Algebraic Data T

Christine Rizkallah
CSE, UNSW
Term 3 2020

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:

Tuples

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 Datatypes

C Structs

Has

```
type Point
midpoint (
  = ((x1+x2
```

```
typ
  f
  f
} poi
poin
  poi
  mid
  mid
  ret
}
```

```
  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*:

We won't have type declarations, named fields or anything lik values can be combined by nesting products, for example a thre

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

Constructors and Eliminators

We can construct a product type similar to Haskell tuples:

The only way to extr fst and snd eliminators:

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

Examples

Example (Midpoint)

```
recfun midpoint ::
    ((Int   Int)   (Int   Int)   (Int   Int)) p_1 =
```

Example (Uncurried Division)

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

Dynamic Semantics

$$\frac{e \quad e'}{\quad} \qquad \frac{e \quad e'}{\quad}$$

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

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

Unit Types

Currently, we hav
useless at first, but it
We'll introduce a t

ritten ()

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

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 t
contain different

Example (Mor

```
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.

This is similar to the Haskell `Either` type.
Our `TrafficLight` type can be expressed (grotesque [illegible]

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

Constructors and Eliminators for Sums

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

$$\frac{\Gamma \vdash e : \tau_1}{\quad} \qquad \frac{\Gamma \vdash e : \tau_2}{\quad}$$

We can branch bas

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \mathsf{InL}\ x \to e_1}$$

(Using concrete syntax here, for readability.)

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

Examples

Example (Traffic Lights)

Our traffic light ty

$$\begin{aligned}
\texttt{Red} &\simeq 1\\
\texttt{Amber} &\simeq 1\\
\texttt{Green} &\simeq 1
\end{aligned}$$

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
```

$$\Downarrow$$

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

Dynamic Semantics

$$\frac{e \mapsto_M e'}{\quad}\qquad\frac{e \mapsto_M e'}{\quad}$$

$$\frac{\quad}{(\textbf{case}\ e\ \textbf{of}\ \mathsf{InL}\ x.\ e_1; \mathsf{InR}\ y.\ e_2) \mapsto_M (\textbf{c} \quad {}_2)}$$

$$\frac{\quad}{(\textbf{case}\ (\mathsf{InL}\ v)\ \textbf{of}\ \mathsf{InL}\ x.\ e_1; \mathsf{InR}}$$

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

The Empty Type

We add another type, called **0**, that has no inhabitants. Because it is empty, there is no way to construct it.
We do have a way to el [illegible]

$$\frac{\text{[illegible]}}{\Gamma \vdash \mathsf{absurd}\ e}$$

If I have a variable of the empty type in scope, we must be looking at an e [illegible]
that will never be evaluated. Therefore, we can assign any type w [illegible]
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: $(\tau_1 + \tau_2) + \tau_3 \quad \tau_1 + (\tau_2 + \tau_3)$
- Identity:
- Commuta

Laws for $(\tau, \times$

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

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

$$\simeq$$

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

Examining our Types

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

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{case}\ e\ \textbf{of}\ \textsf{InL}\ x \to e_1 \ \ldots}$$

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

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

Squinting a Little

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

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

$$\frac{[illegible]}{[illegible]} \qquad \frac{[illegible]}{[illegible]}$$

$$\frac{[illegible]_1 \quad [illegible]_2 \quad [illegible]_1 \quad [illegible]_2}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash \tau_1}{\Gamma \vdash \tau_1} \qquad \frac{[illegible]}{[illegible]_2}$$

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

Does this resemble anything you've seen before?

A surprising coincidence!

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

$$\frac{\Gamma \vdash \text{[illegible]}}{\Gamma \vdash P} \qquad \frac{}{\Gamma \vdash \top}$$

$$\frac{}{\text{[illegible]}} \qquad \frac{}{\text{[illegible]}}$$

$$\frac{\text{[illegible]}}{\Gamma \vdash P}$$

$$\frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \land P_2} \qquad \frac{\Gamma \vdash P_1}{\Gamma \vdash \text{[illegible]}_1} \qquad \frac{\text{[illegible]}}{\text{[illegible]}_2}$$

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

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:

It turns out, no matter what logic you want to define, there is alway

λ-calculus, and vice versa.

Constructive Logic	Ty
Classical Logic	Continuations
Modal Logic	Monads
Linear Logic	Linear Types, Session Types
Separation Logic	Region Types

Examples

Example (Commutativity of Conjunction)

$$andComm :: A \quad B \quad\quad B \quad A$$

This proves A

Example (Transitivity of Implication)

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

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:

$$proof_1 :: \mathsf{P} = \mathsf{NP}$$

$$proof_2 = pro$$

Most common calculi correspond to constructive logic, not cl
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 t

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

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:

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

Typing Rules

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

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

Example

Example

Take our IntL

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

Dynamic Semantics

Nothing interesting here:

$$\frac{\mapsto}{\mapsto} \qquad \frac{\mapsto}{\mapsto}$$

$$\frac{}{\textsf{unroll}\ (\textsf{roll}\ e) \mapsto}$$