Generalizing turtle geometry

An extensible language for vector graphics drawing

Alice Rixte

LaBRI, University of Bordeaux

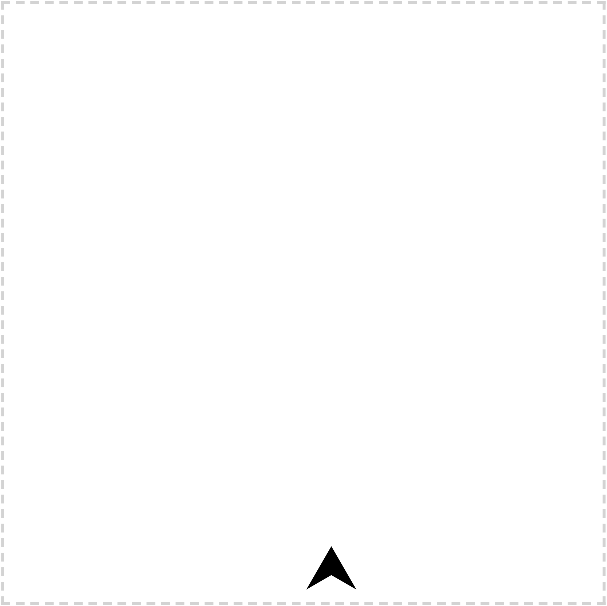
PhD Advisors: David Janin, Martin Laliberté
October 12, 2025
FARM, Singapore

Outline

1. **Introduction**
2. Pandia: A polymorphic language for writing multimedia DSLs
3. Implementation: the `Media` monad
4. Instanciation: a language for vector graphics

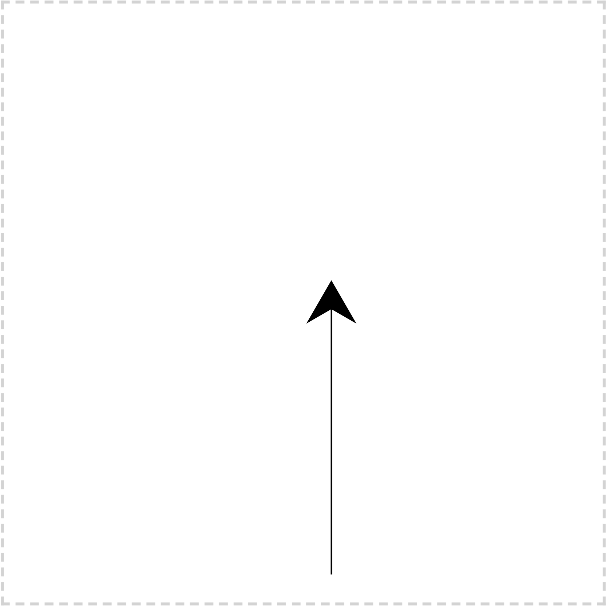
The turtle's original state

```
draw = return ()
```



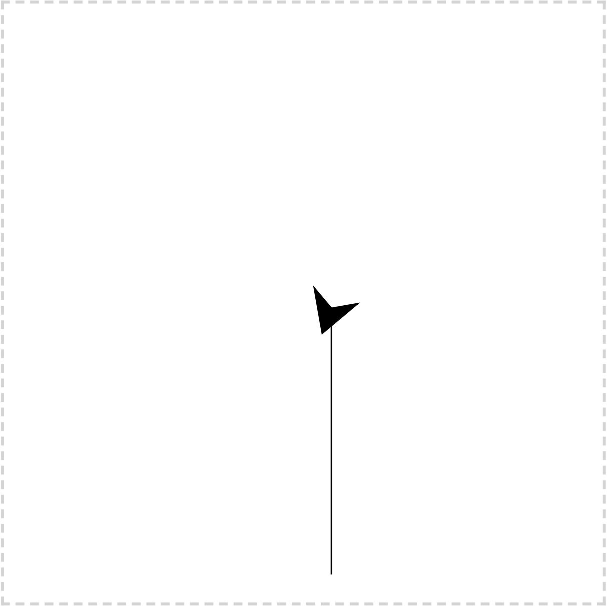
Going forward by 50 pixels

```
draw = do
  forward 50
```



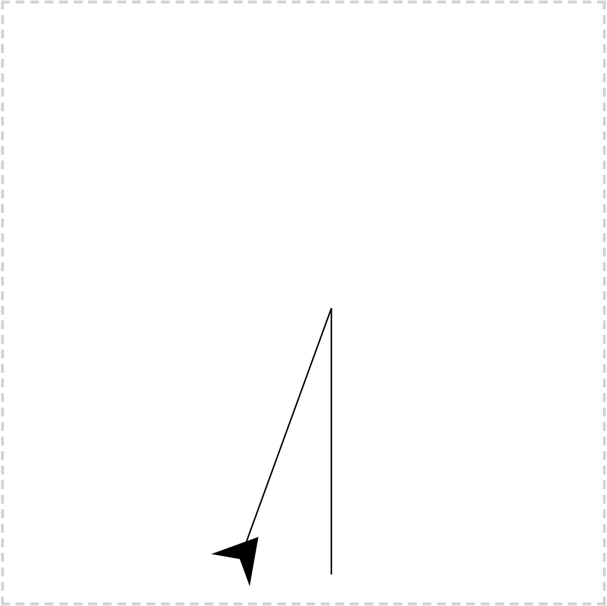
Turning left by 160°

```
draw = do
  forward 50
  left 160
```



