Domain specific embedded languages

Wouter Swierstra

Goals

- What is a DSEL?
- Why use Haskell?
- Show some techniques, tricks & terminology.
- Study examples.

What is a domain-specific language?

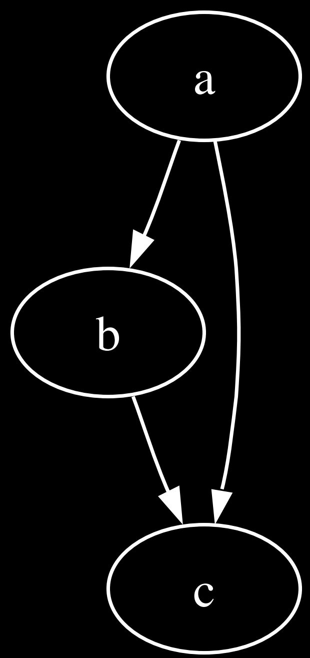
Wikipedia defines a DSL:

> *a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique*

Example:

The dot language for describing directed graphs:

```
digraph untitled
  {
  a -> b;
  b -> c;
  a -> c;
  }
```



DSL

- A domain specific language is a language designed to solve one problem and to solve it well.
- A DSL is not a general purpose programming language.
- Some DSLs are designed to be used by people who are not programmers.

Designing a DSL

Writing a compiler, even for a simple DSL is a lot of work.

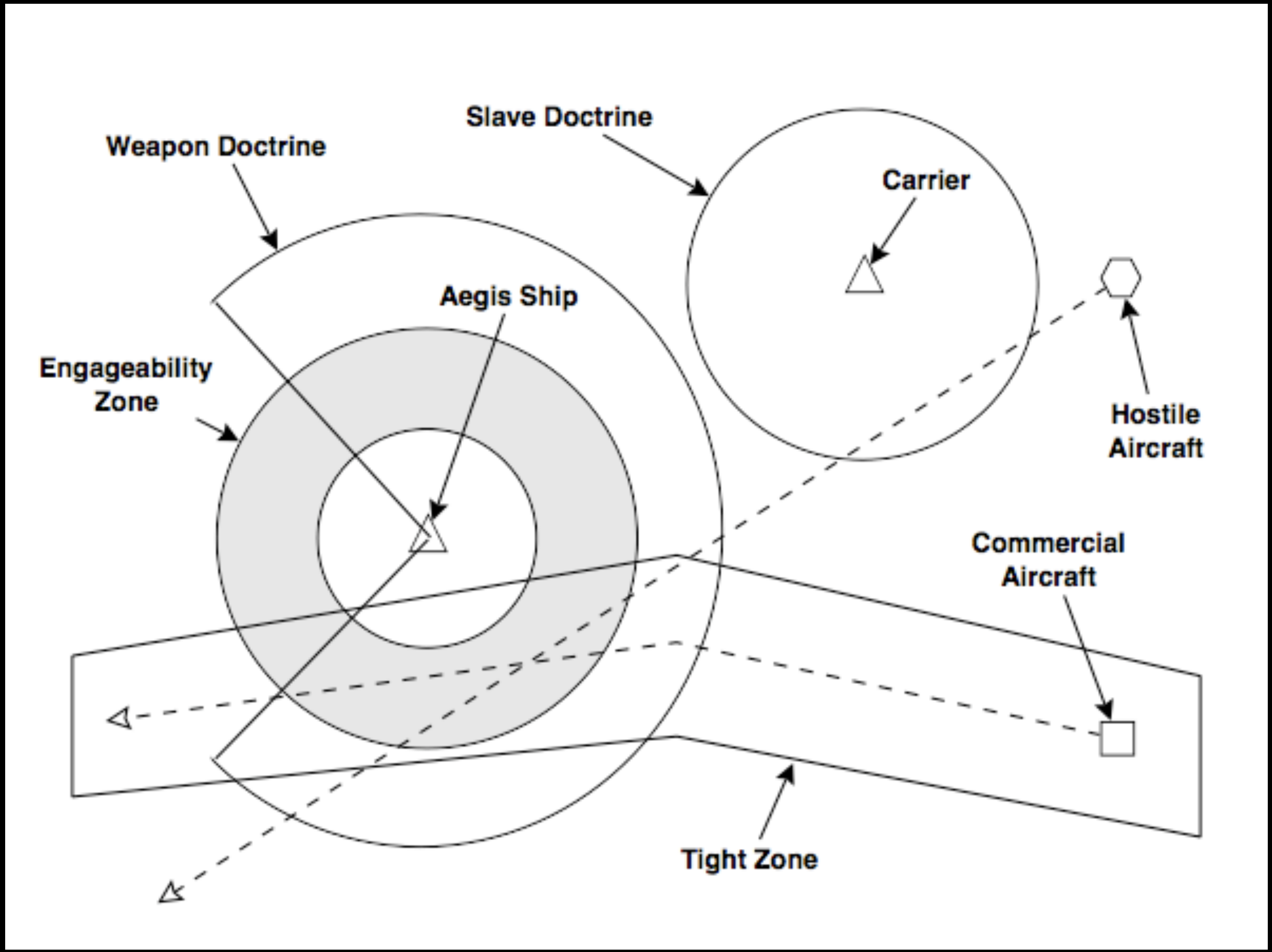
- lexer;
- parser;
- pretty printer;
- “code generation”;
- type checking...

