Recursive Types



Algebraic Data Types; Recursive Types

> Rob Sison UNSW Term 3 2024

Recursive Types

Composite Data Types

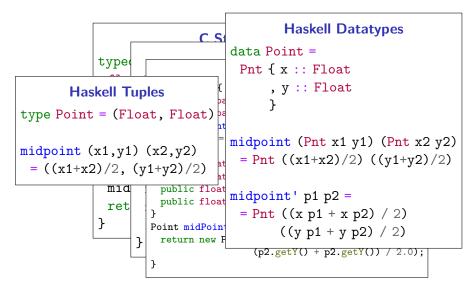
Most of the types we have seen so far are **basic** types, in the sense that they represent built-in machine data representations. Real programming languages have ways to *compose* types to produce new types:



Recursive Types

Combining values conjunctively

We want to store two things in one value.



Recursive Types

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:

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

Recursive Types

Constructors and Eliminators

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

$$\frac{\mathsf{\Gamma} \vdash \mathsf{e}_1 : \tau_1 \qquad \mathsf{\Gamma} \vdash \mathsf{e}_2 : \tau_2}{\mathsf{\Gamma} \vdash (\mathsf{e}_1, \mathsf{e}_2) : \tau_1 \times \tau_2}$$

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

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

Recursive Types

Examples

Example (Midpoint)

recfun midpoint :: $((\operatorname{Int} \times \operatorname{Int}) \rightarrow (\operatorname{Int} \times \operatorname{Int}) \rightarrow (\operatorname{Int} \times \operatorname{Int})) p_1 =$ **recfun** midpoint' :: $((\operatorname{Int} \times \operatorname{Int}) \rightarrow (\operatorname{Int} \times \operatorname{Int})) p_2 =$ $((\operatorname{fst} p_1 + \operatorname{fst} p_2) \div 2, (\operatorname{snd} p_1 + \operatorname{snd} p_2) \div 2)$

Example (Uncurried Division)

recfun div :: ((Int × Int) \rightarrow Int) args = **if** (fst args < snd args) **then** 0 **else** 1 + div (fst args - snd args, snd args) Composite Data

Types as Algebra, Logic

Recursive Types

Dynamic Semantics

$$\frac{e_1 \mapsto_M e'_1}{(e_1, e_2) \mapsto_M (e'_1, e_2)} \qquad \frac{e_2 \mapsto_M e'_2}{(v_1, e_2) \mapsto_M (v_1, e'_2)}$$
$$\frac{e \mapsto e'}{\mathsf{fst} \ e \mapsto_M \mathsf{fst} \ e'} \qquad \frac{e \mapsto e'}{\mathsf{snd} \ e \mapsto_M \mathsf{snd} \ e'}$$
$$\frac{\mathsf{fst} \ (v_1, v_2) \mapsto_M v_1}{\mathsf{snd} \ (v_1, v_2) \mapsto_M v_2}$$

7

Recursive Types

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 ():

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

Recursive Types

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.

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

Recursive Types

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:

 ${ t TrafficLight}\simeq {f 1}+({f 1}+{f 1})$

Constructors and Eliminators for Sums

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

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \ln L \; e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \ln R \; e : \tau_1 + \tau_2}$$

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

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash e_1 : \tau \qquad y : \tau_2, \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\textbf{case } e \text{ of } \ln L \ x \to e_1; \ln R \ y \to e_2) : \tau}$$

Composite Data

Types as Algebra, Logic

Recursive Types

Examples

Example (Traffic Lights)

Our traffic light type has three values as required:

TrafficLight	\simeq	1 + ((1+1))
--------------	----------	-------	-------	---

Red	\simeq	InL ()
Amber	\simeq	lnR(lnL())
Green	\simeq	lnR(lnR())

Examples

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

- $\textcircled{0} \quad {\sf Replace all constructors with 1}$
- 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 \times (Int \times Int)

+ 1 \times Int + 1

+ 1 \times (Int \times (Int \times Int))
```

Composite Data

Types as Algebra, Logic

Recursive Types

Dynamic Semantics

$$\frac{e \mapsto_M e'}{\ln L \ e \mapsto_M \ln L \ e'} \qquad \frac{e \mapsto_M e'}{\ln R \ e \mapsto_M \ln R \ e'}$$

$$e\mapsto_M e'$$

(case e of lnL x. e_1 ; lnR y. e_2) \mapsto_M (case e' of lnL x. e_1 ; lnR y. e_2)

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

(case (lnR v) of lnL x. e_1 ; lnR y. e_2) $\mapsto_M e_2[y := v]$

Recursive Types

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 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. Types as Algebra, Logic ●○○○○○○○ Recursive Types

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: $\mathbf{1} imes au \ \simeq \ au$
- 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?

Types as Algebra, Logic ○●○○○○○○

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:

```
1 \times (\text{Int} \times \text{Int})
+ 1 \times \text{Int} + 1
+ \mathbf{1} \times (\texttt{Int} \times (\texttt{Int} \times \texttt{Int}))
                        \simeq
            Int \times Int
     + Int + 1
     + Int \times (Int \times Int)
```

Types as Algebra, Logic ○○●○○○○○ Recursive Types

Examining our Types

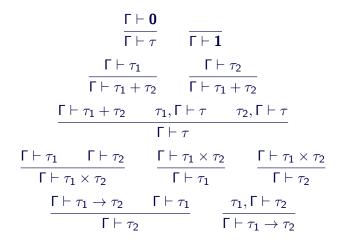
Lets look at the rules for typed lambda calculus extended with sums and products:

${\sf \Gamma} \vdash e: {f 0}$			
$\Gamma \vdash absurd \ e : au$	· Γ⊢():1		
${\sf \Gamma}\vdash {\sf e}:\tau_1$	$\Gamma \vdash e : au_2$		
$\overline{\Gamma\vdashInL\;\boldsymbol{e}:\tau_1+\tau_2}$	$\overline{\Gamma \vdash InR \ e : \tau_1 + \tau_2}$		
$\Gamma \vdash e : \tau_1 + \tau_2 \qquad x : \tau_1, \Gamma \vdash$	$-e_1: au$ $y: au_2, \Gamma \vdash e_2: au$		
$\Gamma \vdash (case \ e \ of \ lnL \ x \rightarrow e_1; lnR \ y \rightarrow e_2) : \tau$			
$\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma$	$\vdash e: \tau_1 \times \tau_2 \qquad \boxed{\Gamma \vdash e: \tau_1 \times \tau_2}$		
$\Gamma \vdash (e_1, e_2) : \tau_1 imes au_2$ Γ	$\vdash fst \ e : \tau_1 \qquad \qquad \Gamma \vdash snd \ e : \tau_2$		
$\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :$	$: \tau_1 \qquad x : \tau_1, \Gamma \vdash e : \tau_2$		
$\Gamma \vdash e_1 \ e_2 : au_2$	$\Gamma \vdash \lambda x. \ e: \tau_1 \rightarrow \tau_2$		

Types as Algebra, Logic ○○○●○○○○ Recursive Types

Squinting a Little

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

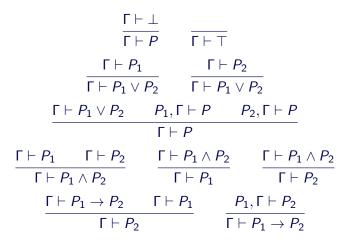


Does this resemble anything you've seen before?

Types as Algebra, Logic ○○○○●○○○ Recursive Types

A surprising coincidence!

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



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 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	

Types as Algebra, Logic ○○○○○○●○ Recursive Types

Examples

Example (Commutativity of Conjunction)

and Comm :: $A \times B \rightarrow B \times A$ and Comm 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.

Types as Algebra, Logic ○○○○○○● Recursive Types

Caveats

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

> $proof_1 :: P = NP$ $proof_1 = proof_1$

> $proof_2 :: P \neq NP$ $proof_2 = proof_2$

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:

 $\neg \neg P \rightarrow P$

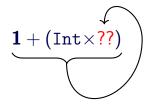
Recursive Types

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

Recursive Types

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

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

Recursive Types

Typing Rules

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

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

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

Composite Data

Types as Algebra, Logic

Recursive Types

Example

Example

Take our IntList example:

```
rec t. 1 + (Int \times t)
```

```
 \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \text{roll}(\text{InL ()}) \\ \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \text{roll}(\text{InR (1, roll (InL ()))}) \\ \begin{bmatrix} 1, 2 \end{bmatrix} = \text{roll}(\text{InR (1, roll (InR (2, roll (InL ())))}))
```

Composite Data

Types as Algebra, Logic

Recursive Types

Dynamic Semantics

Nothing interesting here:

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

unroll (roll e) $\mapsto_M e$