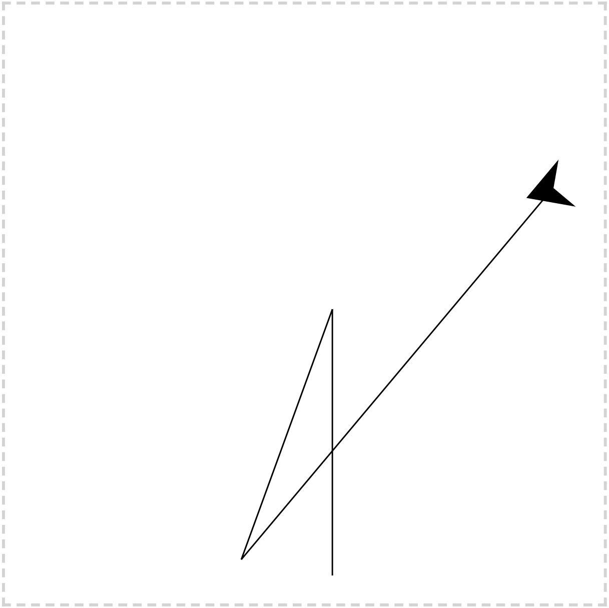
Forward by 50 pixels; left by 160°

```
draw = do
  forward 50
  left 160
  forward 50
  left 160
```



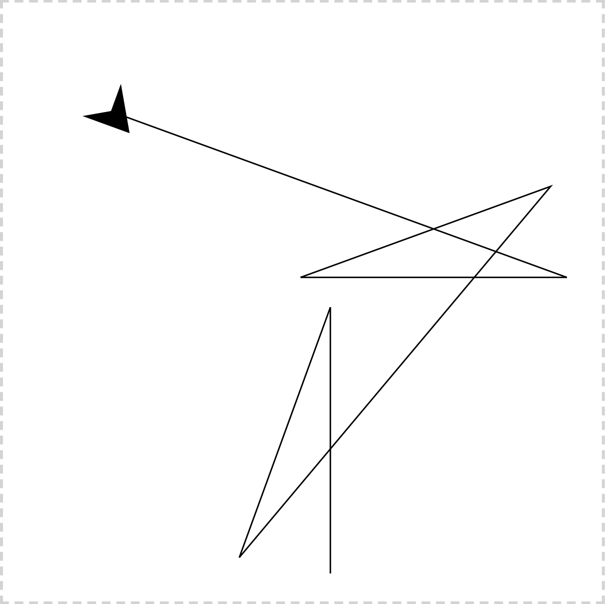
Let's give a name to our squiggle

```
squiggle = do
  forward 50
  left 160
  forward 50
  left 160
  forward 91
  left 150
```

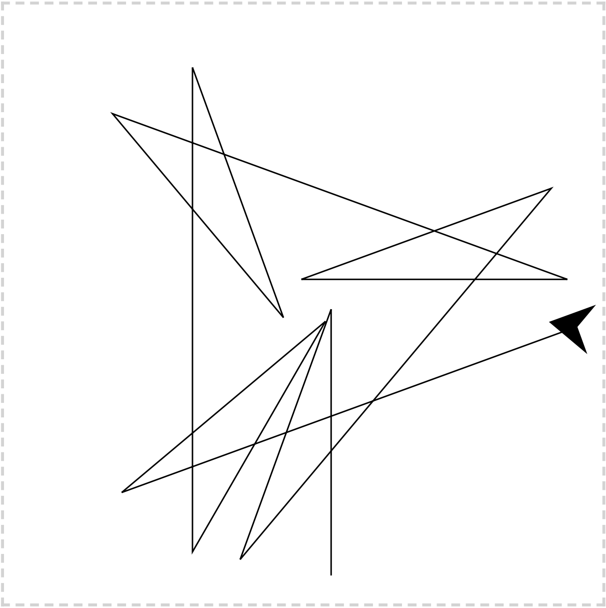


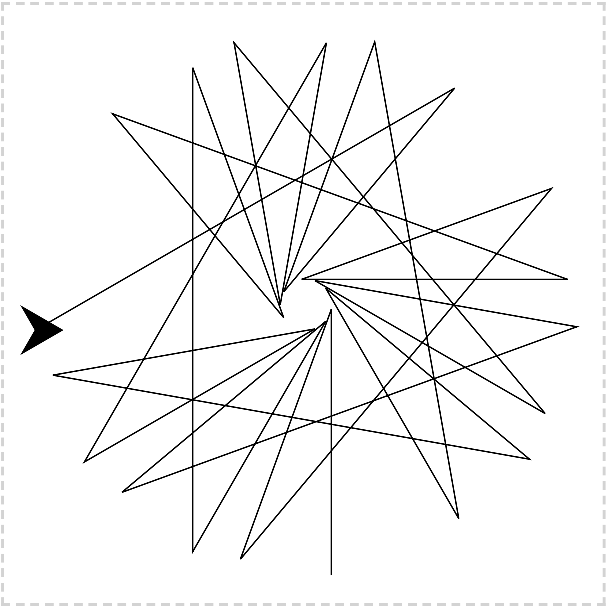
Twice the squiggle

```
draw = do
  squiggle
  squiggle
```