And what if your first design is wrong?

Domain specific embedded languages

- Don't invest implementation effort in a compiler until you are sure about what you and your users want.
- Start by **embedding** your DSL in a general purpose programming language.
- Rapid language prototyping.

NSWC Experiment



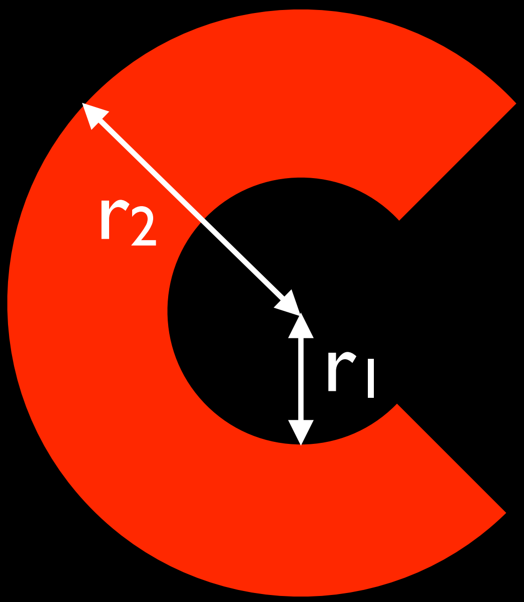
Aegis weapons system

Geo-server

- Central questions:
  - When is a point in a region?
  - How can we describe complex regions?

Monolithic solution

```
public bool isInRange
 (r1 : float, r2 : float,
  x : float, y : float) {
  sqrt(x^2 + y^2) >= r1 &&
  sqrt(x^2 + y^2) <= r2 &&
  ... sin φ ...
}
```



Haskell solution

- There are two **types** of data involved:
  - Points:

    ```
    type Point = (Float, Float)
    ```

  - Regions:

    ```
    type Region = Point -> Bool
    ```

- Once we know the types, the rest is “easy.”

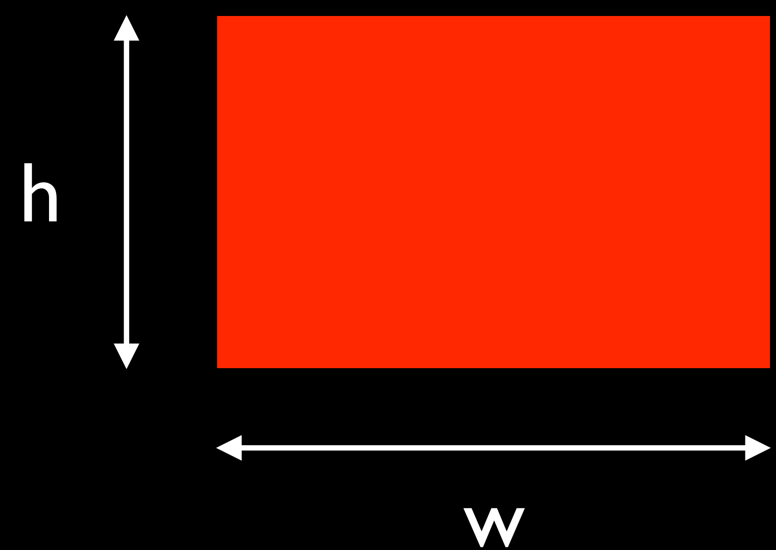
Design goal

- We need to find:
  - the smallest possible primitive combinators to describe simple regions;
  - and region combinators to build "bigger" regions from smaller ones.

