Algebraic Data Types

Johannes Åman Pohjola
UNSW
Term 3 2022
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.

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

**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.

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

Haskell Datatypes

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

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

midpoint' p1 p2 =
  = ((x p1 + x p2) / 2,
     (y p1 + y p2) / 2)
Combining values conjunctively

We want to store two things in one value.

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

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

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.

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

**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' p1 p2 = ((x p1 + x p2) / 2, (y p1 + y p2) / 2)
```

**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;
}
```

**Java**

```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.

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

Haskell Tuples

type Point = (Float, Float)

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

Haskell Datatypes

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

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

C Structs

typedef struct point {
    float x;
    float y;
} point;

point mid_point (point p1, point p2) {
    point mid;
    mid.x = (p1.x + p2.x) / 2.0;
    mid.y = (p2.y + p2.y) / 2.0;
    return mid;
}

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;
}

“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);
}
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:

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \]

The only way to extract each component of the product is to use the \texttt{fst} and \texttt{snd} eliminators:

\[ \Gamma \vdash e : \tau_1 \times \tau_2 \]
\[ \Gamma \vdash \texttt{fst} e : \tau_1 \]
\[ \Gamma \vdash \texttt{snd} e : \tau_2 \]
Examples

Example (Midpoint)

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

Example (Uncurried Division)

```haskell
recfun div :: ((Int × Int) → Int)
args =
  if (fst args < snd args)
  then 0
  else 1 + div (fst args - snd args, snd args)
```
Dynamic Semantics

\[
\begin{align*}
\frac{e_1 \mapsto_M e'_1}{(e_1, e_2) \mapsto_M (e'_1, e_2)} \\
\frac{e_2 \mapsto_M e'_2}{(v_1, e_2) \mapsto_M (v_1, e'_2)} \\
\frac{e \mapsto e'}{\text{fst } e \mapsto_M \text{fst } e'} \\
\frac{\text{snd } e \mapsto_M \text{snd } e'}{\text{fst } (v_1, v_2) \mapsto_M v_1} \\
\frac{\text{snd } (v_1, v_2) \mapsto_M v_2}{}
\end{align*}
\]
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 ():

Γ ⊢ () : 1
Disjunctive Composition

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

Example (Trivalued type)

```haskell
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)

```haskell
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)

```haskell
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:

\[ \text{TrafficLight} \simeq 1 + (1 + 1) \]
Constructors and Eliminators for Sums

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

$$
\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2
$$

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

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

$$
\Gamma \vdash e : \tau_1 + \tau_2 \quad x : \tau_1, \Gamma \vdash e_1 : \tau \quad y : \tau_2, \Gamma \vdash e_2 : \tau
$$

$$
\Gamma \vdash (\text{case} \ e \ \text{of} \ \text{InL} \ x \rightarrow e_1; \ \text{InR} \ y \rightarrow e_2) : \tau
$$

(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:

\[ \text{TrafficLight} \cong 1 + (1 + 1) \]

\[ \text{Red} \cong \text{InL} () \]
\[ \text{Amber} \cong \text{InR} (\text{InL} ()) \]
\[ \text{Green} \cong \text{InR} (\text{InR} ()) \]
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 $\mathsf{+}$.

Example

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

$\cong$

$1 \times (\text{Int} \times \text{Int})$
$+ 1 \times \text{Int} + 1$
$+ 1 \times (\text{Int} \times (\text{Int} \times \text{Int}))$
Dynamic Semantics

\[
\begin{align*}
  e & \mapsto_M e' \\
  \text{InL } e & \mapsto_M \text{InL } e' \\
  \text{InR } e & \mapsto_M \text{InR } e'
\end{align*}
\]

\[
\begin{align*}
  e & \mapsto_M e' \\
  (\text{case } e \text{ of InL } x. \ e_1; \text{InR } y. \ e_2) & \mapsto_M (\text{case } e' \text{ of InL } x. \ e_1; \text{InR } y. \ e_2)
\end{align*}
\]

\[
\begin{align*}
  (\text{case } (\text{InL } v) \text{ of InL } x. \ e_1; \text{InR } y. \ e_2) & \mapsto_M e_1[x := v] \\
  (\text{case } (\text{InR } v) \text{ of InL } x. \ e_1; \text{InR } y. \ e_2) & \mapsto_M e_2[y := v]
\end{align*}
\]
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:

\[
\Gamma \vdash e : 0 \\
\Gamma \vdash \text{absurd } e : ?
\]
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:

\[ \Gamma \vdash e : 0 \]
\[ \Gamma \vdash \text{absurd } e : \tau \]

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 \((\tau, +, 0)\):

- **Associativity**: \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)

What does \(\simeq\) mean here?
Semiring Structure

The types we have defined form an algebraic structure called a \textit{commutative semiring}.