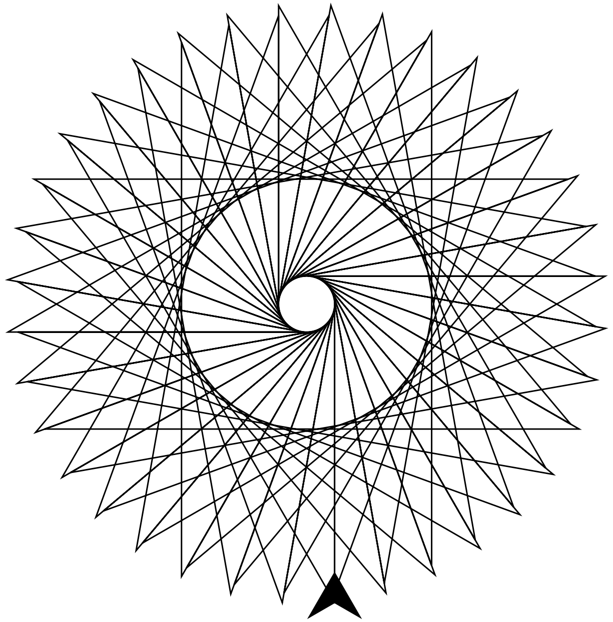
4 and 9 squiggles





36 squiggles

```
draw = repeat 36 squiggle
```



Extensions of turtle graphics

Turtle graphics can be used for 3D, 3D printing or music

Domain	Language
Images	Logo, Python Turtle
3D meshes	Octopus (David Janin, Simon Archipoff), Python 3D Turtle (Yasusi Kanada)
3D printing	Python 3D Turtle
Music	LMuse (David Sharp)

Common patterns

- A set of domain-specific *primitives*
- A *state* that represents a *multidimensional space* in which these primitives are placed.

Domain	Primitives	State
Logo	Segments	2D position, orientation, pen
Vector graphics	Shapes	Stroke color, stroke width, fill color …
3D	Meshes	3D affine matrices
Music	Notes	Pitch, onset time, duration …

The dream

What if we had a language that could be extended to any of the above specific domains?

This is precisely what I am trying to do in my PhD thesis!

Outline

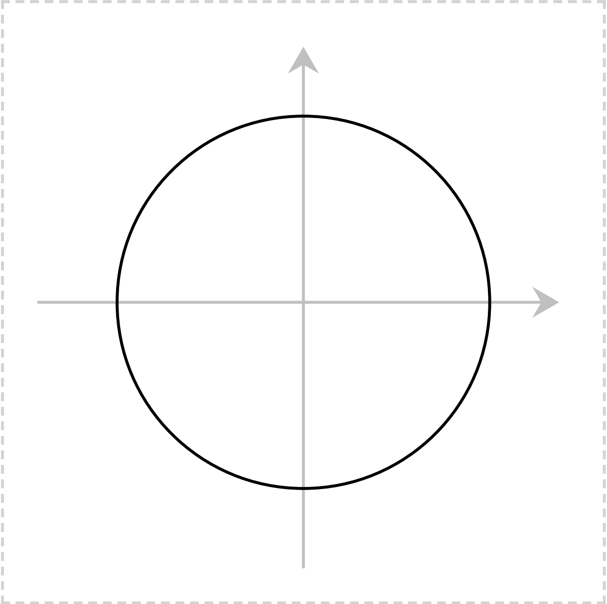
1. Introduction
2. **Pandia: A polymorphic language for writing multimedia DSLs**
    - The `prim` instruction
    - The `change` instruction
    - The `reset` instruction
    - The `transf` instruction
3. Implementation: the `Media` monad
4. Instanciation: a language for vector graphics

Pandia: the `prim` instruction

Renders a primitive

```
prim Circle
```

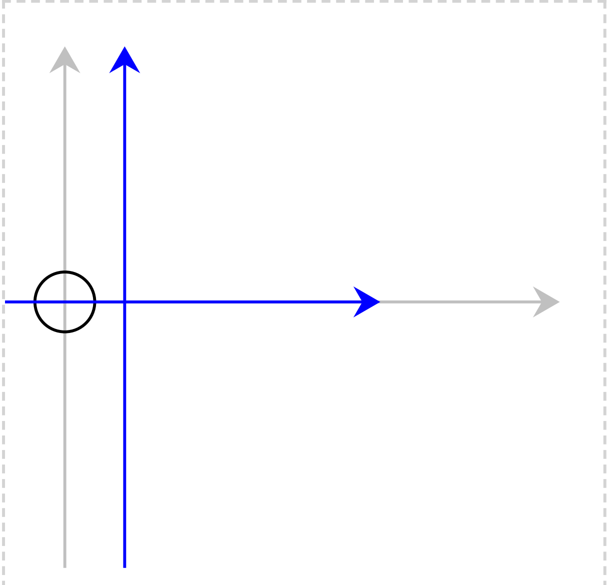
Draw a circle of diameter 1 centered at the origin.

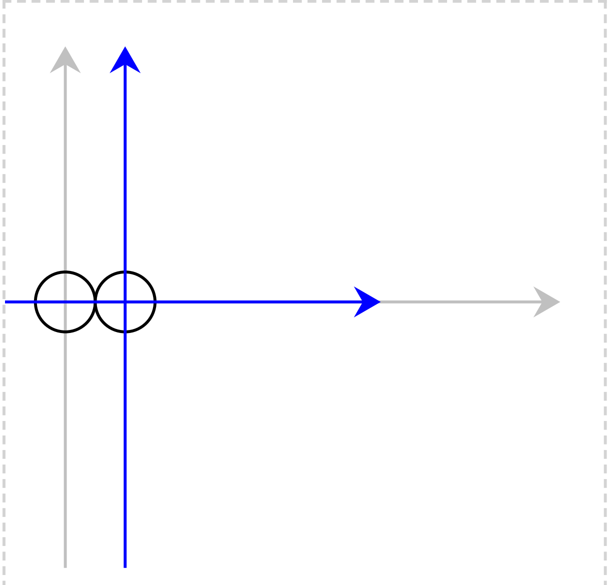


Pandia: the `change` instruction

```
prim Circle
change (moveX 1)
```

```
prim Circle
change (moveX 1)
prim Circle
```