Primitive regions - I

Rectangles centered at the origin:

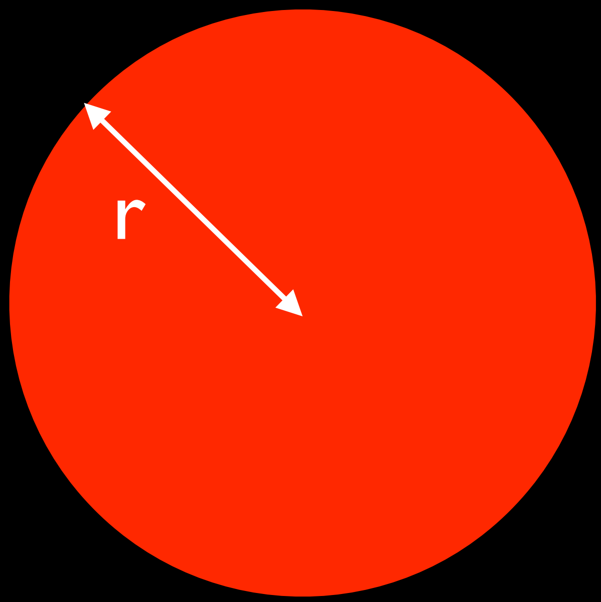
```
type Width  = Float
type Height = Float

rect :: Width ->
  -> Height
  -> Region
rect w h (x,y) =
  x <= w && y <= h
```



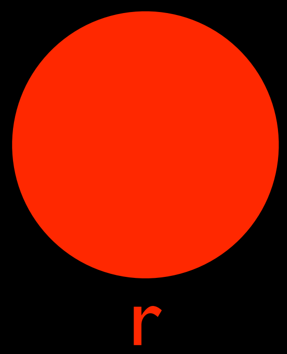
Primitive regions - II

Circles centered at the origin:

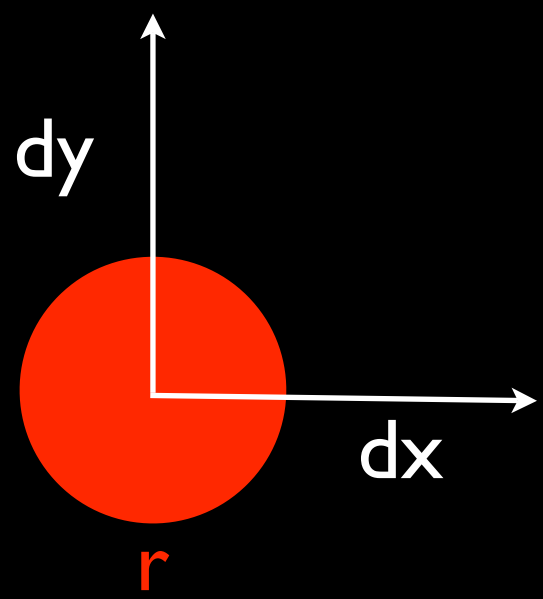
```
type Radius = Float

circle :: Radius -> Region
circle r (x,y) =
  let d = sqrt (x^2 + y^2)
  in d <= r
```



