

LwDSLs in Haskell

Why to use them, what they are, and how they work.

Why do we want to use a LwDSL?

- ▶ Code size reduction/reuse
- ▶ More maintainable
- ▶ More easy to refactor
- ▶ All this can save us **money**



What is a DSL? (LwDSLs in a moment)

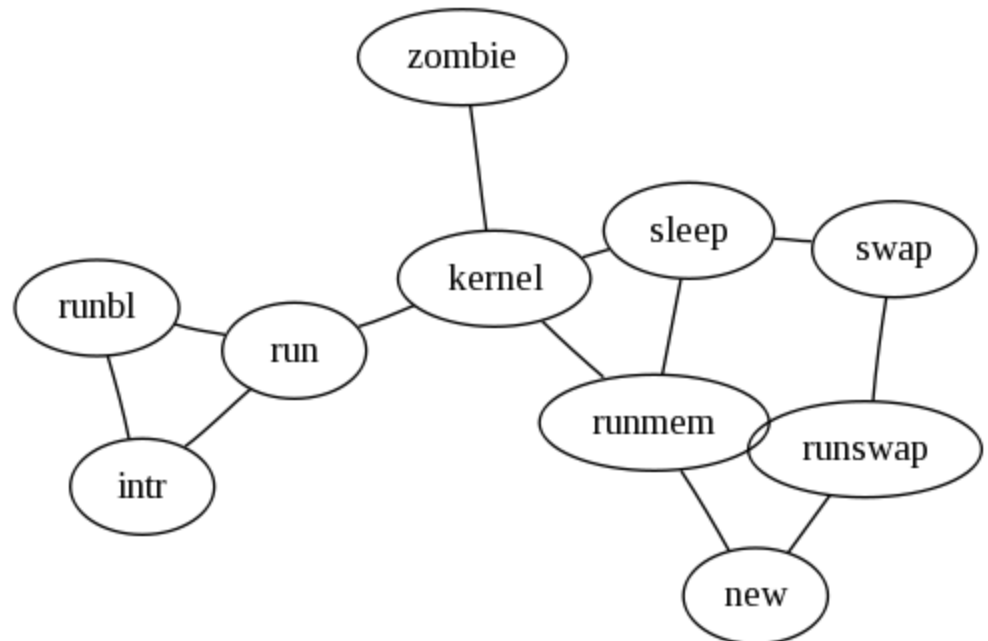
- ▶ A language designed to express exactly, but not more than, the possible concepts of a constrained domain.
- ▶ Boolean Logic Gates?
 - ▶ No string processing.
 - ▶ No explicit mathematical operators.
 - ▶ No ORM, databases, network sockets, or... anything else.
 - ▶ ONLY expressions of gates and their connections



Examples!

▶ Graphviz

```
graph G {  
  run -- intr;  
  intr -- runbl;  
  runbl -- run;  
  run -- kernel;  
  kernel -- zombie;  
  kernel -- sleep;  
  kernel -- runmem;  
  sleep -- swap;  
  swap -- runswap;  
  runswap -- new;  
  runswap -- runmem;  
  new -- runmem;  
  sleep -- runmem;  
}
```



Examples!

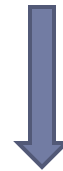
▶ Lex/Flex

```
%{
#include <stdio.h>
}%

%%
start ..... printf("Start command\n");
[0123456789]+ ..... printf("NUMBER\n");
stop ..... printf("Start command\n");
%%
```



"start 123 456 stop"



[start command, NUMBER,
NUMBER, stop command]



Examples!

▶ SQL

Table

First	Last	Age	Phone
Jim	Jones	23	555-5556
Alex	Smith	14	555-5557
Tim	Van Vander	52	555-5558



```
SELECT 'First', 'Age' FROM 'Table';
```



Selected

First	Age
Jim	23
Alex	14
Tim	52



What is a **LwDSL**?

- ▶ A DSL hosted in a general purpose language (C++, Scala, Ruby, Haskell, etc.)
- ▶ Enough framework and syntax to represent the expression in the domain (an Abstract Syntax Tree or graph)
- ▶ Inherits features and libraries of the host language (functions, types, arrays, lists, debuggers, compilers, interpreters).
- ▶ Bypasses a the need for a full grammar/parser/codegen.



