User-defined types

- Type synonyms (typedefs in C)

```
type Point = (Float, Float)
type Path  = [Point]
```

- Algebraic data types
  - Combination of structs and unions
  - together with pointers in C

- Data types can be like structs in C (we call those data types product types)

```
data Point2 = MkPoint2 Float Float
```

this is called a ***data constructor***

fields are not named, characterised by position in the definition

```
typedef struct {
  float x, y;
} Point2;
```

The corresponding definition in C

Data Constructors

- Data constructors are a (special kind of) functions:

```
data Point2 = MkPoint2 Float Float

MkPoint2 :: Float -> Float -> Point2
```

- Arguments to data constructors can always be recovered using pattern matching:

```
distFromZ:: Point2 -> Float
distFromZ (MkPoint2 x y)
  = sqrt (x*x + y*y)
```

Data Constructors

- We already know some other data constructors:

```
(,) :: a -> b -> (a,b)
fst (x, _) = x
```

```
[]:: [a]
(:) :: a -> [a] -> [a]

length []       = 0
length (_ : xs) = 1 + length xs
```

```haskell
data Point2 = MkPoint2 Float Float

point :: Point2
point = MkPoint2 1.3 2.45
```

```c
typedef struct {
  float x, y;
} Point2;


Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```

- Data types can be like structs in C (we call those data types product types)

```
data Point2 = MkPoint2
  { xPoint :: Float
  , yPoint :: Float
  }
```

fields are named,
characterised by position
in the definition and unique name

```
typedef struct {
          float x, y;
} Point2;
```

The corresponding definition in C

```
data Point2 = MkPoint2
  { xPoint :: Float
  , yPoint :: Float
  }

point :: Point2
point = MkPoint2 1.3 2.45
– or
point = MkPoint2 {yPoint = 2.45, xPoint = 1.3}
```

fields can also be named

```
typedef struct {
  unsigned int x, y;
} Point2;


Point2 point = {1.3, 2.45};
// or
Point2 point;
point.x = 1.3;
point.y = 2.45
```

```haskell
 data Point2 = MkPoint2
   { xPoint :: Float
   , yPoint :: Float
   }

-- the above definition brings three functions
-- into scope:
MkPoint2 :: Float -> Float -> Point2   -- constructor
xPoint :: Point2 -> Float  -- access function for x
yPoint :: Point2 -> Float  -- access function for y


-- using pattern matching to access components
distance :: Point2 -> Point2 -> Float
distance (MkPoint2 x1 y1) (MkPoint2 x2 y2) =
  sqrt ((x2 - x1)^2  + (y2 - y1)^2)

-- using access functions
distance p1 p2 =
  sqrt ((xPoint p2 - xPoint p1)^2  +
        (yPoint p2 - yPoint p1)^2)
```

- Problem: define a type to model shapes. A shape can be a rectangle (position, width, height) or a circle (position, radius)
- Data types can be like unions in C (we call those data types sum types)

```
data Shape = Rectangle Point Float Float
           | Circle    Point Float
```

```
enum tag {RECTANGLE_SHAPE, CIRCLE_SHAPE};
struct mkRectangle {
  enum tag       theTag;
  float height;
  float width;
}
struct mkCircle {
  enum tag theTag;
  point     pos;
  radius    float;
}
typedef union {
  struct mkCircle    aCircle;
  struct mkRectangle aRectangle;
} Shape;
```

The definition in C

UNSW THE UNIVERSITY OF NEW SOUTH WALES SYDNEY • AUSTRALIA

Product-Sum Types

- We call Haskell's data types also product-sum types
- They can be recursive as well
- In contrast to data types in C, but much like generics in Java and C#, Haskell data types can be parameterised

Type parameter

```
data Maybe a = Nothing | Just a
```

Identifiers in Haskell

- Alphanumeric with underscores (_) and prime symbols (')
- Case matters

Functions & variables	lower case	`map, pi, (+), (++)`
Data constructors	Upper case	`True, Nothing, (:)`
Type variables	lower case	`a, b, c, eltType`
Type constructors	`Upper case`	`Int, Bool, IO`

Next Thursday: guest lecture

- Patrick Flanagan (Jane St, Hongkong)

- Thu, 15 March