

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

# Composite Data Types

Most of the types we have seen so far are **basic** types, in the sense that they represent built-in machine data representations. Real programming languages feature ways to **compose** types together to produce new types, such as:

Classes

Tuples

Structs

Unions

Records

# Combining values conjunctively

We want to store two things in one value.

(might want to use non-compact slides for this one)

C Structs

Java  
"Better" Java

Has

type Point

midpoint (

= ((x1+x2

type

float

float

} poi

point }

poi Pe

mid

mid

ret

}

cl

```
class Point {
```

```
  private float x;
```

```
  private float y;
```

```
  public Point (float x, float y) {
```

```
    this.x = x; this.y = y;
```

```
  }
```

```
  public float getX() {return this.x;}
```

```
  public float getY() {return this.y;}
```

```
  public float setX(float x) {this.x=x;}
```

```
  public float setY(float y) {this.y=y;}
```

```
Point midpoint (Point p1, Point p2) {
```

```
  return new Point((p1.getX() + p2.getX()) / 2.0,
```

```
                  (p2.getY() + p2.getY()) / 2.0);
```

```
}
```

y2)

## Product Types

In MinHS, we will have a very minimal way to accomplish this, called a *product type*:

$$\tau_1 \times \tau_2$$

We won't have type declarations, named fields or anything like that. More than two values can be combined by nesting products, for example a three dimensional vector:

$$\text{Int} \times (\text{Int} \times \text{Int})$$

## Constructors and Eliminators

We can **construct** a product type similar to Haskell tuples:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

The only way to extract each component of the product is to use the `fst` and `snd` eliminators:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2}$$

## Examples

### Example (Midpoint)

```
recfun midpoint :: ((Int × Int) → (Int × Int) → Int) p1 =  
  recfun midpoint' :: ((Int × Int) → Int) p2 =  
    ((fst p1 + fst p2) ÷ 2, (snd p1 + snd p2) ÷ 2)
```

### Example (Uncurried Division)

```
recfun div :: ((Int × Int) → Int) args =  
  if (fst args < snd args)  
    then 0  
    else div (fst args - snd args, snd args)
```

# Dynamic Semantics

$$\frac{e_1 \mapsto_M e'_1}{(e_1, e_2) \mapsto_M (e'_1, e_2)}$$

$$\frac{e_2 \mapsto_M e'_2}{(v_1, e_2) \mapsto_M (v_1, e'_2)}$$

$$\frac{e \mapsto e'}{\text{fst } e \mapsto_M \text{fst } e'}$$

$$\frac{e \mapsto e'}{\text{snd } e \mapsto_M \text{snd } e'}$$

$$\frac{}{\text{fst } (v_1, v_2) \mapsto_M v_1}$$

$$\frac{}{\text{snd } (v_1, v_2) \mapsto_M v_2}$$

# Unit Types

Currently, we have no way to express a type with just **one** value. This may seem useless at first, but it becomes useful in combination with other types.

We'll introduce a type, **1**, pronounced *unit*, that has exactly one inhabitant, written `()`:

$$\frac{}{\Gamma \vdash () : \mathbf{1}}$$



## Disjunctive Composition

We can't, with the types we have, express a type with exactly **three** values.

### Example (Trivalued type)

```
data TrafficLight = Red | Amber | Green
```

In general we want to express data that can be **one** of multiple **alternatives**, that contain different bits of data.

### Example (More elaborate alternatives)

```
type Length = Int
type Angle = Int
data Shape = Rect Length Length
           | Circle Length | Point
           | Triangle Angle Length Length
```

This is awkward in many languages. In Java we'd have to use inheritance. In C we'd have to use unions.

## Sum Types

We will use *sum types* to express the possibility that data may be one of two forms.

$$\tau_1 + \tau_2$$

This is similar to the Haskell `Either` type.

Our `TrafficLight` type can be expressed (grotesquely) as a sum of units:

$$\text{TrafficLight} \simeq \mathbf{1} + (\mathbf{1} + \mathbf{1})$$

## Constructors and Eliminators for Sums

To make a value of type  $\tau_1 + \tau_2$ , we invoke one of two **constructors**:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{InL } e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{InR } e : \tau_1 + \tau_2}$$

We can branch based on which alternative is used using **pattern matching**:

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad x : \tau_1, \Gamma \vdash e_1 : \tau \quad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\text{case } e \text{ of InL } x \rightarrow e_1; \text{InR } y \rightarrow e_2) : \tau}$$

(Using concrete syntax here, for readability.)

(Feel free to replace it with abstract syntax of your choosing.)

# Examples

## Example (Traffic Lights)

Our traffic light type has three values as required:

$$\text{TrafficLight} \simeq \mathbf{1} + (\mathbf{1} + \mathbf{1})$$

$$\text{Red} \simeq \text{InL } ()$$

$$\text{Amber} \simeq \text{InR } (\text{InL } ())$$

$$\text{Green} \simeq \text{InR } (\text{InR } (\text{InL } ()))$$

## Examples

We can convert most (non-recursive) Haskell types to equivalent MinHs types now.