Laws for \((\tau, +, 0)\):

- **Associativity:** \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity:** \(0 + \tau \simeq \tau\)

What does \(\simeq\) mean here?
Semiring Structure

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

Laws for \((\tau, +, 0)\):

- **Associativity:** \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity:** \(0 + \tau \simeq \tau\)
- **Commutativity:** \(\tau_1 + \tau_2 \simeq \tau_2 + \tau_1\)

What does \(\simeq\) mean here?
Semiring Structure

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

Laws for $(\tau, +, 0)$:

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

Laws for $(\tau, \times, 1)$

- **Associativity**: $(\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)$

What does $\simeq$ mean here?
Semiring Structure

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

Laws for \((\tau, +, 0)\):

- **Associativity**: \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity**: \(0 + \tau \simeq \tau\)
- **Commutativity**: \(\tau_1 + \tau_2 \simeq \tau_2 + \tau_1\)

Laws for \((\tau, \times, 1)\):

- **Associativity**: \((\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)\)
- **Identity**: \(1 \times \tau \simeq \tau\)
Semiring Structure

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

Laws for \((\tau, +, 0)\):

- **Associativity**: \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity**: \(0 + \tau \simeq \tau\)
- **Commutativity**: \(\tau_1 + \tau_2 \simeq \tau_2 + \tau_1\)

Laws for \((\tau, \times, 1)\):

- **Associativity**: \((\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)\)
- **Identity**: \(1 \times \tau \simeq \tau\)
- **Commutativity**: \(\tau_1 \times \tau_2 \simeq \tau_2 \times \tau_1\)

What does \(\simeq\) mean here?
Semiring Structure

The types we have defined form an algebraic structure called a \textit{commutative semiring}.

Laws for \((\tau, +, 0)\):

- **Associativity**: \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity**: \(0 + \tau \simeq \tau\)
- **Commutativity**: \(\tau_1 + \tau_2 \simeq \tau_2 + \tau_1\)

Laws for \((\tau, \times, 1)\):

- **Associativity**: \((\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)\)
- **Identity**: \(1 \times \tau \simeq \tau\)
- **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 \((τ, +, 0)\):
- Associativity: \((τ_1 + τ_2) + τ_3 \simeq τ_1 + (τ_2 + τ_3)\)
- Identity: \(0 + τ \simeq τ\)
- Commutativity: \(τ_1 + τ_2 \simeq τ_2 + τ_1\)

Laws for \((τ, \times, 1)\):
- Associativity: \((τ_1 \times τ_2) \times τ_3 \simeq τ_1 \times (τ_2 \times τ_3)\)
- Identity: \(1 \times τ \simeq τ\)
- Commutativity: \(τ_1 \times τ_2 \simeq τ_2 \times τ_1\)

Combining \(\times\) and \(+\):
- Distributivity: \(τ_1 \times (τ_2 + τ_3) \simeq (τ_1 \times τ_2) + (τ_1 \times τ_3)\)
- Absorption: \(0 \times τ \simeq 0\)
Semiring Structure

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

Laws for \((\tau, +, 0)\):
- **Associativity**: \((\tau_1 + \tau_2) + \tau_3 \simeq \tau_1 + (\tau_2 + \tau_3)\)
- **Identity**: \(0 + \tau \simeq \tau\)
- **Commutativity**: \(\tau_1 + \tau_2 \simeq \tau_2 + \tau_1\)

Laws for \((\tau, \times, 1)\):
- **Associativity**: \((\tau_1 \times \tau_2) \times \tau_3 \simeq \tau_1 \times (\tau_2 \times \tau_3)\)
- **Identity**: \(1 \times \tau \simeq \tau\)
- **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**: \(0 \times \tau \simeq 0\)

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:

\[
\begin{align*}
1 \times (\text{Int} \times \text{Int}) \\
+ & \ 1 \times \text{Int} \\
+ & \ 1 \\
+ & \ 1 \times (\text{Int} \times (\text{Int} \times \text{Int}))
\end{align*}
\]

\[
\simeq
\]

\[
\begin{align*}
\text{Int} \times \text{Int} \\
+ & \ \text{Int} \\
+ & \ \text{Int} \times (\text{Int} \times \text{Int})
\end{align*}
\]
Examining our Types