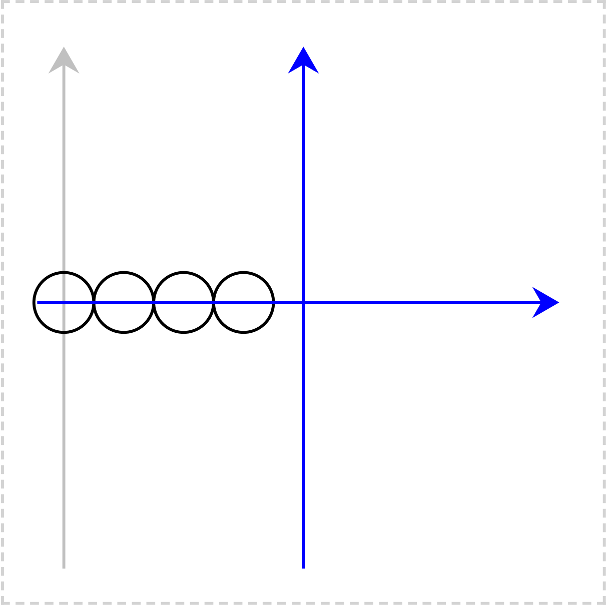


Pandia: the `change` instruction

```
circleLine n =
  repeat n $
    prim Circle
    change (moveX 1)
```

4 circles aligned vertically.

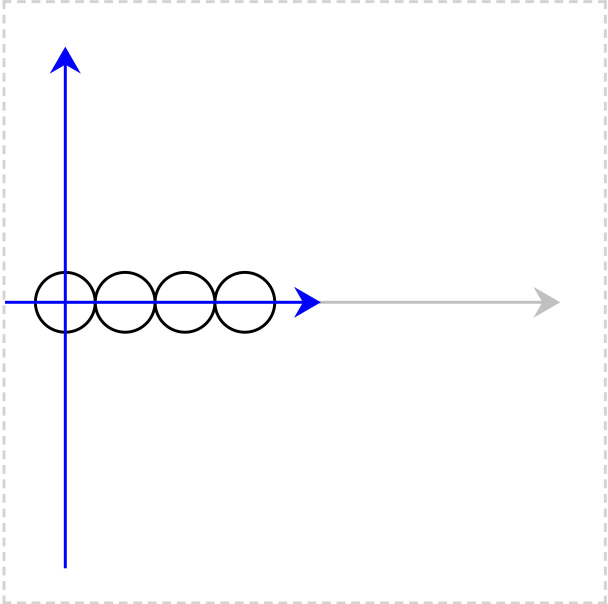
```
circleLine 4
```



Pandia: the `reset` instruction

Forgets the state changes within a scope

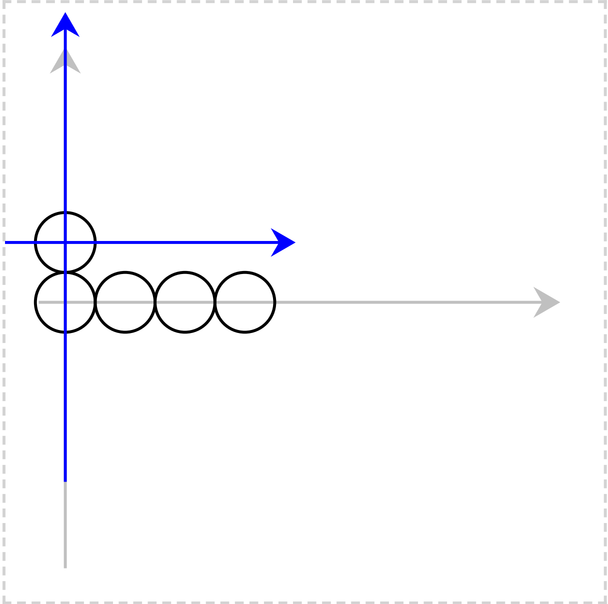
```
reset $ do
  repeat 4 $ do
    prim Circle
    change (moveX 1)
```



Pandia: the `reset` instruction

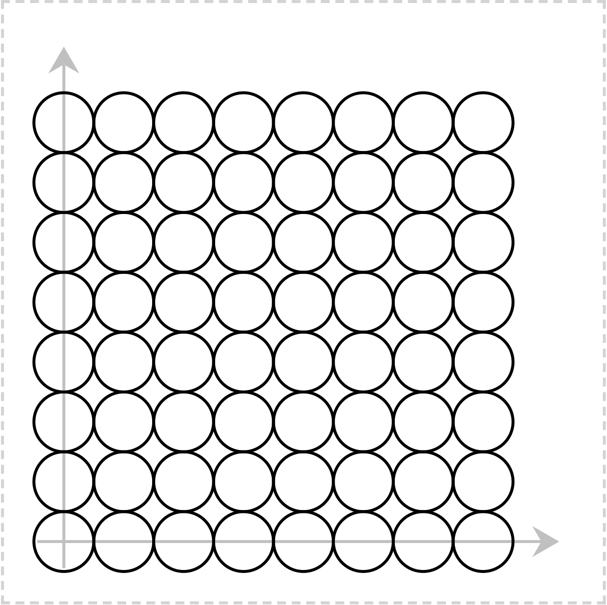
Forgets the state changes within a scope

```
reset $ do
  repeat 4 $ do
    prim Circle
    change (moveX 1)
change (moveY 1)
prim Circle
```