- ① Replace all constructors with **1**
- ② Add a  $\times$  between all constructor arguments.
- ③ Change the `|` character that separates constructors to a `+`.

### Example

```
data Shape = Rect Length Length
           | Circle Length | Point
           | Triangle Angle Length Length
```

$\simeq$

```
1 × (Int × Int)
+ 1 × Int + 1
+ 1 × (Int × (Int × Int))
```

# Dynamic Semantics

$$\frac{e \mapsto_M e'}{\text{InL } e \mapsto_M \text{InL } e'} \quad \frac{e \mapsto_M e'}{\text{InR } e \mapsto_M \text{InR } e'}$$

$$\frac{e \mapsto_M e'}{(\text{case } e \text{ of InL } x. e_1; \text{InR } y. e_2) \mapsto_M (\text{case } e' \text{ of InL } x. e_1; \text{InR } y. e_2)}$$

$$\frac{}{(\text{case } (\text{InL } v) \text{ of InL } x. e_1; \text{InR } y. e_2) \mapsto_M e_1[x := v]}$$

$$\frac{}{(\text{case } (\text{InR } v) \text{ of InL } x. e_1; \text{InR } y. e_2) \mapsto_M e_2[y := v]}$$

## The Empty Type

We add another type, called **0**, that has **no** inhabitants. Because it is empty, there is no way to construct it.

We do have a way to eliminate it, however:

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \text{absurd } e : \tau}$$

If I have a variable of the **empty** type in scope, we must be looking at an expression that will **never** be evaluated. Therefore, we can assign any type we like to this expression, because it will never be executed.

## Semiring Structure

These types we have defined form an algebraic structure called a *commutative semiring*.

Laws for  $(\tau, +, \mathbf{0})$ :

- Associativity:  $(\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)$
- Identity:  $\mathbf{0} + \tau \simeq \tau$
- Commutativity:  $\tau_1 + \tau_2 \simeq \tau_2 + \tau_1$

Laws for  $(\tau, \times, \mathbf{1})$

- Associativity:  $(\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)$
- Identity:  $\mathbf{1} \times \tau \simeq \tau$
- Commutativity:  $\tau_1 \times \tau_2 \simeq \tau_2 \times \tau_1$

Combining  $\times$  and  $+$ :

- Distributivity:  $\tau_1 \times (\tau_2 + \tau_3) \simeq (\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)$
- Absorption:  $\mathbf{0} \times \tau \simeq \mathbf{0}$

What does  $\simeq$  mean here?



# Isomorphism

Two types  $\tau_1$  and  $\tau_2$  are *isomorphic*, written  $\tau_1 \simeq \tau_2$ , if there exists a *bijection* between them. This means that for each value in  $\tau_1$  we can find a unique value in  $\tau_2$  and vice versa.

We can use isomorphisms to simplify our Shape type:

$$\begin{aligned}
 & \mathbf{1} \times (\text{Int} \times \text{Int}) \\
 + & \mathbf{1} \times \text{Int} + \mathbf{1} \\
 + & \mathbf{1} \times (\text{Int} \times (\text{Int} \times \text{Int}))
 \end{aligned}$$

$$\simeq$$

$$\begin{aligned}
 & \text{Int} \times \text{Int} \\
 + & \text{Int} + \mathbf{1} \\
 + & \text{Int} \times (\text{Int} \times \text{Int})
 \end{aligned}$$

## Examining our Types

Lets look at the rules for typed lambda calculus extended with sums and products:

$$\begin{array}{c}
 \frac{}{\Gamma \vdash e : \mathbf{0}} \\
 \frac{}{\Gamma \vdash \text{absurd } e : \tau} \quad \frac{}{\Gamma \vdash () : \mathbf{1}} \\
 \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{InL } e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{InR } e : \tau_1 + \tau_2} \\
 \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad x : \tau_1, \Gamma \vdash e_1 : \tau \quad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\text{case } e \text{ of InL } x \rightarrow e_1; \text{InR } y \rightarrow e_2) : \tau} \\
 \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{fst } e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{snd } e : \tau_2} \\
 \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \quad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}
 \end{array}$$

## Squinting a Little

Lets remove all the **terms**, leaving just the types and the contexts:

$$\begin{array}{c}
 \frac{\Gamma \vdash \mathbf{0}}{\Gamma \vdash \tau} \quad \frac{}{\Gamma \vdash \mathbf{1}} \\
 \\
 \frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 + \tau_2} \quad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 + \tau_2} \\
 \\
 \frac{\Gamma \vdash \tau_1 + \tau_2 \quad \tau_1, \Gamma \vdash \tau \quad \tau_2, \Gamma \vdash \tau}{\Gamma \vdash \tau} \\
 \\
 \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_1} \quad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_2} \\
 \\
 \frac{\Gamma \vdash \tau_1 \rightarrow \tau_2}{\Gamma \vdash \tau_2} \quad \frac{\Gamma \vdash \tau_1}{\tau_1, \Gamma \vdash \tau_2} \quad \frac{\tau_1, \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \rightarrow \tau_2}
 \end{array}$$

Does this resemble anything you've seen before?

