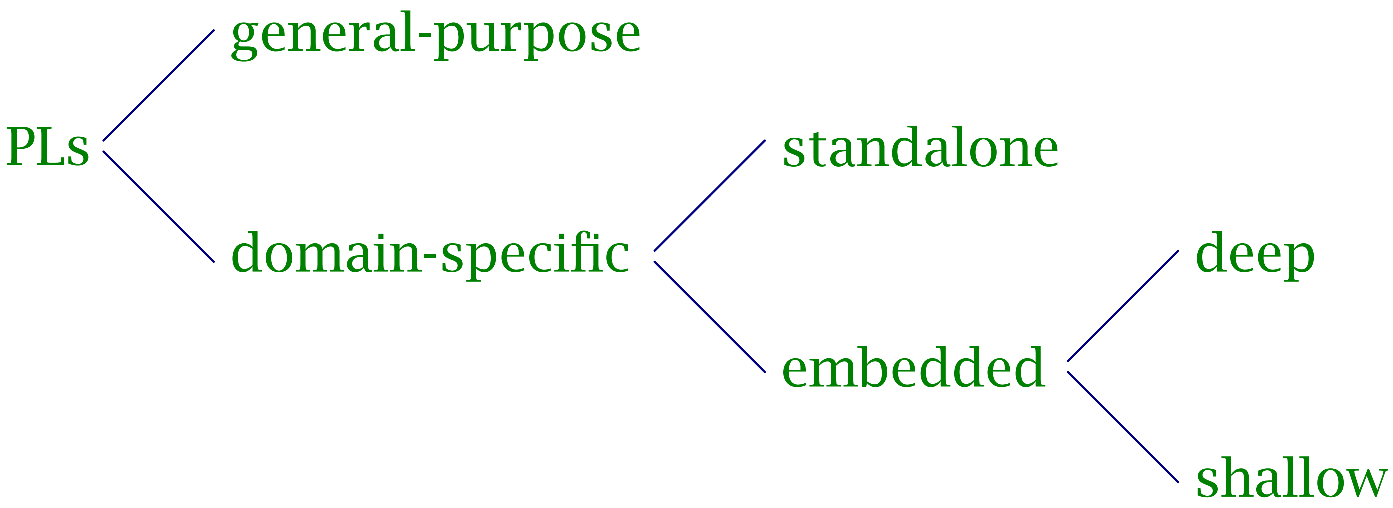
# Folding Domain-Specific Languages: Deep and Shallow Embeddings

*Jeremy Gibbons, University of Oxford*

*AiPL, Heriot Watt, August 2014*

# 1. Context




Embedded DSLs seem to be most popular in FP; cf OO. Why is that?

- *algebraic datatypes:* lightweight definitions of tree-shaped data
- *higher-order functions:* programs parametrized by other programs

See my papers in CEFP 2013 and ICFP 2014 for more.

# 2. Algebraic datatypes for DSLs

Deep embedding centred around ASTs.

*Lightweight algebraic datatypes* an essential feature:

- observers inductively defined over structure
- optimizations and transformations via tree manipulation

(Incidentally, algebraic datatypes also very convenient as a marshalling format for interoperation.)

## 2.1. A simple language

A *deeply embedded* expression language:

```
data ExprD :: * where
  Val :: Integer         → ExprD
  Add :: ExprD → ExprD → ExprD
```

For example, the expression $3 + (4 + 5)$ is represented by the term

```
expr :: ExprD
expr = Add (Val 3) (Add (Val 4) (Val 5))
```

```
   +
  / \
 3   +
    / \
   4   5
```

## 2.2. One semantics

To evaluate an *ExprD*, yielding an *Integer*:

$$
\begin{aligned}
&evalD :: ExprD \to Integer \\
&evalD\ (Val\ n) \quad\ \ = n \\
&evalD\ (Add\ x\ y) = evalD\ x + evalD\ y
\end{aligned}
$$

so $evalD\ expr = 12$.

