Lens: Using Lenses

- Nested query

```
getPlayerGoals :: Player -> Int
getPlayerGoals player =
  player ^. playerStats . goals
```

```
playerStats :: Lens' Player PlayerStats
```

```
goals :: Lens' PlayerStats Int
```

```
(playerStats . goals) :: Lens' Player Int
```

Lens: Using Lenses

- Update

```
setPlayerSalary :: Player -> Int -> Player
setPlayerSalary player newSalary =
  (playerSalary  .~  newSalary) player
```

```
setPlayerSalary :: Player -> Int -> Player
setPlayerSalary player newSalary =
  player  &  playerSalary .~ newSalary
```

```
player.playerSalary = newSalary;
```

Lens: Using Lenses

- Update

```
increasePlayerSalary :: Player -> Int -> Player
increasePlayerSalary player raise =
  player & playerSalary  +~  raise
```

```
player.playerSalary += raise;
```

Lens: Using Lenses

- Nested update

```
incrementPlayerGoals :: Player -> Player
incrementPlayerGoals player =
  player & playerStats . goals +~ 1
```

```
player.playerStats.goals += 1;
```

Lens: Using Lenses

- Lists

```
incrAllPlayersGamesPlayed :: Team -> Team
incrAllPlayersGamesPlayed team =
  team & teamPlayers . traverse
       . playerStats . gamesPlayed +~ 1
```

```
traverse :: Lens' [a] a
```

```
foreach(player in team.teamPlayers)
  player.playerStats.gamesPlayed += 1;
```

Lens: Using Lenses

- Lists

```
totalGamesPlayed :: Team -> Int
totalGamesPlayed team =
  team & sumOf ( teamPlayers . traverse
               . playerStats . gamesPlayed)
```

Lens: Creating Lenses

```
{-# LANGUAGE TemplateHaskell #-}
```

```
data Player = Player
  { _playerName   :: String
  , _playerSalary :: Int
  , _playerStats  :: PlayerStats}
```

```
makeLenses ''Player
```

```
playerName   :: Lens' Player String
playerSalary :: Lens' Player Int
playerStats  :: Lens' Player PlayerStats
```

Lens

- Easy access and manipulation of nested data
- Steep learning curve

Table of Contents

Functional Reactive Programming

- Declarative event handling
  - UI
  - Animation
  - Robotics
- Implementations
  - Haskell: 20+ implementations on hackage
  - Sodium (Java, Haskell, C++)
  - Flapjax (Javascript)
  - Threepenny-Gui

Threepenny-Gui Setup

```
cabal install threepenny-gui
```

Functional Reactive Programming

- Time variation

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

```
time :: Behavior Time
```

```
afterMidnight :: Behavior Bool
```

Functional Reactive Programming

- Time variation

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

```
mouseClicks :: Event MouseButton
```

Arithmetic Example

```
bInput1 :: Behavior String
bInput1 = bValue input1
```

```
bInput1Num :: Behavior (Maybe Int)
bInput1Num = (liftA readMaybe) bInput1
```

```
bInput2 :: Behavior String
bInput2 = bValue input2
```

```
bInput2Num :: Behavior (Maybe Int)
bInput2Num = (liftA readMaybe) bInput2
```

Arithmetic Example

```
bInput1Num :: Behavior (Maybe Int)
bInput2Num :: Behavior (Maybe Int)
```

```
bSum :: Behavior (Maybe Int)
bSum = (liftA2 (liftM2 (+))) bInput1Num bInput2Num
```

```
bResult :: Behavior String
bResult = (liftA showMaybe) bSum
```

```
element result  #  sink text bResult
```

Counter Example

```
eUp :: Event ()
eUp = UI.click buttonUp
```

```
eIncrement :: Event (Int -> Int)
eIncrement = fmap (\() -> (+1)) eUp
```

```
eDown :: Event ()
eDown = UI.click buttonDown
```

```
eDecrement :: Event (Int -> Int)
eDecrement = fmap (\() -> (subtract 1)) eDown
```

Counter Example

```
eIncrement :: Event (Int -> Int)
eDecrement :: Event (Int -> Int)
```

```
eChange :: Event (Int -> Int)
eChange = unionWith (.) eIncrement eDecrement
```

```
bCounter :: Behavior Int
bCounter = accumB 0 eChange
```

```
bResult :: Behavior String
bResult = liftA show bCounter
```

```
element result # sink text bResult
```

Non-GUI Events

```
data Message = Message
  { msgSender :: String
  , msgText   :: String }
```

```
eIncoming :: Event Message
```

```
bHistory :: Behavior [Message]
bHistory = accumB [] (fmap (:) eIncoming)
```

Functional Reactive Programming

- Declarative reactions to changes over time
- Composable

Table of Contents

Summary

- Diagrams
- Lens
- Functional Reactive Programming