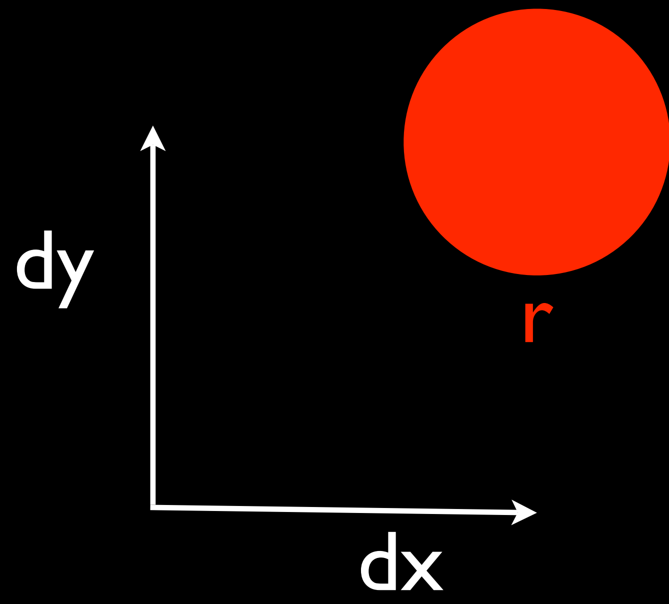
Shifting



Shifting

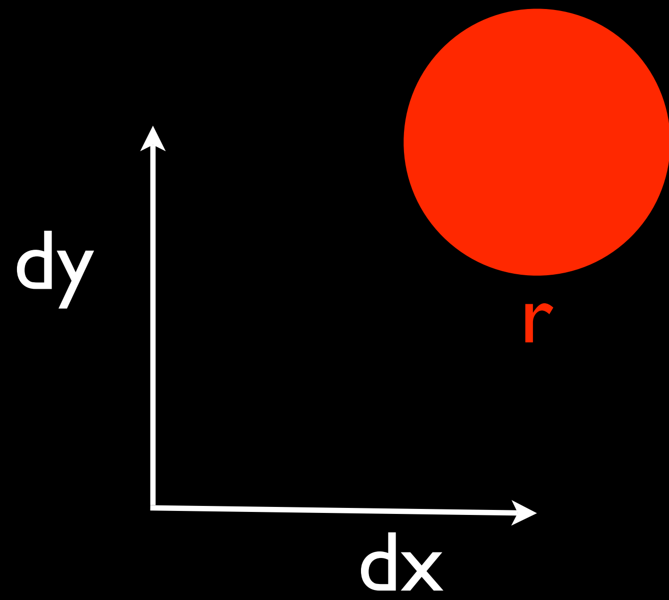


Shifting



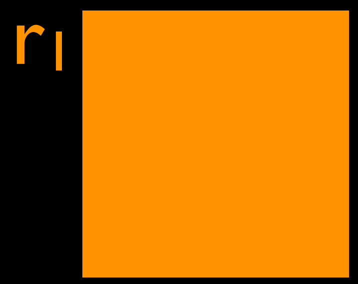
Shifting

```
shift :: (Float, Float)
  -> Region
  -> Region
shift (dx,dy) r =
  \(x,y) ->
    r (x - dx, y - dy)
```