## 2.3. Another semantics

To print an *ExprD*, yielding a *String*:

$$
\begin{aligned}
&\mathit{printD} :: \mathit{ExprD} \to \mathit{String} \\
&\mathit{printD}\ (\mathit{Val}\ n) \quad\ \ = \mathit{show}\ n \\
&\mathit{printD}\ (\mathit{Add}\ x\ y) = \mathit{paren}\ (\mathit{printD}\ x +\!\!+ \texttt{"+"} +\!\!+ \mathit{printD}\ y)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{paren} :: \mathit{String} \to \mathit{String} \\
&\mathit{paren}\ s = \texttt{"("} +\!\!+ s +\!\!+ \texttt{")"}
\end{aligned}
$$

so $\mathit{printD}\ \mathit{expr} = \texttt{"(3+(4+5))"}$.

## 2.4. Deep embedding—summary

- syntax of language represented by *algebraic datatypes*
- semantics expressed by *recursive functions*
- easy to provide multiple semantics

# 3. Shallow embedding

Here's an alternative representation of expressions: as their evaluation.

$$\textbf{type}\ \mathit{ExprS}_1 = \mathit{Integer}$$

$$\mathit{val}_1 :: \mathit{Integer} \to \mathit{ExprS}_1$$
$$\mathit{val}_1\ n = n$$

$$\mathit{add}_1 :: \mathit{ExprS}_1 \to \mathit{ExprS}_1 \to \mathit{ExprS}_1$$
$$\mathit{add}_1\ x\ y = x + y$$

$$\mathit{exprS}_1 :: \mathit{ExprS}_1$$
$$\mathit{exprS}_1 = \mathit{add}_1\ (\mathit{val}_1\ 3)\ (\mathit{add}_1\ (\mathit{val}_1\ 4)\ (\mathit{val}_1\ 5))$$

Now the evaluation semantics is easy:

$$\mathit{evalS}_1 :: \mathit{ExprS}_1 \to \mathit{Integer}$$
$$\mathit{evalS}_1\ x = x \qquad \text{-- !}$$

The syntax has been discarded; *only semantics* is left.

## 3.1. Another shallow embedding

—this time, under the *print* interpretation:

$$\textbf{type}\ \mathit{ExprS}_2 = \mathit{String}$$

$$\mathit{val}_2 :: \mathit{Integer} \to \mathit{ExprS}_2$$
$$\mathit{val}_2\ n = \mathit{show}\ n$$

$$\mathit{add}_2 :: \mathit{ExprS}_2 \to \mathit{ExprS}_2 \to \mathit{ExprS}_2$$
$$\mathit{add}_2\ x\ y = \mathit{paren}\ (x +\!\!+ \texttt{"+"} +\!\!+ y)$$

For example,

$$\mathit{exprS}_2 :: \mathit{ExprS}_2$$
$$\mathit{exprS}_2 = \mathit{add}_2\ (\mathit{val}_2\ 3)\ (\mathit{add}_2\ (\mathit{val}_2\ 4)\ (\mathit{val}_2\ 5))$$

Again, the semantics is trivial:

$$\mathit{printS}_2 :: \mathit{ExprS}_2 \to \mathit{String}$$
$$\mathit{printS}_2\ x = x \qquad \text{-- !}$$

## 3.2. Deep versus shallow embedding

Deep:

- syntax of language represented by algebraic datatypes
- semantics expressed by recursive functions
- easy to provide multiple interpretations

Shallow:

- no explicit representation of syntax, *only semantics*
- *no separate 'observers'* required
- but what about multiple interpretations?

# 4. Higher-order functions for DSLs

What about both interpretations at once, with a shallow embedding?

$$\textbf{type}\ ExprS_3 = (Integer, String)$$

$$evalS_3 :: ExprS_3 \rightarrow Integer$$
$$evalS_3\ (n, s) = n$$

$$printS_3 :: ExprS_3 \rightarrow String$$
$$printS_3\ (n, s) = s$$

$$val_3 :: Integer \rightarrow ExprS_3$$
$$val_3\ n = (n, show\ n)$$

$$add_3 :: ExprS_3 \rightarrow ExprS_3 \rightarrow ExprS_3$$
$$add_3\ x\ y = (evalS_3\ x + evalS_3\ y, paren\ (printS_3\ x + \!\!+ \texttt{"+"} + \!\!+ printS_3\ y))$$

Note that with lazy evaluation, if only one interpretation is demanded then only that one will be computed.