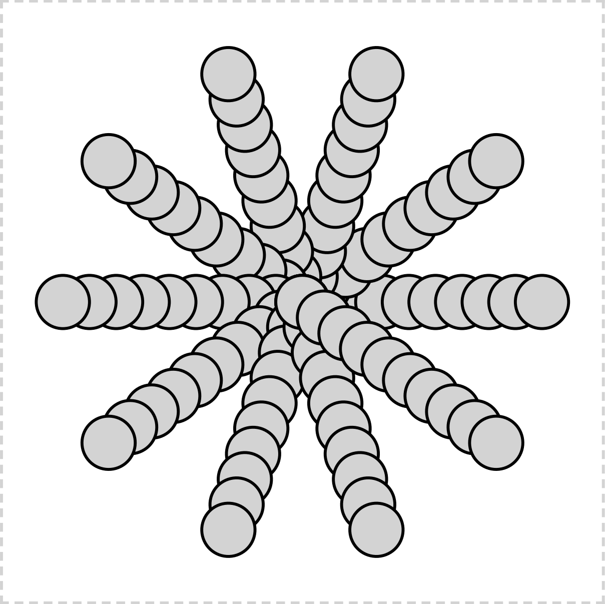
Pandia: the `reset` instruction

```
repeat 8 $ do
  reset $ do
    repeat 8 $ do
      prim Circle
      change (moveX 1)
  change (moveY 1)
```



More applications for reset

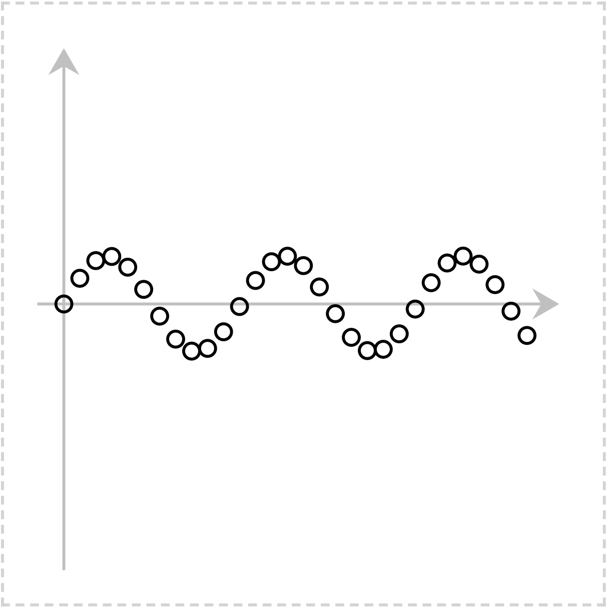
```
circleStar n =
  fill grey
  repeat n $ do
    reset $
      repeat n $
        prim Circle
        change (moveX 0.5)
    left (360 / n)
```



Pandia: the `transf` instruction

Applies a state change to all primitives within a scope

```
sinX s =
  s {s = s.y + sin s.x}

circles =
  transf sinX $
    repeat 30 $ do
      circle
      moveX 1
```



Outline

1. Introduction
2. Pandia: A polymorphic language for writing multimedia DSLs
3. **Polymorphic implementation: the media monad**
    - The media monad
    - The render monoid
    - The turtle state
    - Primitives
    - State changes
4. Instanciation into vector graphics

The media monad

```
newtype Media c s p w a = Media (c -> s -> (a,w,s))
```

```
prim :: p -> Media c s p w ()
change :: c -> Media c s p w ()

transf :: c -> Media c s p w a
            -> Media c s p w a

reset :: Media c s p w a
      -> Media c s p w a
```

- `c` the type of all **state changes**
- `s` the type of **states**
- `p` the primitives
- `w` the **rendering monoid**
- `a` the return value

The monoid of state changes

- To modify the state, we can use **state functions** `s -> s`
- Often, we only want to authorize a subset of **state changes** `c` of those functions, say linear transformations.
- We ask `c` to be a **monoid** such that there is a monoid morphism from `c` to `s -> s`, in other words `c` **acts on** `s`:

  ```
  class Monoid c => LActMn s c where
     lact :: c -> s -> s
  ```

  Monoid action laws:

  ```
  lact mempty s ≡ s
  lact (c1 <> c2) s  ≡ lact c1 (lact c2 s)
  ```

`change`

- The `change` instruction make a state change act on the current state.

```
change :: LActMn s c => c -> Media c s p w ()

change c = Media $ \ _ s ->
  ((), mempty, lact c s) -- state change acts on state
```

`reset`

- The `reset` instruction forgets about the state changes within its scope.

```
reset :: Media c s p w a -> Media c s p w a

reset (Media m) = Media $\ c s ->
    let (a, w, _) = m c s in  -- discard new state
        (a, w, s)             -- return old state
```

`transf`

- The `transf` instruction composes a state change to the context.

```haskell
transf :: Monoid c =>
  c -> Media c s p w a -> Media c s p w a

transf c1 (Media m) = Media $ \c2 s ->
  m (c2 <> c1) s --composes state change with context
```

`prim`

- Specify how to render a primitive given the current state:

```haskell
class RenderPrim s p w where
  renderPrim :: s -> p -> w
```

- The `prim` instruction make the context act on current state, and use the result to render the primitve.

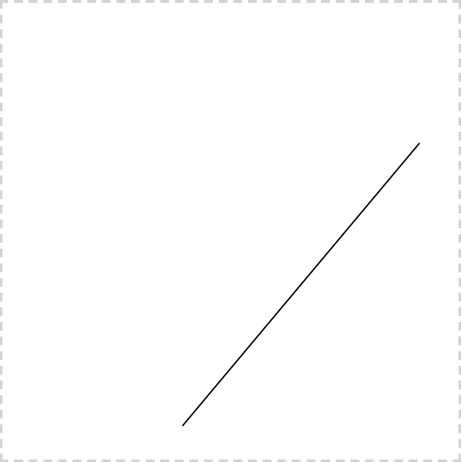
```haskell
prim :: (RenderPrim s p w, LActMn s c) =>
  p -> Media c s p w ()

prim p = Media $ \c s ->
  ((), renderPrim (lact c s) p, s)
  -- render a primitive and don't modify the state
```