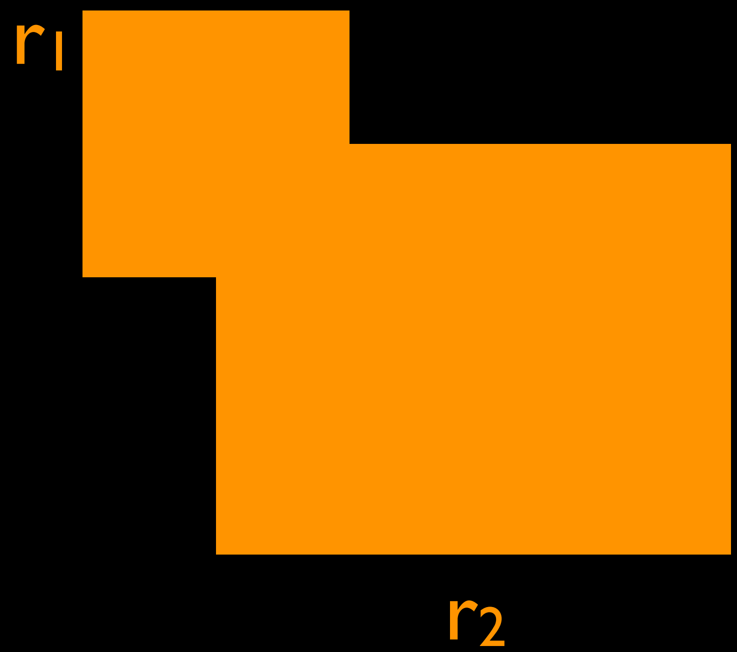


Intersection

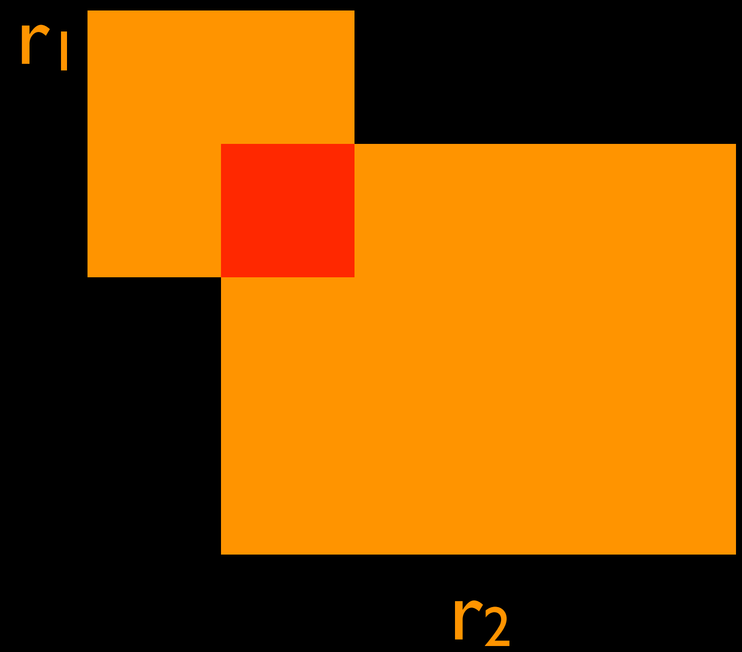
Intersection



Intersection

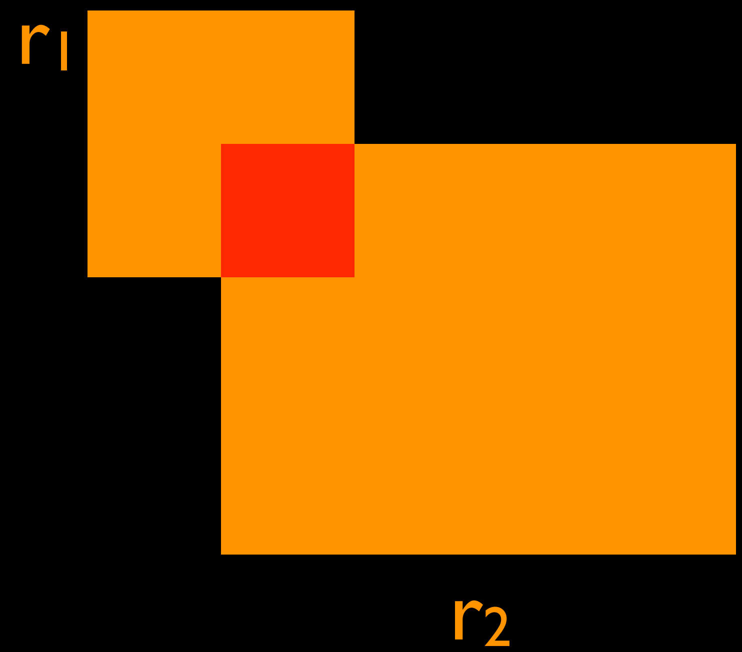


Intersection



Intersection

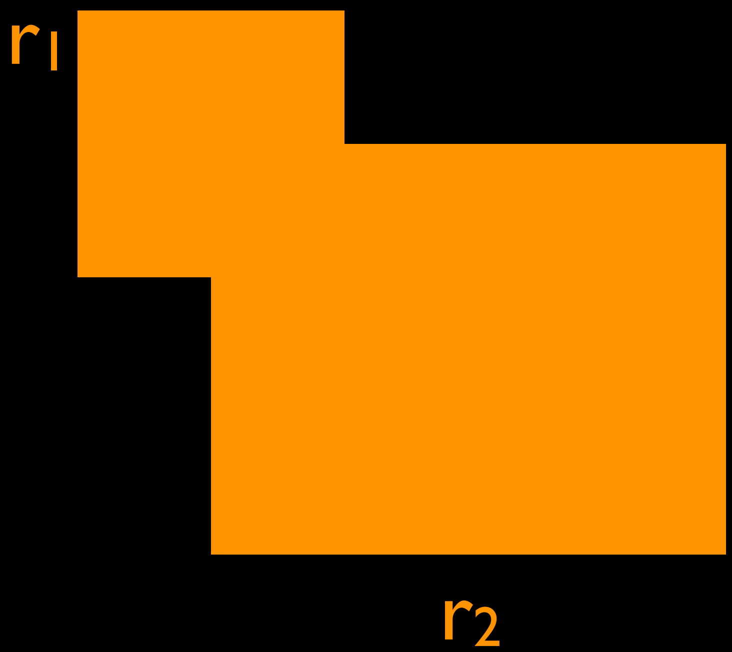
```
(/\) :: Region
  -> Region
  -> Region
r1 /\ r2 = \p ->
   r1 p && r2 p
```