But with three interpretations? Ten? Unforeseen interpretations?

## 4.1. What makes an interpretation?

What do the different interpretations have in common?
More importantly, how do they differ?

- a *semantic domain*
- an *interpretation of values* in this domain (a function)
- an *interpretation of addition* in this domain (a binary operator)

So let's capture these varying ingredients:

$$\textbf{type}\ \mathit{ExprAlg}\ a = (\mathit{Integer} \rightarrow a, a \rightarrow a \rightarrow a)$$

Mathematically, the ingredients of an interpretation are an 'algebra'.

## 4.2. Parametrized interpretation of shallow embedding

Now, a term is represented as a *parametrized interpretation*:
if you tell it how to interpret, it will give you back the interpretation.

$$\textbf{type}\ ExprS\ a = ExprAlg\ a \to a$$

$$\begin{aligned} &valS :: Integer \to ExprS\ a \\ &valS\ n = \lambda(f, g) \to f\ n \\ &addS :: ExprS\ a \to ExprS\ a \to ExprS\ a \\ &addS\ x\ y = \lambda(f, g) \to g\ (x\ (f, g))\ (y\ (f, g)) \end{aligned}$$

Provides the same surface syntax as before; for example,

$$\begin{aligned} &exprS :: ExprS\ a \\ &exprS = addS\ (valS\ 3)\ (addS\ (valS\ 4)\ (valS\ 5)) \end{aligned}$$

## 4.3. Instantiating the parametrized interpretation

It's quite general—given

$$\begin{aligned} &\mathit{evalAlg} :: \mathit{ExprAlg}\ \mathit{Integer} \\ &\mathit{evalAlg} = (\mathit{id}, (+)) \end{aligned}$$

$$\begin{aligned} &\mathit{printAlg} :: \mathit{ExprAlg}\ \mathit{String} \\ &\mathit{printAlg} = (\mathit{show}, \lambda s\ t \rightarrow \mathit{paren}\ (s \mathbin{+\!\!+} \texttt{"+"} \mathbin{+\!\!+} t)) \end{aligned}$$

we have:

$$\begin{aligned} &\mathit{exprS}\ \mathit{evalAlg} &&= 12 \\ &\mathit{exprS}\ \mathit{printAlg} &&= \texttt{"(3+(4+5))"} \end{aligned}$$

## 4.4. Church encoding

Where did *ExprAlg* come from?

Consider fold function for *ExprD* algebraic datatype:

$$
\begin{aligned}
&fold :: (Integer \to a, a \to a \to a) \to ExprD \to a\\
&fold\ (f, g)\ (Val\ n) \quad\ \ = f\ n\\
&fold\ (f, g)\ (Add\ x\ y) = g\ (fold\ (f, g)\ x)\ (fold\ (f, g)\ y)
\end{aligned}
$$

Swap the arguments around:

$$
\begin{aligned}
&flipFold :: ExprD \to (Integer \to a, a \to a \to a) \to a\\
&\qquad \text{-- equivalently, } ExprD \to (\forall a.ExprAlg\ a \to a)\\
&flipFold\ (Val\ n) \quad\ \ (f, g) = f\ n\\
&flipFold\ (Add\ x\ y)\ (f, g) = g\ (flipFold\ x\ (f, g))\ (flipFold\ y\ (f, g))
\end{aligned}
$$

This is known as the *Church* (or *Böhm–Berarducci*) encoding of *expr*, and type $\forall a.ExprAlg\ a$ as the encoding of datatype *Expr*.

## 4.5. Polymorphic interpretation of shallow embedding

Alternatively, using *type classes* (poor person's modules):

```
class ExprC a where
  valC :: Integer → a
  addC :: a → a → a
```

Interpretations at *Integer* and *String* types:

```
instance ExprC Integer where
  valC n     = n
  addC x y = x + y
instance ExprC String where
  valC n     = show n
  addC x y = paren (x ++ "+" ++ y)
```

Then DSL term has polymorphic type:

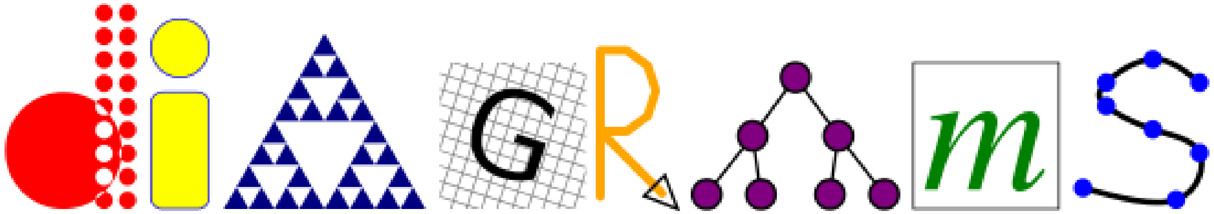
$$exprC :: ExprC\ a \Rightarrow a$$
$$exprC = addC\ (valC\ 3)\ (addC\ (valC\ 4)\ (valC\ 5))$$

and can be interpreted at any type in the type class *Expr*:

$$evalExpr :: Integer$$
$$evalExpr = exprC$$
$$printExpr :: String$$
$$printExpr = exprC$$

# 5. Exercises: Diagrams

Embedded DSL for vector graphics, inspired by Brent Yorgey's



(http://projects.haskell.org/diagrams/)

We'll build a simpler language in the same style.

## 5.1. Shapes

Deep embedding:

```
data Shape
    = Rectangle Double Double   -- width, height
    | Ellipse Double Double     -- xradius, yradius
    | Triangle Double           -- side length (equilateral)
```

Not very exciting, because not recursive.

## 5.2. Styles

```
type StyleSheet = [Styling]
data Styling
    = FillColour Col
    | StrokeColour Col
    | StrokeWidth Double
data Col = Red | Blue | Bisque | ...   -- and many more!
```

Default is for no fill, and very thin black strokes.

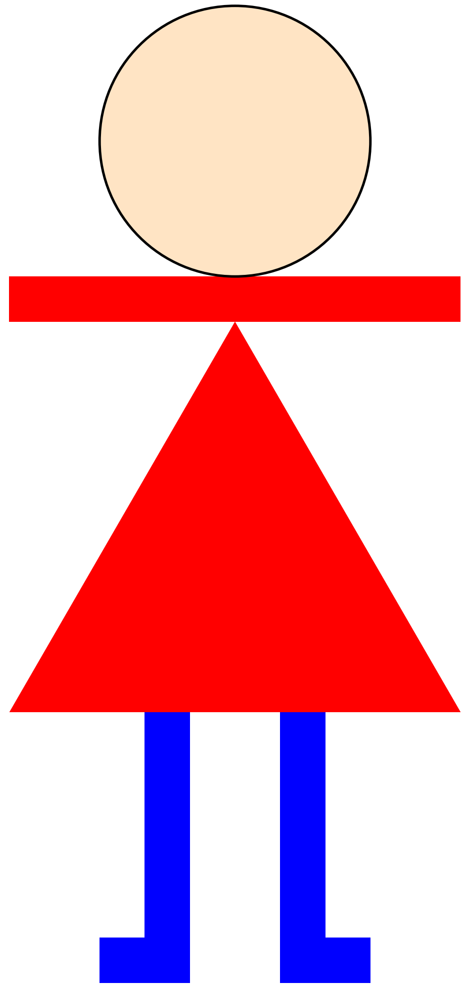
## 5.3. Pictures

$$
\begin{aligned}
&\textbf{data}\ \mathit{Picture} \\
&\quad = \mathit{Place}\ \mathit{StyleSheet}\ \mathit{Shape} \\
&\quad \mid\ \mathit{Above}\ \mathit{Picture}\ \mathit{Picture} \\
&\quad \mid\ \mathit{Beside}\ \mathit{Picture}\ \mathit{Picture}
\end{aligned}
$$

Alignment is by centres.

## 5.4. Red dress and blue stockings

```
figure :: Picture
figure =
  Place [StrokeWidth 0.1, FillColour bisque]
          (Ellipse 3 3) `Above`
  Place [FillColour red, StrokeWidth 0]
          (Rectangle 10 1) `Above`
  Place [...] (Triangle 10) `Above`
  (Place [...] (Rectangle 1 5) `Beside`
   Place [StrokeWidth 0] (Rectangle 2 5) `Beside`
   Place [...] (Rectangle 1 5)) `Above`
  (Place ... `Beside` ...)
```



(Note blank rectangle.)