Examples of LwDSLs (in C)

▶ OpenGL

```
▶ glBegin(GL_TRIANGLES);  
    glVertex3f( 0.0, 1.0, 0.0);  
    glVertex3f(-1.0,-1.0, 0.0);  
    glVertex3f( 1.0,-1.0, 0.0);  
    glEnd();
```

▶ Protothreads

```
▶ struct pt pt;  
PT_THREAD(do_io(struct pt * pt))  
{  
    PT_BEGIN(pt);  
    begin_async_io_action();  
    PT_WAIT_UNTIL(pt, async_io_finished());  
    PT_END(pt);  
}
```



Two Forms of LwDSL (EmbeddedDSL)

▶ Shallow Embedding

- ▶ Evaluation is done immediately. Runtime of the host language executes and emits output.
- ▶ “Target-less”

▶ Deep Embedding

- ▶ Evaluation produces an intermediate structure. Other tools convert intermediate structure to target or evaluate it directly.
- ▶ Not the final form.



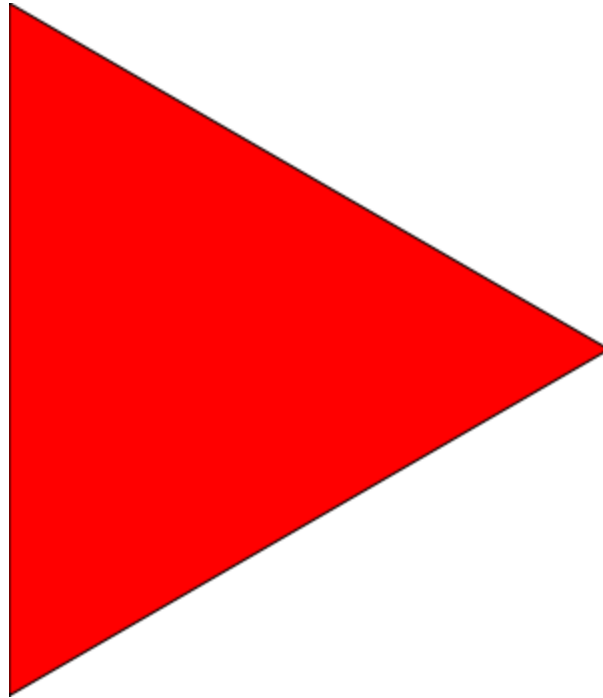
Example of Shallow Embedding

- ▶ Brent Yorgey's 'Diagrams' package
 - ▶ Simple interface for describing a 2D diagrammed scene
 - ▶ Diagram primitives include:
 - ▶ 'vcat' – vertical concatenation
 - ▶ 'hcat' – horizontal concatenation
 - ▶ 'regPoly' – create regular polygon
 - ▶ 'fillColor' – fill a polygon with a specific color
 - ▶ 'rotate' – rotates a polygon
 - ▶ Compilation command: 'renderAs'
 - ▶ If we wanted a red square, we write:
 - ▶ `fillColor red (regPoly 4 1)`



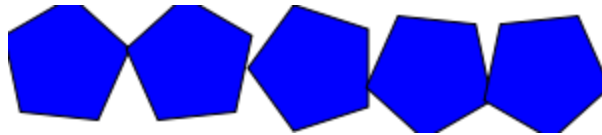
Building Up a Diagram

- ▶ `fillColor red (regPoly 3 1)`



Building Up a Diagram

```
▶ dgram = let rotations = [1/6,2/6,3/6,4/6,5/6]
          pgon = fillColor blue (regPoly 5 1)
          pgons = zipWith rotate rotations (cycle [pgon])
          in hcat pgons
```



Diagrams, cont.

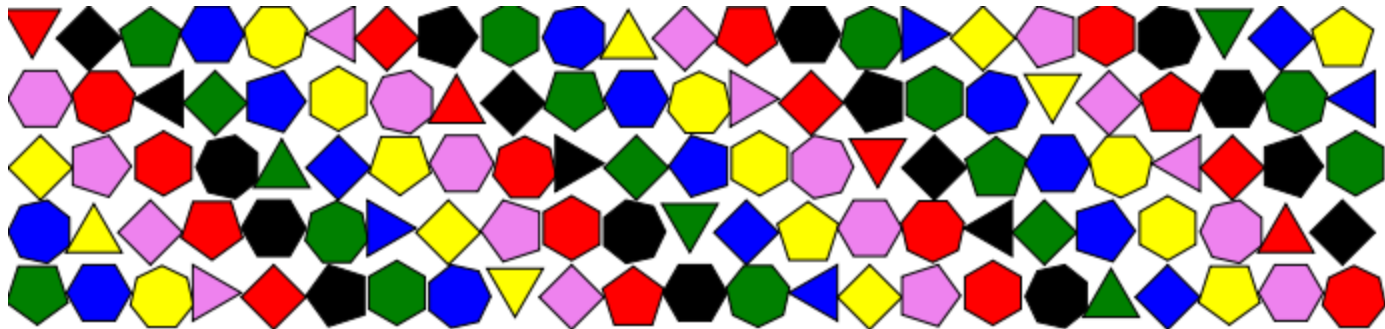
```
module Main where

import Data.List
import Graphics.Rendering.Diagrams

numCols,numRows :: Int
numCols = 5
numRows = 5
main = renderAs SVG "out2.svg" (Width width) dgram
  where width = (fromIntegral numCols) * 30
  -- The diagram
  dgram = vcat $ take numRows rows
    where rows = map hcat $ plines crps
  -- The rows of polygons
  plines [] = []
  plines pls = let (l,ls) = splitAt numCols pls
                  in l : (plines ls)
  -- Colored and rotated polygons
  crps = zipWith fillColor cs rps
    where rps = zipWith rotate rs ps
          rs = cycle . map (/ 4) $ [1..4] -- 4 rotations
          ps = cycle . map (flip regPoly 1) $ [3..7] -- 5 polygons
          cs = cycle [red,black,green,blue,yellow,violet] -- 6 colors
```