## A surprising coincidence!

Types are exactly the same structure as *constructive logic*:

$$\begin{array}{c}
 \frac{\Gamma \vdash \perp}{\Gamma \vdash P} \quad \frac{}{\Gamma \vdash \top} \\
 \\
 \frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2} \quad \frac{\Gamma \vdash P_2}{\Gamma \vdash P_1 \vee P_2} \\
 \\
 \frac{\Gamma \vdash P_1 \vee P_2 \quad P_1, \Gamma \vdash P \quad P_2, \Gamma \vdash P}{\Gamma \vdash P} \\
 \\
 \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \quad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \quad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2} \\
 \\
 \frac{\Gamma \vdash P_1 \rightarrow P_2 \quad \Gamma \vdash P_1}{\Gamma \vdash P_2} \quad \frac{P_1, \Gamma \vdash P_2}{\Gamma \vdash P_1 \rightarrow P_2}
 \end{array}$$

This means, if we can construct a **program** of a certain **type**, we have also created a constructive **proof** of a certain **proposition**.

## The Curry-Howard Isomorphism

This correspondence goes by many names, but is usually attributed to **Haskell Curry** and **William Howard**.

It is a *very deep* result:

Programming	Logic
Types	Propositions
Programs	Proofs
Evaluation	Proof Simplification

It turns out, no matter what logic you want to define, there is always a corresponding  $\lambda$ -calculus, and vice versa.

Constructive Logic	Typed $\lambda$ -Calculus
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

## Examples

### Example (Commutativity of Conjunction)

$andComm :: A \times B \rightarrow B \times A$   
 $andComm\ p = (snd\ p, fst\ p)$

This proves  $A \wedge B \rightarrow B \wedge A$ .

### Example (Transitivity of Implication)

$transitive :: (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C)$   
 $transitive\ f\ g\ x = g\ (f\ x)$

Transitivity of implication is just **function composition**.

## Caveats

All functions we define have to be **total and terminating**.  
Otherwise we get an **inconsistent** logic that lets us prove false things:

$$\begin{aligned} \text{proof}_1 &:: P = NP \\ \text{proof}_1 &= \text{proof}_1 \end{aligned}$$

$$\begin{aligned} \text{proof}_2 &:: P \neq NP \\ \text{proof}_2 &= \text{proof}_2 \end{aligned}$$

Most common calculi correspond to **constructive** logic, not **classical** ones, so principles like the **law of excluded middle** or **double negation elimination** do **not** hold:

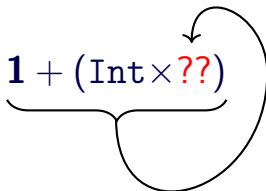
$$\neg\neg P \rightarrow P$$

# Inductive Structures

What about types like lists?

```
data IntList = Nil | Cons Int IntList
```

We **can't** express these in MinHS yet:



We need a way to do recursion!



# Recursive Types

We introduce a new form of type, written **rec**  $t. \tau$ , that allows us to refer to the entire type:

$$\begin{aligned} \text{IntList} &\simeq \mathbf{rec} \ t. \ \mathbf{1} + (\text{Int} \times t) \\ &\simeq \mathbf{1} + (\text{Int} \times (\mathbf{rec} \ t. \ \mathbf{1} + (\text{Int} \times t))) \\ &\simeq \mathbf{1} + (\text{Int} \times (\mathbf{1} + (\text{Int} \times (\mathbf{1} + (\text{Int} \times (\mathbf{1} + (\text{Int} \times t)))))) \\ &\simeq \dots \end{aligned}$$

# Typing Rules

We construct a recursive type with `roll`, and unpack the recursion one level with `unroll`:

$$\frac{\Gamma \vdash e : \tau[t := \mathbf{rec} \ t. \ \tau]}{\Gamma \vdash \mathbf{roll} \ e : \mathbf{rec} \ t. \ \tau}$$

$$\frac{\Gamma \vdash e : \mathbf{rec} \ t. \ \tau}{\Gamma \vdash \mathbf{unroll} \ e : \tau[t := \mathbf{rec} \ t. \ \tau]}$$

# Example

## Example

Take our IntList example:

$$\text{rec } t. \mathbf{1} + (\text{Int} \times t)$$

$$[] = \text{roll } (\text{InL } ( ))$$

$$[1] = \text{roll } (\text{InR } (1, \text{roll } (\text{InL } ( ))))$$

$$[1, 2] = \text{roll } (\text{InR } (1, \text{roll } (\text{InR } (2, \text{roll } (\text{InL } ( ))))))$$

# Dynamic Semantics

Nothing interesting here:

$$\frac{e \mapsto_M e'}{\text{roll } e \mapsto_M \text{roll } e'} \quad \frac{e \mapsto_M e'}{\text{unroll } e \mapsto_M \text{unroll } e'}$$

$$\frac{}{\text{unroll } (\text{roll } e) \mapsto_M e}$$