## 5.5. Transformations

To align pictures, we'll need to translate them.

$$\mathbf{type}\ \mathit{Pos} = \mathit{Complex}\ \mathit{Double}$$

$$\begin{aligned}
&\mathbf{data}\ \mathit{Transform}\\
&\quad = \mathit{Identity}\\
&\quad \mid \mathit{Translate}\ \mathit{Pos}\\
&\quad \mid \mathit{Compose}\ \mathit{Transform}\ \mathit{Transform}
\end{aligned}$$

We represent 2D point $(x, y)$ by Haskell $(x :+ y) :: \mathit{Complex}\ \mathit{Double}$.

$$\begin{aligned}
&\mathit{transformPos} :: \mathit{Transform} \to \mathit{Pos} \to \mathit{Pos}\\
&\mathit{transformPos}\ \mathit{Identity} &&= \mathit{id}\\
&\mathit{transformPos}\ (\mathit{Translate}\ p) &&= (p+)\\
&\mathit{transformPos}\ (\mathit{Compose}\ t\ u) &&= \mathit{transformPos}\ t \circ \mathit{transformPos}\ u
\end{aligned}$$

This is a deep embedding. How about shallow?

## 5.6. Simplified pictures

$$\textbf{type}\ \mathit{Drawing} = [\,(\mathit{Transform}, \mathit{StyleSheet}, \mathit{Shape})\,] \quad \text{-- centred on origin}$$

$$\textbf{type}\ \mathit{Extent} = (\mathit{Pos}, \mathit{Pos}) \quad \text{-- (lower left, upper right)}$$

$$\begin{array}{l}
\mathit{unionExtent} :: \mathit{Extent} \to \mathit{Extent} \to \mathit{Extent} \\
\mathit{unionExtent}\ (\mathit{llx}_1 :\!+\ \mathit{lly}_1, \mathit{urx}_1 :\!+\ \mathit{ury}_1)\ (\mathit{llx}_2 :\!+\ \mathit{lly}_2, \mathit{urx}_2 :\!+\ \mathit{ury}_2) \\
\quad = (\mathit{min}\ \mathit{llx}_1\ \mathit{llx}_2 :\!+\ \mathit{min}\ \mathit{lly}_1\ \mathit{lly}_2, \mathit{max}\ \mathit{urx}_1\ \mathit{urx}_2 :\!+\ \mathit{max}\ \mathit{ury}_1\ \mathit{ury}_2)
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{shapeExtent} :: \mathit{Shape} \to \mathit{Extent}} \\
\mathit{shapeExtent}\ (\mathit{Ellipse}\ \mathit{xr}\ \mathit{yr}) & = (-(\mathit{xr} :\!+\ \mathit{yr}), \mathit{xr} :\!+\ \mathit{yr}) \\
\mathit{shapeExtent}\ (\mathit{Rectangle}\ w\ h) & = (-({}^{w}/_{2} :\!+\ {}^{h}/_{2}), {}^{w}/_{2} :\!+\ {}^{h}/_{2}) \\
\mathit{shapeExtent}\ (\mathit{Triangle}\ s) & = (-({}^{s}/_{2} :\!+\ \sqrt{3} \times {}^{s}/_{4}), {}^{s}/_{2} :\!+\ \sqrt{3} \times {}^{s}/_{4})
\end{array}$$

$$\begin{array}{l}
\mathit{drawingExtent} :: \mathit{Drawing} \to \mathit{Extent} \\
\mathit{drawingExtent} = \mathit{foldr}1\ \mathit{unionExtent} \circ \mathit{map}\ \mathit{getExtent}\ \textbf{where} \\
\quad \mathit{getExtent}\ (t, _, s) = \textbf{let}\ (\mathit{ll}, \mathit{ur}) = \mathit{shapeExtent}\ s \\
\qquad\qquad\qquad\qquad\quad \textbf{in}\ (\mathit{transformPos}\ t\ \mathit{ll}, \mathit{transformPos}\ t\ \mathit{ur})
\end{array}$$

