COMP3161/9164 Concepts of Programming Languages

Algebraic Data Types; Recursive Types

Rob Sison
UNSW

Term 3 2024

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 have ways to *compose* types to produce new types:

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 have ways to *compose* types to produce new types:

Tuples

Structs

Records

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 have ways to *compose* types to produce new types:

Classes

Tuples

Structs

Records

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 have ways to *compose* types to produce new types:

Classes

Tuples

Structs

Unions

Records

Combining values conjunctively

We want to store two things in one value.

C Structs

```c
typedef struct point {
  float x;
  float y;
} point;
point midPoint (point p1, point p2) {
  point mid;
  mid.x = (p1.x + p2.x) / 2.0;
  mid.y = (p2.y + p2.y) / 2.0;
  return mid;
}
```

Combining values conjunctively

We want to store two things in one value.

C Structs

```
typec
  flo
  flo
} poi
poin
  poi
  mid
  mid
  ret
}
```

Java

```
class Point {
  public float x;
  public float y;
}
Point midPoint (Point p1, Point p2) {
  Point mid = new Point();
  mid.x = (p1.x + p2.x) / 2.0;
  mid.y = (p2.y + p2.y) / 2.0;
  return mid;
}
```

Combining values conjunctively

We want to store two things in one value.

C Structs

```
typed
  flo
  flo
} poi
poin
  poi
  mid
  mid
  ret
}
```

Java

```
cl



}
P


}
```

“Better” Java

```
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);
}
```

Combining values conjunctively

We want to store two things in one value.

C Structs

```
typed
  fl
mid
ret
}
```

Java

```
}
```

"Better" Java

```
            {
          oat x;
          oat y;
          nt (float x, float y) {
          = x; this.y = y;

          at getX() {return this.x;}
          at 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);
}
```

Haskell Tuples

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

midpoint (x1,y1) (x2,y2)
  = ((x1+x2)/2, (y1+y2)/2)
```

Combining values conjunctively

We want to store two things in one value.

Haskell Tuples

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

midpoint (x1,y1) (x2,y2)
  = ((x1+x2)/2, (y1+y2)/2)
```

Haskell Datatypes

```haskell
data Point =
  Pnt { x :: Float
      , y :: Float
      }

midpoint (Pnt x1 y1) (Pnt x2 y2)
  = Pnt ((x1+x2)/2) ((y1+y2)/2)

midpoint' p1 p2 =
  = Pnt ((x p1 + x p2) / 2)
        ((y p1 + y p2) / 2)
```

C St

```
typed
  mid
  ret
}
}
```

```
  public float
  public float
}
Point midPoin
  return new P
             (p2.getY() + p2.getY()) / 2.0);
}
```

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:

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

Constructors and Eliminators

We can construct a product type similar to Haskell tuples:

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

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

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e : \tau_2}$$

Examples

Example (Midpoint)

$$\begin{array}{l}\textbf{recfun}\ \mathit{midpoint}\ :: \\ \quad ((\mathtt{Int} \times \mathtt{Int}) \to (\mathtt{Int} \times \mathtt{Int}) \to (\mathtt{Int} \times \mathtt{Int}))\ p_1 = \\ \quad \textbf{recfun}\ \mathit{midpoint}'\ :: \\ \quad\quad ((\mathtt{Int} \times \mathtt{Int}) \to (\mathtt{Int} \times \mathtt{Int}))\ p_2 = \\ \quad\quad ((\mathsf{fst}\ p_1 + \mathsf{fst}\ p_2) \div 2, (\mathsf{snd}\ p_1 + \mathsf{snd}\ p_2) \div 2)\end{array}$$

Example (Uncurried Division)

$$\begin{array}{l}\textbf{recfun}\ \mathit{div}\ :: ((\mathtt{Int} \times \mathtt{Int}) \to \mathtt{Int})\ \mathit{args} = \\ \quad \textbf{if}\ (\mathsf{fst}\ \mathit{args} < \mathsf{snd}\ \mathit{args}) \\ \quad\quad \textbf{then}\ 0 \\ \quad\quad \textbf{else}\ 1 + \mathit{div}\ (\mathsf{fst}\ \mathit{args} - \mathsf{snd}\ \mathit{args}, \mathsf{snd}\ \mathit{args})\end{array}$$

Dynamic Semantics

$$\frac{e_1 \mapsto_M e_1'}{(e_1, e_2) \mapsto_M (e_1', e_2)} \qquad \frac{e_2 \mapsto_M e_2'}{(v_1, e_2) \mapsto_M (v_1, e_2')}$$

$$\frac{e \mapsto e'}{\mathsf{fst}\ e \mapsto_M \mathsf{fst}\ e'} \qquad \frac{e \mapsto e'}{\mathsf{snd}\ e \mapsto_M \mathsf{snd}\ e'}$$

$$\frac{}{\mathsf{fst}\ (v_1, v_2) \mapsto_M v_1} \qquad \frac{}{\mathsf{snd}\ (v_1, v_2) \mapsto_M v_2}$$