Outline

1. Introduction
2. Pandia: A polymorphic language for writing multimedia DSLs
3. Polymorphic implementation: the media monad
4. **Instanciation into vector graphics**
    - Turtle graphics
    - Bézier paths
    - Vector graphics

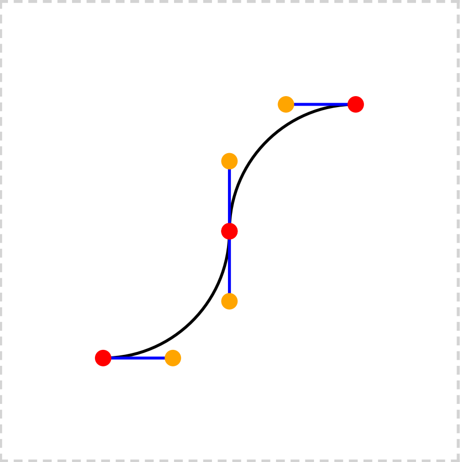
Primitives instanciations

Turtle graphics:



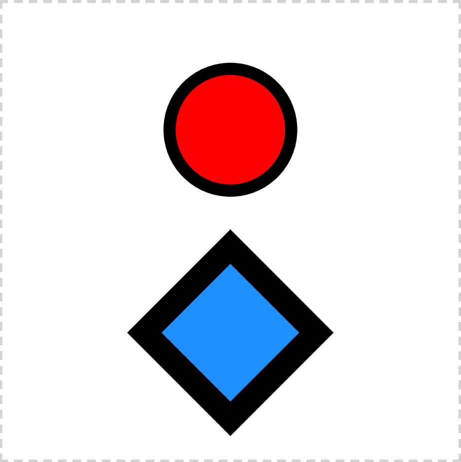
```
Segment = (V2 a, V2 a)
```

Bézier paths:



```
PathPoint =
        Anchor | Control
```

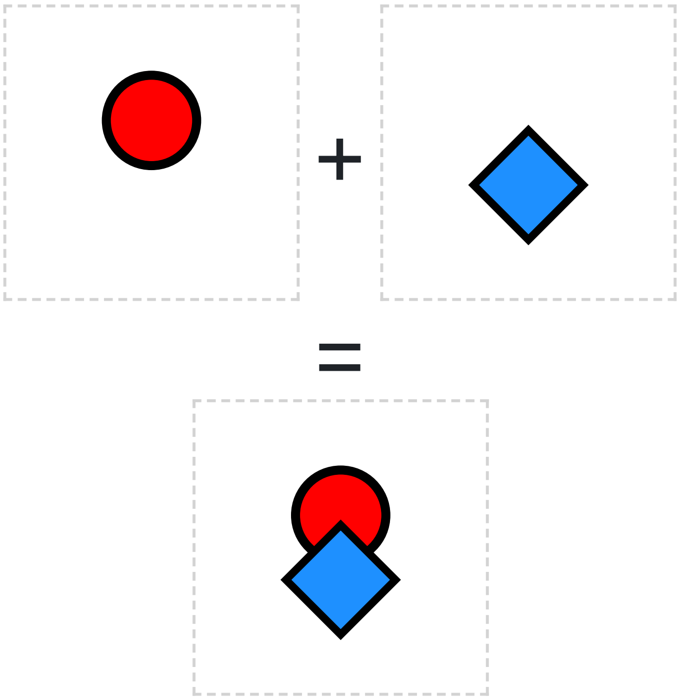
Vector graphics:



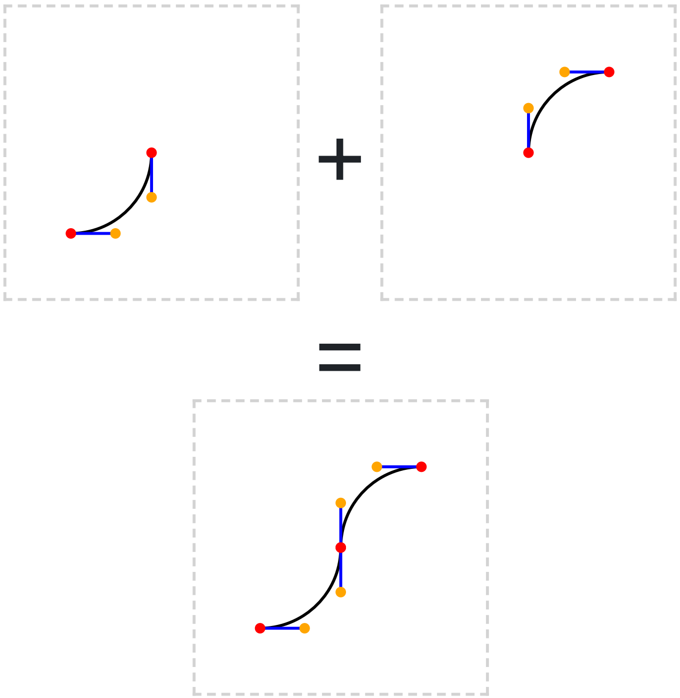
```
Path = [PathPoint]

Shape =
  Open Path | Closed Path
```

Rendering monoids

Image superimposition



Bézier path concatenation:



State instanciation

Let `s` be the set of all possible turtle states.