## 5.7. Simplifying pictures

$$
\begin{aligned}
&\mathit{drawPicture} :: \mathit{Picture} \to \mathit{Drawing}\\
&\mathit{drawPicture}\ (\mathit{Place}\ u\ s) \quad = \mathit{drawShape}\ u\ s\\
&\mathit{drawPicture}\ (\mathit{Above}\ p\ q) = \mathit{drawPicture}\ p\ \text{`}\mathit{aboveD}\text{`}\ \mathit{drawPicture}\ q\\
&\mathit{drawPicture}\ (\mathit{Beside}\ p\ q) = \mathit{drawPicture}\ p\ \text{`}\mathit{besideD}\text{`}\ \mathit{drawPicture}\ q
\end{aligned}
$$

All the work is in the individual operations:

$$
\begin{aligned}
&\mathit{drawShape} :: \mathit{StyleSheet} \to \mathit{Shape} \to \mathit{Drawing}\\
&\mathit{aboveD}, \mathit{besideD} :: \mathit{Drawing} \to \mathit{Drawing} \to \mathit{Drawing}
\end{aligned}
$$

## 5.8. Simplifying pictures

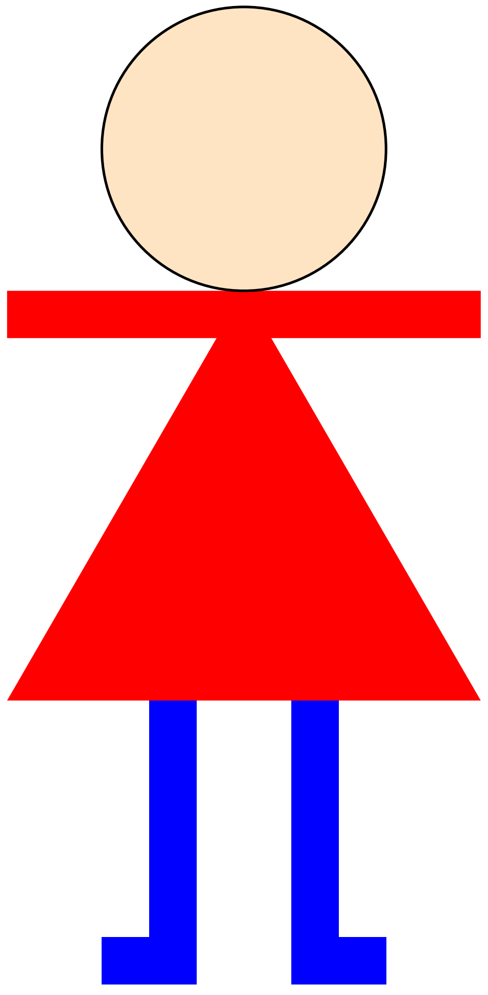
```
drawShape :: StyleSheet → Shape → Drawing
drawShape u s = [ (Identity, u, s) ]

aboveD, besideD :: Drawing → Drawing → Drawing
pd `aboveD` qd = transformDrawing (Translate (0 :+ qury)) pd ++
                 transformDrawing (Translate (0 :+ plly)) qd where
  (pllx :+ plly, pur)  = drawingExtent pd
  (qll, qurx :+ qury)  = drawingExtent qd

pd `besideD` qd = transformDrawing (Translate (qllx :+ 0)) pd ++
                  transformDrawing (Translate (purx :+ 0)) qd where
  (pll, purx :+ pury) = drawingExtent pd
  (qllx :+ qlly, qur) = drawingExtent qd

transformDrawing :: Transform → Drawing → Drawing
transformDrawing t = map (λ(t′, u, s) → (Compose t t′, u, s))
```

## **5.9.** *InFrontOf*, *FlipV*



## 5.10. Generating SVG

Simple-minded XML—all markup, no content:

```
data XML = Element String [Attr] [XML]
type Attr = (String, String)

assemble :: Drawing → XML
assemble d = Element "svg" (drawingAttrs d)
  [Element "g" (groupAttrs d) (map diagramShape d)]

diagramShape :: (Transform, StyleSheet, Shape) → XML

writeSVG :: FilePath → XML → IO ()
writeSVG f ss = writeFile f (unlines ss)
```

(see code for details).