Finished Product



Example of Deep Embedding

- ▶ Tom Hawkins' 'Atom' package
 - ▶ Synthesizes hard real time systems.
 - ▶ Targets C code
 - ▶ Atom programs produce C programs which we compile and run on the target.
 - ▶ Gives guaranteed limits to time/space usage
 - ▶ Includes source-level coverage data



Haskell – An Excellent Choice

- ▶ **Strong, expressive, types**
 - ▶ Help us make sure our LwDSL is properly constrained
 - ▶ Types express high-level design
 - ▶ Some ‘business logic’ can be statically checked rather than dynamically
- ▶ **Terse**
 - ▶ Expressions are dense and meaningful
 - ▶ `qs [] = []`
`qs (x:xs) = qs (filter (<x) xs) ++ [x] ++ qs (filter (>= x) xs)`
 - ▶ Very little syntactic overhead
- ▶ **Tracking state is easier (most objects are immutable)**
 - ▶ Intermediate structures are easier to “rewind”
- ▶ **Higher Order Functions and Partial Evaluation**
 - ▶ Composing concepts made easy



In-depth Example – Simple Boolean Gates

- ▶ Gator – LwDSL for describing boolean logic gate arrays.
 - ▶ Intentionally simple interface
 - ▶ Intermediate structure (deep embedding)
 - ▶ Single implemented target, many possible
- ▶ Basic Primitives
 - ▶ Inputs
 - ▶ Outputs
 - ▶ AND/OR/XOR (more can be added easily)
 - ▶ 'connect'



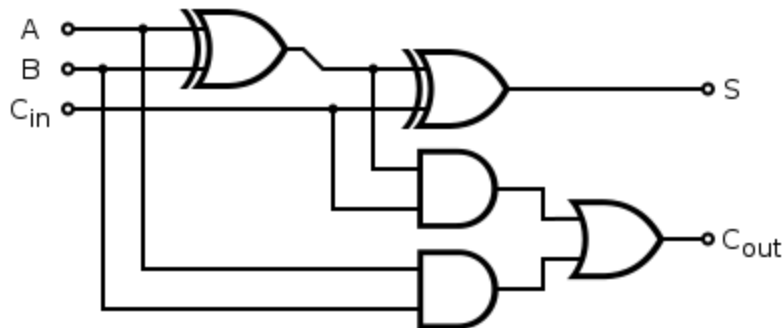
Gator's Intermediate Structure

- ▶ Our circuits form a graph.
 - ▶ Gates with wires to other gates.
 - ▶ Inputs to Outputs
- ▶ This intermediate form can be transformed into several things
 - ▶ A graphical dump of the intermediate structure
 - ▶ Code to simulate the gates
 - ▶ Direct evaluation of the intermediate structure



Simple Example – Generic Full Adder

- ▶ Adds two bits together.



```
fullAdder inA inB inC = do
  xor0 <- inA ^^ inB
  xor1 <- xor0 ^^ inC

  and0 <- xor0 && inC
  and1 <- inA && inB

  or0 <- and0 || and1

  return (xor1, or0)
```



Using the Generic Full Adder

- ▶ Defines inputs and outputs. Ties output of full adder to the outputs.

```
logic = do
    inA <- newInputN "A"
    inB <- newInputN "B"
    inC <- newInputN "Cin"
    outS <- newOutputN "S"
    outC <- newOutputN "Cout"

    (s,c) <- fullAdder inA inB inC

    connect s outS
    connect c outC
```



Combining the Full Adder

▶ 4 Bit Adder

```
logic = do
    [...define inputs...]

    (s0,c0) <- fullAdder inA0 inB0 inC
    (s1,c1) <- fullAdder inA1 inB1 c0
    (s2,c2) <- fullAdder inA2 inB2 c1
    (s3,c3) <- fullAdder inA3 inB3 c2
```



Questions?