Turtle graphics:

```
s = Record
[ "affine" :> M33 Double
, "pen"    :> Bool
]
```

- Affine matrix:
  - Position
  - Orientation
- Pen up or down

Bézier paths:

```
s = Record
[ "affine" :> M33 Double
]
```

- Only affine matrix

Vector graphics:

```
Record
[ "affine" :> M33 Double
, "pen"    :> Bool
, "fill"   :> Record
  [ "color"   :> Rgb
  ]
, "stroke" :> Record
  [ "color"  :> Rgb
  , "width"  :> Double
  ]
]
```

An example of state change: space rotation

For the sake of simplicity, we will allow all state functions.

```
newtype Endo s = Endo (s -> s)
```

Thanks to the field polymorphism, the `rotation` state change remains polymorphic.

```
rotation :: HasField "affine" s (M33 a) => a -> Endo s
rotation θ = Endo $ \s ->
  s {affine = s.affine <>
```

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}\}$$

Syntactic sugar: `left` and `right`

Using `change` and `rotation` to define `left` :

```haskell
left :: HasField "affine" s (M33 a) => a -> Media c s p w ()
left a = change (rotation (degToRad a))

right = left (-a)
```

Extensibility comes from field polymorphism

Since the state is polymorphic, we can use `left` and `right` with *any media* whose state contains an affine matrix !

Domain-specific instanciation

Turtle graphics:

```
type MediaTurtle =
 Media ChangeTurtle
       StateTurtle
       Segment
       Image
```

Evaluation:

```
execTurtle ::
  MediaTurtle a ->
  Image
```

Bézier paths:

```
type MediaPath =
  Media ChangePath
        StatePath
        PathPoint
        Path
```

Evaluation:

```
execPath ::
  MediaPath a ->
  Path
```

Vector graphics:

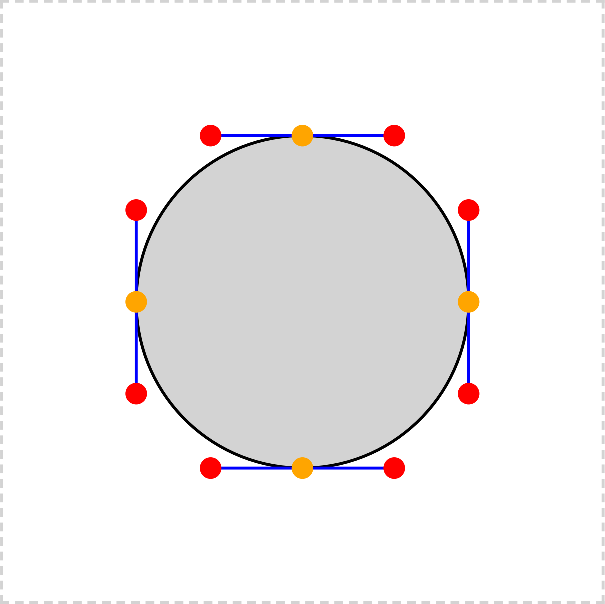
```
type MediaShapes =
 Media ChangeShapes
       StateShapes
       Path
       Image
```

Evaluation:

```
execShapes ::
  MediaShapes a ->
  Image
```

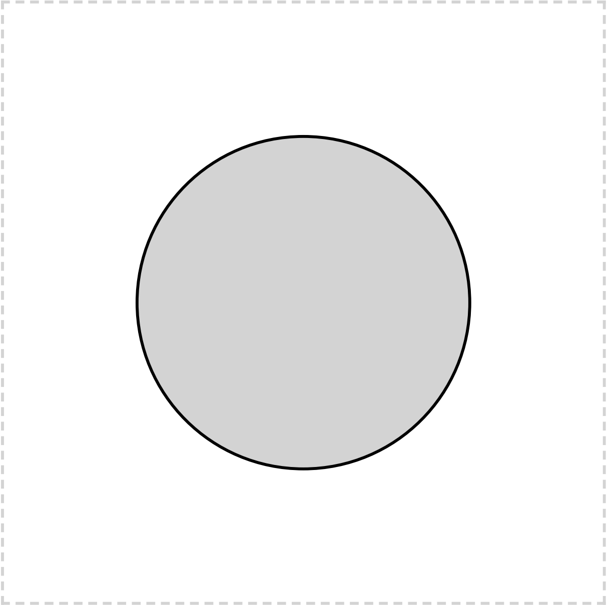
Approximation of a circle as a bezier path

```haskell
bezierCircle :: MediaPath ()
bezierCircle = repeat 4 $ do
    moveY 1
    moveX (-a)
    bezier

    moveX a
    anchor

    moveX a
    bezier

    moveX (-a)
    moveY (-1)
    left (360 / 8)
  where
    a = (4 / 3) * tan (pi / 8)
```



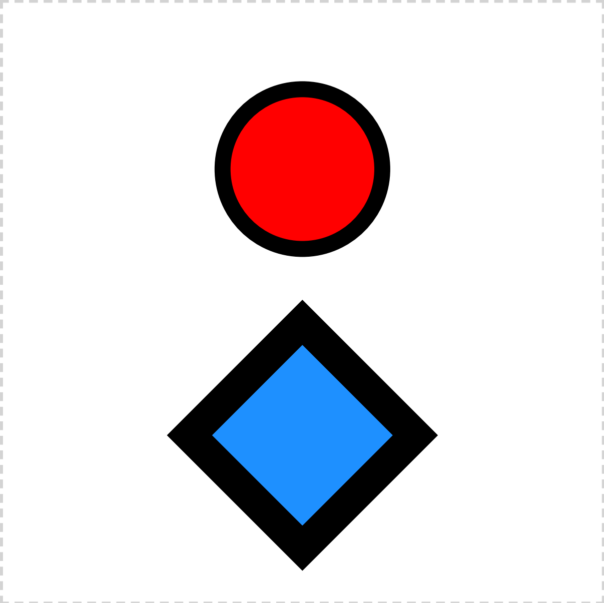
Using the bezier circle as a shape primitive

```
circle :: MediaShapes ()
circle =
  prim $ Closed $
    execPath bezierCircle
```