## 5.11. Transformations again

Two interpretations of deeply embedded *Transforms*:

$$\begin{aligned}
&\mathit{transformPos} :: \mathit{Transform} \rightarrow (\mathit{Pos} \rightarrow \mathit{Pos})\\
&\mathit{transformPos}\ \mathit{Identity} &&= \mathit{id}\\
&\mathit{transformPos}\ (\mathit{Translate}\ p) &&= (p+)\\
&\mathit{transformPos}\ (\mathit{Compose}\ t\ u) &&= \mathit{transformPos}\ t \circ \mathit{transformPos}\ u
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{transformDrawing} :: \mathit{Transform} \rightarrow (\mathit{Drawing} \rightarrow \mathit{Drawing})\\
&\mathit{transformDrawing}\ t = \mathit{map}\ (\lambda(t', u, s) \rightarrow (\mathit{Compose}\ t\ t', u, s))
\end{aligned}$$

Shallow embedding with two fixed observers?
Parametrized observer? Polymorphic observer?

## 5.12. Tiles

Extend *Shape* language with marked tiles:

```
type TileMarkings = [[Pos]]
data Picture = ... | Tile Double TileMarkings
```

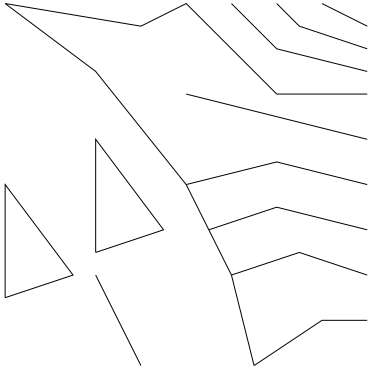
and *Transform* language with scaling and quarter-turns:

```
data Transform = ... | Scale Double | Rot
```

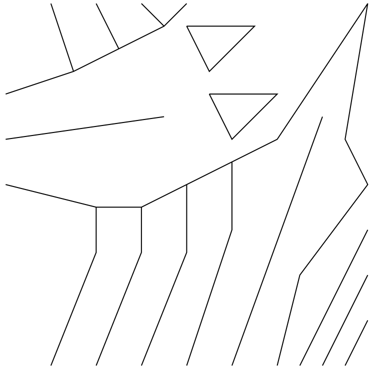
Some markings:

```
markingsP :: TileMarkings
markingsP = [[(4 :+ 4), (6 :+ 0)],
             [(0 :+ 3), (3 :+ 4), (0 :+ 8), (0 :+ 3)],
             [(4 :+ 5), (7 :+ 6), (4 :+ 10), (4 :+ 5)],
             [(11 :+ 0), (10 :+ 4), (8 :+ 8), (4 :+ 13), (0 :+ 16)],
             [(11 :+ 0), (14 :+ 2), (16 :+ 2)] ...]
```