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

\[
\begin{align*}
\Gamma &\vdash e : 0 \\
\Gamma &\vdash \text{absurd} \ e : \tau \\
\Gamma &\vdash (\ ) : 1 \\
\Gamma &\vdash e : \tau_1 \\
\Gamma &\vdash \text{InL} \ e : \tau_1 + \tau_2 \\
\Gamma &\vdash e : \tau_1 + \tau_2 \\
\Gamma &\vdash x : \tau_1, \Gamma \vdash e_1 : \tau \\
\Gamma &\vdash y : \tau_2, \Gamma \vdash e_2 : \tau \\
\Gamma &\vdash (\text{case } e \text{ of } \text{InL} \ x \rightarrow e_1; \text{InR} \ y \rightarrow e_2) : \tau \\
\Gamma &\vdash e_1 : \tau_1 \\
\Gamma &\vdash e_2 : \tau_2 \\
\Gamma &\vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\Gamma &\vdash e : \tau_1 \times \tau_2 \\
\Gamma &\vdash \text{fst} \ e : \tau_1 \\
\Gamma &\vdash e_1 : \tau_1 \rightarrow \tau_2 \\
\Gamma &\vdash e_2 : \tau_1 \\
\Gamma &\vdash e_1 \ e_2 : \tau_2 \\
\Gamma &\vdash \lambda x. \ e : \tau_1 \rightarrow \tau_2 \\
\Gamma &\vdash e : \tau_2 \\
\end{align*}
\]
Squinting a Little

Let's remove all the terms, leaving just the types and the contexts:

\[
\begin{align*}
\Gamma & \vdash 0 & \Gamma & \vdash 1 \\
\hline
\Gamma & \vdash \tau & \Gamma & \vdash \tau \\
\hline
\Gamma & \vdash \tau_1 & \Gamma & \vdash \tau_2 \\
\hline
\Gamma & \vdash \tau_1 + \tau_2 & \Gamma & \vdash \tau_1 + \tau_2 \\
\hline
\Gamma & \vdash \tau_1 + \tau_2 & \Gamma & \vdash \tau_1 + \tau_2 \\
& \hline
\tau_1, \Gamma & \vdash \tau & \tau_2, \Gamma & \vdash \tau \\
& \hline
\Gamma & \vdash \tau \\
\hline
\Gamma & \vdash \tau_1 & \Gamma & \vdash \tau_2 \\
\hline
\Gamma & \vdash \tau_1 \times \tau_2 & \Gamma & \vdash \tau_1 \times \tau_2 \\
& \hline
\Gamma & \vdash \tau_1 \times \tau_2 & \Gamma & \vdash \tau_1 \times \tau_2 \\
& \hline
\Gamma & \vdash \tau_1 \rightarrow \tau_2 & \Gamma & \vdash \tau_1 \rightarrow \tau_2 \\
& \hline
\tau_1, \Gamma & \vdash \tau_2 & \Gamma & \vdash \tau_1 \rightarrow \tau_2 \\
& \hline
\Gamma & \vdash \tau_2 & \Gamma & \vdash \tau_2
\end{align*}
\]

Does this resemble anything you've seen before?
A surprising coincidence!

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

\[
\begin{align*}
\Gamma \vdash \bot & \quad \Gamma \vdash \top \\
\Gamma \vdash P_1 & \quad \Gamma \vdash P_2 \\
\Gamma \vdash P_1 \lor P_2 & \\
\Gamma \vdash P_1 \land P_2 & \quad P_1, \Gamma \vdash P & \quad P_2, \Gamma \vdash P \\
\Gamma \vdash P & \\
\Gamma \vdash P_1 \rightarrow P_2 & \quad \Gamma \vdash P_1 & \quad P_1, \Gamma \vdash P_2 & \quad \Gamma \vdash P_1 \rightarrow P_2
\end{align*}
\]

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:

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Propositions</td>
</tr>
<tr>
<td>Programs</td>
<td>Proofs</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Proof Simplification</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Programming</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Types</td>
<td>Propositions</td>
</tr>
<tr>
<td>Programs</td>
<td>Proofs</td>
</tr>
<tr>
<td>Evaluation</td>
<td>Proof Simplification</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Constructive Logic</th>
<th>Typed $\lambda$-Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Logic</td>
<td>Continuations</td>
</tr>
<tr>
<td>Modal Logic</td>
<td>Monads</td>
</tr>
<tr>
<td>Linear Logic</td>
<td>Linear Types, Session Types</td>
</tr>
<tr>
<td>Separation Logic</td>
<td>Region Types</td>
</tr>
</tbody>
</table>
Examples

Example (Commutativity of Conjunction)

\[ \text{andComm} :: A \times B \to B \times A \]
\[ \text{andComm } p = (\text{snd } p, \text{fst } p) \]

This proves \( A \land B \to B \land A \).
Examples

Example (Commutativity of Conjunction)

\[
\text{andComm} :: A \times B \rightarrow B \times A \\
\text{andComm } p = (\text{snd } p, \text{fst } p)
\]

This proves \( A \land B \rightarrow B \land A \). 