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, $\mathbf{1}$, 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
```

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:

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

Constructors and Eliminators for Sums

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

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{InL}\ x \to e_1; \mathsf{InR}\ y \to e_2) : \tau}$$

Examples

Example (Traffic Lights)

Our traffic light type has three values as required:

$$\begin{array}{lcl}
\texttt{TrafficLight} & \simeq & \mathbf{1} + (\mathbf{1} + \mathbf{1}) \\
\\
\texttt{Red} & \simeq & \mathsf{InL}\ () \\
\texttt{Amber} & \simeq & \mathsf{InR}\ (\mathsf{InL}\ ()) \\
\texttt{Green} & \simeq & \mathsf{InR}\ (\mathsf{InR}\ ())
\end{array}$$

Examples

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

1. Replace all constructors with $\mathbf{1}$
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
```

$$\simeq$$

$$\begin{array}{ll} & \mathbf{1} \times (\texttt{Int} \times \texttt{Int}) \\ + & \mathbf{1} \times \texttt{Int} + \mathbf{1} \\ + & \mathbf{1} \times (\texttt{Int} \times (\texttt{Int} \times \texttt{Int})) \end{array}$$

Dynamic Semantics

$$\frac{e \mapsto_M e'}{\mathsf{InL}\ e \mapsto_M \mathsf{InL}\ e'} \qquad \frac{e \mapsto_M e'}{\mathsf{InR}\ e \mapsto_M \mathsf{InR}\ e'}$$

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

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

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

The Empty Type

We add another type, called $\mathbf{0}$, that has no inhabitants. Because it is empty, there is no way to construct it.
We do have a way to eliminate it, however:

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

The Empty Type

We add another type, called $\mathbf{0}$, that has no inhabitants. Because it is empty, there is no way to construct it.
We do have a way to eliminate it, however:

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

If a variable of the empty type is 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

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

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

- Associativity: $(\tau_1 + \tau_2) + \tau_3 \;\simeq\; \tau_1 + (\tau_2 + \tau_3)$

Semiring Structure

The 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: $\mathbf{0} + \tau \;\simeq\; \tau$

Semiring Structure

The 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: $\tau_1 + \tau_2 \;\simeq\; \tau_2 + \tau_1$

Semiring Structure

The 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: $(\tau_1 \times \tau_2) \times \tau_3 \;\simeq\; \tau_1 \times (\tau_2 \times \tau_3)$

Semiring Structure

The 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: $\mathbf{1} \times \tau \;\simeq\; \tau$

Semiring Structure

The 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: $\tau_1 \times \tau_2 \;\simeq\; \tau_2 \times \tau_1$

Semiring Structure

The 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: $\tau_1 \times (\tau_2 + \tau_3) \ \simeq \ (\tau_1 \times \tau_2) + (\tau_1 \times \tau_3)$

Semiring Structure

The 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: $\mathbf{0} \times \tau \;\simeq\; \mathbf{0}$

Semiring Structure

The 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 τ_1 and τ_2 are *isomorphic*, written $\tau_1 \simeq \tau_2$, if there exists a *bijection* between them. This means that for each value in τ_1 we can find a unique value in τ_2 and vice versa.

We can use isomorphisms to simplify our `Shape` type:

$$\begin{array}{ll} & \mathbf{1} \times (\texttt{Int} \times \texttt{Int}) \\ + & \mathbf{1} \times \texttt{Int} + \mathbf{1} \\ + & \mathbf{1} \times (\texttt{Int} \times (\texttt{Int} \times \texttt{Int})) \end{array}$$

$$\simeq$$

$$\begin{array}{ll} & \texttt{Int} \times \texttt{Int} \\ + & \texttt{Int} + \mathbf{1} \\ + & \texttt{Int} \times (\texttt{Int} \times \texttt{Int}) \end{array}$$

Examining our Types

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

$$\frac{\Gamma \vdash e : \mathbf{0}}{\Gamma \vdash \mathsf{absurd}\ e : \tau} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{InL}\ x \to e_1; \mathsf{InR}\ y \to e_2) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{snd}\ e : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{x : \tau_1, \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

Squinting a Little

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

$$\frac{\Gamma \vdash \mathbf{0}}{\Gamma \vdash \tau} \qquad \frac{}{\Gamma \vdash \mathbf{1}}$$

$$\frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 + \tau_2 \qquad \tau_1, \Gamma \vdash \tau \qquad \tau_2, \Gamma \vdash \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_1} \qquad \frac{\Gamma \vdash \tau_1 \times \tau_2}{\Gamma \vdash \tau_2}$$

$$\frac{\Gamma \vdash \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1}{\Gamma \vdash \tau_2} \qquad \frac{\tau_1, \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \to \tau_2}$$

Does this resemble anything you've seen before?