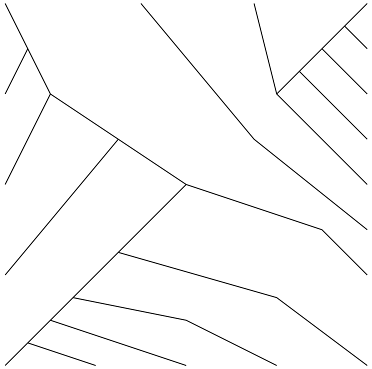
## 5.13. Four fish in boxes



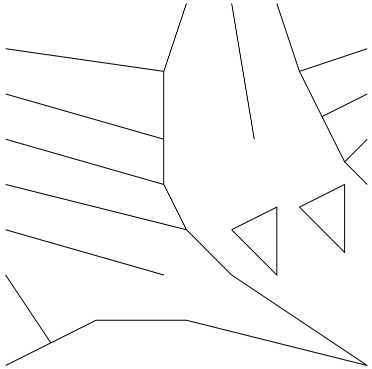
*fishP*



*fishQ*



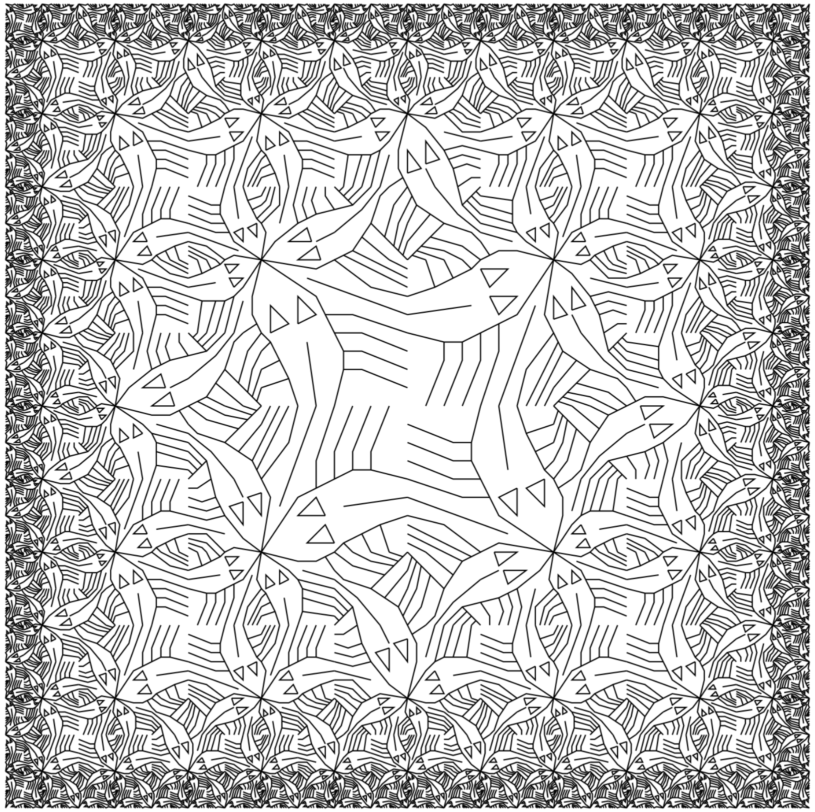
*fishR*

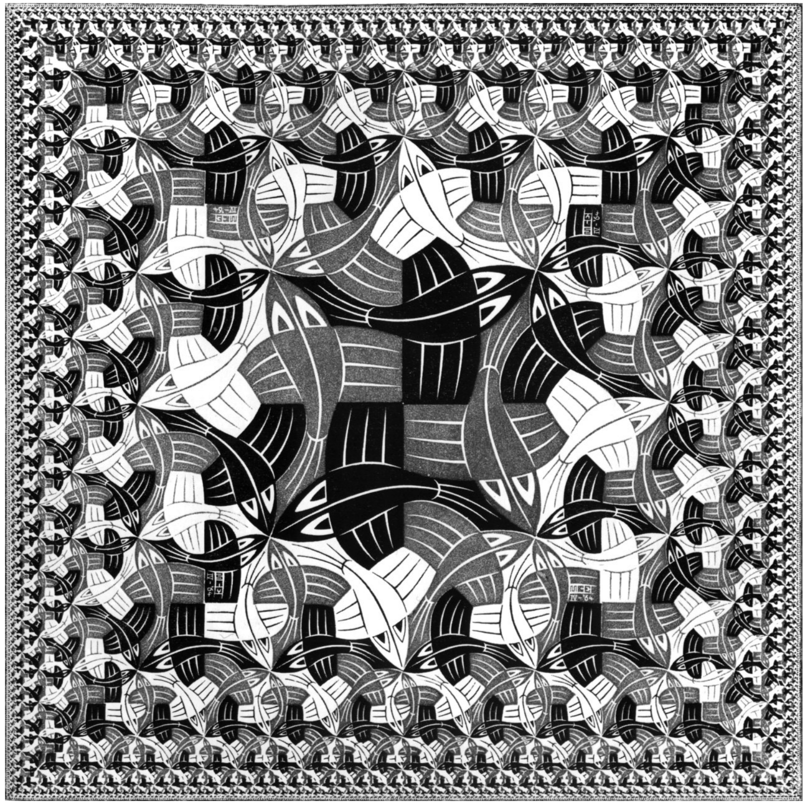


*fishS*

# 5.14. Square limit

With a little bit of scaling and rotation. . .





(After Henderson, *Functional Geometry*, 1982—after Escher, 1964.)