Example (Transitivity of Implication)

\[
\text{transitive} :: (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) \\
\text{transitive } f \ g \ x = g (f \ x)
\]

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:

\[
\text{proof}_1 :: P = \text{NP} \\
\text{proof}_1 = \text{proof}_1
\]

\[
\text{proof}_2 :: P \neq \text{NP} \\
\text{proof}_2 = \text{proof}_2
\]

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?

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

We can't express these in MinHS yet:

```
1 + (Int × ??)
```
Inductive Structures

What about types like lists?

data IntList = Nil | Cons Int IntList

We can't express these in MinHS yet:

\[ 1 + (\text{Int} \times \text{??}) \]

We need a way to do recursion!
Recursive Types

We introduce a new form of type, written \( \text{rec } t. \, \tau \), that allows us to refer to the entire type:

\[
\text{IntList} \cong \text{rec } t. 1 + (\text{Int} \times t)
\]
Recursive Types

We introduce a new form of type, written \( \text{rec } t. \, \tau \), that allows us to refer to the entire type:

\[
\text{IntList} \quad \simeq \quad \text{rec } t. \, 1 + (\text{Int} \times t) \\
\quad \simeq \quad 1 + (\text{Int} \times (\text{rec } t. \, 1 + (\text{Int} \times t))))
\]
Recursive Types

We introduce a new form of type, written \( \text{rec } t. \tau \), that allows us to refer to the entire type:

\[
\begin{align*}
\text{IntList} & \cong \text{rec } t. 1 + (\text{Int} \times t) \\
& \cong 1 + (\text{Int} \times (\text{rec } t. 1 + (\text{Int} \times t))) \\
& \cong 1 + (\text{Int} \times (1 + (\text{Int} \times (\text{rec } t. 1 + (\text{Int} \times t))))) \\
& \cong ... 
\end{align*}
\]
Typing Rules

We construct a recursive type with \textit{roll}, and unpack the recursion one level with \textit{unroll}:

\[
\begin{align*}
\Gamma & \vdash \text{roll } e : \text{rec } t. \tau \\
\end{align*}
\]
Typing Rules

We construct a recursive type with \textit{roll}, and unpack the recursion one level with \textit{unroll}:

\[
\Gamma \vdash e : \tau[t := \text{rec } t. \tau]
\]

\[
\Gamma \vdash \text{roll } e : \text{rec } t. \tau
\]
Typing Rules

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

\[
\begin{align*}
\Gamma \vdash e : \tau[t := \text{rec } t. \, \tau] \\
\Gamma \vdash \text{roll } e : \text{rec } t. \, \tau \\
\Gamma \vdash e : \text{rec } t. \, \tau \\
\Gamma \vdash \text{unroll } e : \tau[t := \text{rec } t. \, \tau]
\end{align*}
\]
Example

Take our IntList example:

\[
\text{rec } t. 1 + (\text{Int } \times t)
\]

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

Take our IntList example:

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

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

Take our IntList example:

\[\text{rec } t. \ 1 + (\text{Int} \times t)\]

\[
\begin{align*}
[] & = \text{roll (InL (\text{()}) )} \\
[1] & = \text{roll (InR (1, roll (InL (\text{()}) ) ) )} \\
[1, 2] & =
\end{align*}
\]
Example

Take our IntList example:

\[ \text{rec } t. \ 1 + (\text{Int} \times t) \]

\[
\begin{align*}
[] &= \text{roll (InL ()}) \\
[1] &= \text{roll (InR (1, roll (InL ()))))} \\
[1, 2] &= \text{roll (InR (1, roll (InR (2, roll (InL ()))))))}
\end{align*}
\]
Dynamic Semantics

Nothing interesting here:

\[
\begin{align*}
  e & \mapsto_M e' \\
  \text{roll } e & \mapsto_M \text{roll } e' \\
  e & \mapsto_M e' \\
  \text{unroll } e & \mapsto_M \text{unroll } e' \\
  \text{unroll } (\text{roll } e) & \mapsto_M e
\end{align*}
\]