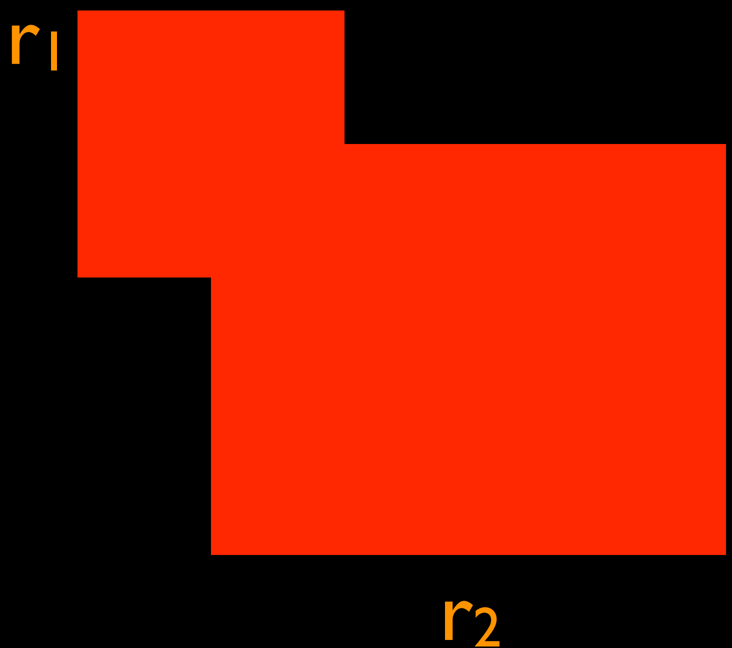
Union

Union

Union

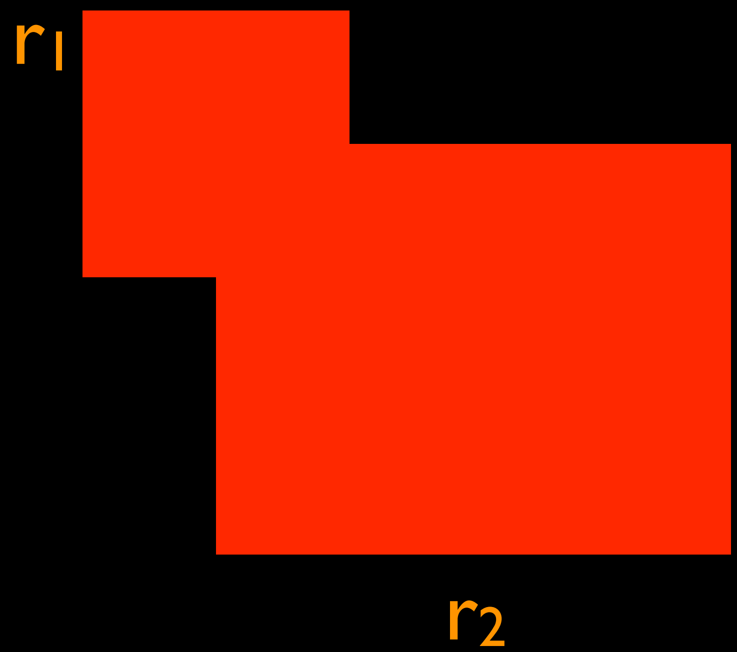


Union

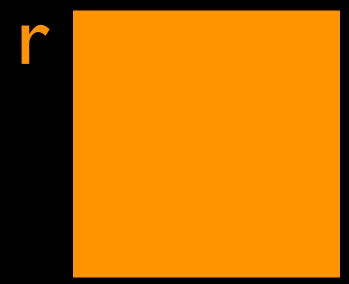


Union

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



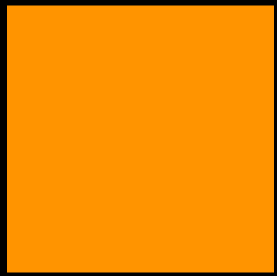
Negation



Negation

Negation

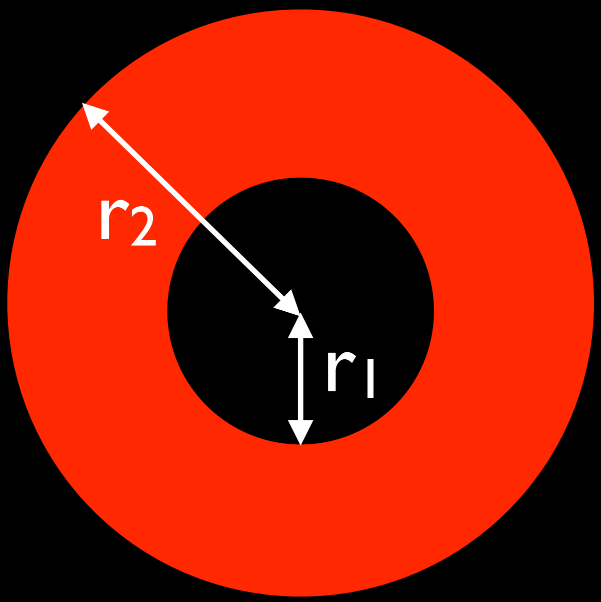
r



Negation

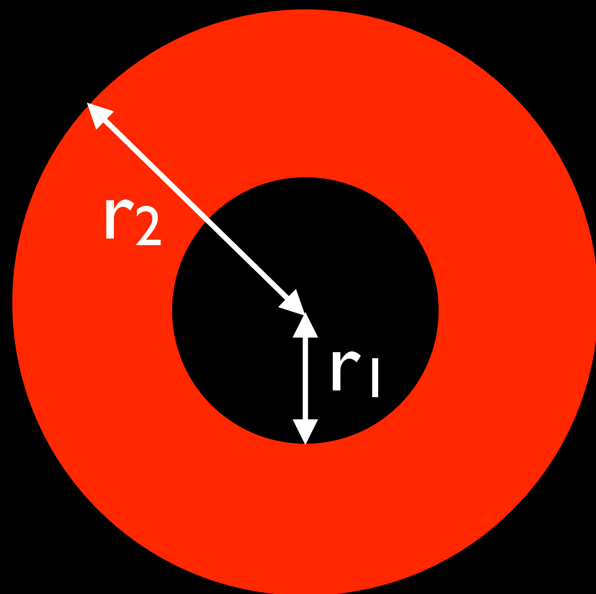

```
outside ::
  Region -> Region
outside r =
  \p -> not (r p)
```

Using combinators - annulus



Using combinators - annulus

```
annulus r1 r2 =
  outside (circle r1)
  /\ circle r2
```



NSWC

- The NSWC compared different the development time and lines of code necessary to write a geo-server in different languages.
- Haskell did really well.

Financial crisis

Financial derivatives

options

swaps

spreads

Financial derivatives

straddles

floors

options

swaptions

swaps

spreads

European options

Financial derivatives

straddles

caps

futures

captions

American options

floors

Financial contracts

- Would you rather:
  - Get 100 SEK now and give me 105 SEK in one month;
  - choose in one month to:
    - either pay me 700 SEK but receive 710 SEK in one month;
    - or receive 1000 SEK but pay me 1200 SEK in one year’s time.