Fun with other parameters

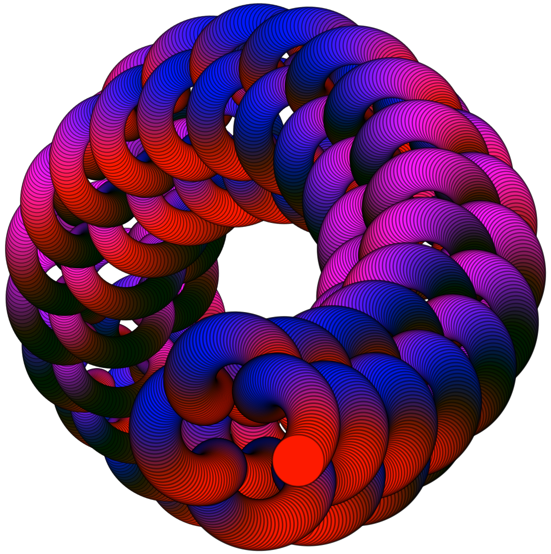
```
shapes :: MediaShapes ()
shapes = do
  fill red
  stroke black
  circle

  moveY (-2)
  fill blue
  stroke $ width 2
  left 45
  square
```



Key takeaways

To define a Domain-Specific Language with Pandia, one needs to specify:

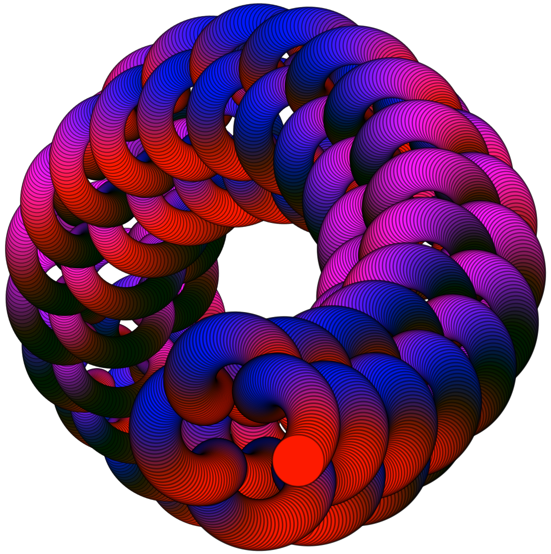
- A **rendering monoid** `w`
- A set of **states** `s`
- A set `p` of **primitives**
  `renderPrim :: s -> p -> w`
- A monoid of **state changes** acts on the states `s`
  `lact :: c -> s -> s`



Annex A. Spirograph

- the `wheel` transformation spins the space
- `redNBlue` alternates between red and blue depending on time

```
spirograph =
  transf (wheel 70 1)   $
  transf (wheel 30 20)  $
  transf (wheel 10 100) $
  transf redNBlue       $
    repeat 8180 $ do
      circle
      delay 0.001
```



Annex B. Extensible records

- We want to implement `left`, `right`, `penup`, `pendown` as **state changes** of the form `Endo s`.
- The turtles state must contain:
  - the turtle's **position**
  - the turtle's **orientation**
  - whether the **pen** is down or up

```
type TurtleState = Record
  [ "affine" :> M33 Double -- affine matrix
  , "pen"    :> Bool
  ]
```

Annex B. Extensible records

```haskell
type TurtleState = Record
  [ "affine" :> M33 Double
  , "pen"    :> Bool
  ]
```

Using `OverloadedRecordDot` and `OverloadedRecordUpdate` :

- **Read** the pen state:

  ```haskell
  s.pen -- read
  ```

- **Modify** the pen state:

  ```haskell
  s{pen = True} -- set
  ```

Annex C. Virtual record fields

```
instance HasField "transl" (M33 Double) (V2 Double)
  where
```

$$hasField \begin{pmatrix} M & & x \\ & & y \\ 0 & 0 & 1 \end{pmatrix} = (\lambda \begin{pmatrix} x' \\ y' \end{pmatrix} \to \begin{pmatrix} M & & x' \\ & & y' \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix})$$

- **Read** the turtle's position:

```
s.affine.transl
```

- **Modify** the turtle's position

```
s {affine.transl = s.affine.transl + V2 1 1}
```

Annex D. Implementing `forward`

1. Move the turtle *without* leaving a trace behind.

```
move v = change $ Endo $
  \s -> s {affine.transl = s.affine.transl + v}
```

Annex D. Implementing `forward`

2. Combine `move` and `segment` :

```
forward :: (HasField "affine" s (M33 a), HasField "pen" s Bool)
  => a -> Media (Endo s) s Segment w ()

forward y = do
  s <- get
  move (V2 0 y)
  if s.pen do
    s' <- get
    segment s.affine.transl s'.affine.transl
  else
    return ()
```

- The state `s` and the media monoid `w` are still polymorphic.

Annex E. `Media` is a monad

```
newtype Media c s p w a = Media (c -> s -> (a,w,s))
```

When the images `w` form a **monoid**, `Media` is a monad

```
return a = Media $ \ _ s -> (a, mempty, s)

Media m >>= Media f = Media $ \ c s ->
              let  (a, w, s') = m c s
                   (b, w', s'') = f a c s'
              in
                   (b, w <> w', s'')
```

Annex E. `Media` is a state monad

- Access the state:

```haskell
get :: Media s w s
get = Media $ \ _ s -> (s, mempty, s)
```

- Modify the state:

```haskell
modify :: (s -> s) -> Media s w ()
modify f = Media $ \ _ s -> ((), mempty, f s)
```

Annex E. `Media` is a writer monad

```haskell
tell :: w -> Media s w ()
tell w = Media $ \ _ s -> ((), w, s)
```