A surprising coincidence!

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

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{\Gamma \vdash P_1}{\Gamma \vdash P_1 \vee P_2} \qquad \frac{\Gamma \vdash P_2}{\Gamma \vdash P_1 \vee P_2}$$

$$\frac{\Gamma \vdash P_1 \vee P_2 \qquad P_1, \Gamma \vdash P \qquad P_2, \Gamma \vdash P}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \wedge P_2} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_1} \qquad \frac{\Gamma \vdash P_1 \wedge P_2}{\Gamma \vdash P_2}$$

$$\frac{\Gamma \vdash P_1 \to P_2 \qquad \Gamma \vdash P_1}{\Gamma \vdash P_2} \qquad \frac{P_1, \Gamma \vdash P_2}{\Gamma \vdash P_1 \to P_2}$$

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

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 λ-calculus, and vice versa.

Constructive Logic	Typed λ-Calculus
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

Examples

Example (Commutativity of Conjunction)

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

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

Examples

Example (Commutativity of Conjunction)

$$\begin{aligned} \mathit{andComm} &:: A \times B \to B \times A \\ \mathit{andComm}\ p &= (\mathrm{snd}\ p, \mathrm{fst}\ p) \end{aligned}$$

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

Example (Transitivity of Implication)

$$\begin{aligned} \mathit{transitive} &:: (A \to B) \to (B \to C) \to (A \to C) \\ \mathit{transitive}\ f\ g\ x &= g\ (f\ x) \end{aligned}$$

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:

$$\mathit{proof}_1 :: \mathsf{P} = \mathsf{NP}$$
$$\mathit{proof}_1 = \mathit{proof}_1$$

$$\mathit{proof}_2 :: \mathsf{P} \neq \mathsf{NP}$$
$$\mathit{proof}_2 = \mathit{proof}_2$$

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:

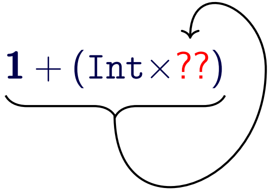
$$\mathbf{1} + (\texttt{Int} \times ??)$$

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 $\mathbf{rec}\ t.\ \tau$, that allows us to refer to the entire type:

$$\texttt{IntList} \quad \simeq \quad \mathbf{rec}\ t.\ \mathbf{1} + (\texttt{Int} \times t)$$

Recursive Types

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

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

Recursive Types

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

$$
\begin{array}{lcl}
\texttt{IntList} & \simeq & \textbf{rec}\ t.\ \mathbf{1} + (\texttt{Int} \times t) \\
 & \simeq & \mathbf{1} + (\texttt{Int} \times (\textbf{rec}\ t.\ \mathbf{1} + (\texttt{Int} \times t))) \\
 & \simeq & \mathbf{1} + (\texttt{Int} \times (\mathbf{1} + (\texttt{Int} \times (\textbf{rec}\ t.\ \mathbf{1} + (\texttt{Int} \times t))))) \\
 & \simeq & \cdots
\end{array}
$$

Typing Rules

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

$$\frac{??}{\Gamma \vdash \mathsf{roll}\ e : \mathbf{rec}\ t.\ \tau}$$

Typing Rules

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

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

Typing Rules

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

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

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

Example

Example

Take our `IntList` example:

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

$$[] \quad = $$

Example

Example

Take our `IntList` example:

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

$$\begin{array}{lcl} [] & = & \text{roll (InL ())} \\ [1] & = & \end{array}$$

Example

Example

Take our `IntList` example:

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

$$\begin{aligned} [] &= \mathsf{roll}\ (\mathsf{InL}\ ()) \\ [1] &= \mathsf{roll}\ (\mathsf{InR}\ (1, \mathsf{roll}\ (\mathsf{InL}\ ()))) \\ [1,2] &= \end{aligned}$$

Example

Example

Take our `IntList` example:

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

$$\begin{array}{lcl} [] & = & \mathsf{roll}\ (\mathsf{InL}\ ()) \\ [1] & = & \mathsf{roll}\ (\mathsf{InR}\ (1, \mathsf{roll}\ (\mathsf{InL}\ ()))) \\ [1,2] & = & \mathsf{roll}\ (\mathsf{InR}\ (1, \mathsf{roll}\ (\mathsf{InR}\ (2, \mathsf{roll}\ (\mathsf{InL}\ ()))))) \end{array}$$

Dynamic Semantics

Nothing interesting here:

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

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