Simple example

- The zero-coupon discount bond:

  ```
  zcb :: Date -> Double -> Currency -> Contract
  ```

  A contract `zcb t x k` means "receive `x` units of currency `k` on the date `t`"

- Should this be a primitive?

Even simpler examples

- The empty contract:

```
empty :: Contract
```

- The receive one unit now:

```
one :: Currency -> Contract
```

Combinators

- Combine two contracts:

```
and :: Contract -> Contract -> Contract
```

- Choose between contracts:

```
or :: Contract -> Contract -> Contract
```

- Reverse a contract:

```
give :: Contract -> Contract
```

Problems

- How do we deal with
  - dates?
  - currency fluctuations?
  - interest rates?
  - weather forecasts?
  - ....

Observables

- We need to describe the set of values that are not known statically, but influence the value of a contract.
- Examples:
  - `today :: Obs Date`
  - `sekTogbp :: Obs Double`
  - `mmRainInCorfu :: Obs Int`

Observables

- I'm going to assume a fixed set of observables, implemented by someone else:
- but present functions to manipulate them;
- and functions to use them to construct contracts.

Observable combinators

- Constants:

```
constant :: a -> Obs a
```

- Choose between contracts:

```
lift :: (a -> b) -> Obs a -> Obs b
lift2 :: (a -> b -> c) ->
  Obs a -> Obs b -> Obs c
```

Example:

- More than three centimeters of rain in Corfu:

```
lift2 (>) rainInCorfu (const 30)
```

- This gives a value of type `Obs Bool`

Using observables

- Scaling contracts:

```
scale :: Obs Double
  -> Contract -> Contract
```

- Conditional contracts:

```
when :: Obs Bool
  -> Contract -> Contract
```

- And several others...

Zero-coupon discount bonds revisited

- We can now describe the zcb using these combinators:

```
at :: Date -> Obs Bool

at t = lift2 (==) date (const t)


zcb :: Date -> Double -> Currency -> Contract

zcb t x c = at t (scale (const x) (one c))
```

Review

- So far we have:
  - a language for describing contracts;
  - separated static structure from the observable values;
  - seen some simple examples.
- But how do we implement these functions?

Implementing contracts

```
data Contract =
    Zero
  | One
  | Give Contract
  | And Contract Contract
  | Scale (Obs Double) Contract
  | ....
```

AST

- We have a small language for describing financial contracts.
- The design of the contract language focussed on finding the right types.
- We can't do anything with contracts yet – we have only written down an abstract syntax tree.

Valuation

- Banks have complex stochastic *financial models* to try and predict the market's behaviour.
- To estimate a contract's value, we need to *compile* the contract AST to these models.
- Doing so requires lots of help from domain experts – but there's no more language design.

Shallow and deep

- DSELs come in two flavours:
  - the values of the DSL coincide with those of the host language (*shallow embedding)*;
  - the values of the DSL have an explicit representation in the host language (*deep embedding)*.

Deeply embedded regions

```
data Regions =
      Circle Float
    | Rectangle Float Float
    | And Region Region
    | ...
```

- **Pro:** add different semantics (like generating images, etc.)
- **Con:** more work/syntactic overhead.

Lessons

- Haskell's fancy features help:
  - higher order functions;
  - polymorphism;
  - algebraic data types;
  - type classes;
  - inobtrusive syntax.

Lots of DSELs

- Images;
- Music;
- Logic programming;
- Parsing;
- Pretty printers
- Data base queries;
- Hardware design;
- Automatic testing;
- Animation;
- Diagrams;
- Tree traversals;
- Web formlets....

Really wacky ones...

```
main = runBASIC $ do

  10 LET x =: 1

  20 PRINT “Hello Basic world!”

  30 LET X =: X + 1

  40 IF X <> 11 THEN 20

  50 END
```

Further reading

- *Haskell vs. Ada vs. Awk vs. ... An Experiment in Software Prototyping Productivity.* Paul Hudak and Mark Jones.
- *Composing contracts: an adventure in financial engineering.* Simon Peyton Jones, Jean Marc Eber, and Julian Seward.
- *The Fun of Programming.* Edited by Jeremy Gibbons and Oege de Moor.