Algebraic Data Types

Hype for Types

January 26, 2026

Outline

- Look at types we already know from a different angle

Outline

- Look at types we already know from a different angle
- Formalize some important new type concepts

Outline

- Look at types we already know from a different angle
- Formalize some important new type concepts
- break the universe

Introduction to Counting

Warning

Be prepared to learn some very serious math such as

$$1 + 2 = 3$$

bool and **order**

Notation

Write $|\tau|$ to denote the number of elements in type τ[a].

[a] this does not work quite well with polymorphism unfortunately.

```
datatype bool  = false | true
datatype order = LESS | EQUAL | GREATER
```

What size are they?

bool and **order**

Notation

Write $|\tau|$ to denote the number of elements in type τ[a].

[a] this does not work quite well with polymorphism unfortunately.

```
datatype bool  = false | true
datatype order = LESS | EQUAL | GREATER
```

What size are they?

$$|\textbf{bool}| = 2$$
$$|\textbf{order}| = 3$$

bool and **order**

Notation

Write $|\tau|$ to denote the number of elements in type τ[a].

[a] this does not work quite well with polymorphism unfortunately.

```
datatype bool  = false | true
datatype order = LESS | EQUAL | GREATER
```

What size are they?

$$|\textbf{bool}| = 2$$
$$|\textbf{order}| = 3$$

Often, we refer to **bool** as 2 and **order** as 3:

$$\texttt{true} : 2$$
$$\texttt{LESS} : 3$$

Products

Products

Question

What is $|\tau_1 \times \tau_2|$?

Products

Question

What is $|\tau_1 \times \tau_2|$?

$|\tau_1| \times |\tau_2|$ - hence, the notation.

Products

Question

What is $|\tau_1 \times \tau_2|$?

$|\tau_1| \times |\tau_2|$ - hence, the notation.

For example,

$$\begin{aligned} |\textbf{bool} \times \textbf{order}| &= |\textbf{bool}| \times |\textbf{order}| \\ &= 2 \times 3 \\ &= 6 \end{aligned}$$

What do you know!

Theorem: Commutativity of Products

For all τ_1, τ_2:

$$\tau_1 \times \tau_2 \simeq \tau_2 \times \tau_1$$

Theorem: Associativity of Products

For all τ_1, τ_2, τ_3:

$$\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$$

What do you know!

Theorem: Commutativity of Products

For all τ_1, τ_2:

$$\tau_1 \times \tau_2 \simeq \tau_2 \times \tau_1$$

Theorem: Associativity of Products

For all τ_1, τ_2, τ_3:

$$\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$$

Question

How do we know?

Proving Type Isomorphisms

To prove that $\tau \simeq \tau'$, we need a *bijection* between τ and τ'.

Proving Type Isomorphisms

To prove that $\tau \simeq \tau'$, we need a *bijection* between τ and τ'.

We write two (total) functions, $f : \tau \to \tau'$ and $f' : \tau' \to \tau$, such that f and f' are *inverses*.

$$\texttt{f' (f x)} \cong \texttt{x}$$

$$\texttt{f (f' x)} \cong \texttt{x}$$

Associativity of Products: Proved!

Let's prove associativity of products:

$$\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$$

Associativity of Products: Proved!

Let's prove associativity of products:

$$\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$$

Need to write:

$$f : \tau_1 \times (\tau_2 \times \tau_3) \to (\tau_1 \times \tau_2) \times \tau_3$$
$$f' : (\tau_1 \times \tau_2) \times \tau_3 \to \tau_1 \times (\tau_2 \times \tau_3)$$

Associativity of Products: Proved!

Let's prove associativity of products:

$$\tau_1 \times (\tau_2 \times \tau_3) \simeq (\tau_1 \times \tau_2) \times \tau_3$$

Need to write:

$$f : \tau_1 \times (\tau_2 \times \tau_3) \to (\tau_1 \times \tau_2) \times \tau_3$$
$$f' : (\tau_1 \times \tau_2) \times \tau_3 \to \tau_1 \times (\tau_2 \times \tau_3)$$

Nice!

$$f = \texttt{fn (a,(b,c)) => ((a,b),c)}$$
$$f' = \texttt{fn ((a,b),c) => (a,(b,c))}$$

Multiplicative Identity?

Follow-Up

Is there an identity element, “1”?

$$\tau \times 1 = \tau$$
$$1 \times \tau = \tau$$

Multiplicative Identity?

Follow-Up

Is there an identity element, “1”?

$$\tau \times 1 = \tau$$
$$1 \times \tau = \tau$$

Yes - **unit**!

Multiplicative Identity?

Follow-Up

Is there an identity element, "1"?

$$\tau \times 1 = \tau$$
$$1 \times \tau = \tau$$

Yes - **unit**!

Theorem

For all types τ:

$$\tau \times \textbf{unit} \simeq \tau$$
$$\textbf{unit} \times \tau \simeq \tau$$

Sums

Increment

Question

Is there such thing as $\tau + 1$?

Increment

Question

Is there such thing as $\tau + 1$?

Answer

Yes! τ **option**.

Increment

Question

Is there such thing as $\tau + 1$?

Answer

Yes! τ **option**.

SOME x	(τ choices)
NONE	(1 choice)

Sums

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```
[1]

[1] In the Standard ML Basis, (almost) the `Either` structure!

Sums

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```
[1]

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{Left}\ e : \tau_1 + \tau_2}\ (\textsc{left}) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{Right}\ e : \tau_1 + \tau_2}\ (\textsc{right})$$

[1] In the Standard ML Basis, (almost) the `Either` structure!

Sums

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```
[1]

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{Left}\ e : \tau_1 + \tau_2}\ (\textsc{left}) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{Right}\ e : \tau_1 + \tau_2}\ (\textsc{right})$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : \tau}\ (\textsc{case})$$

[1] In the Standard ML Basis, (almost) the `Either` structure!

Sums

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```
[1]

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \textbf{Left}\ e : \tau_1 + \tau_2}\ (\textsc{left}) \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \textbf{Right}\ e : \tau_1 + \tau_2}\ (\textsc{right})$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{case}\ e\ \textbf{of}\ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : \tau}\ (\textsc{case})$$

And of course...

For all τ_1, τ_2:

$$|\tau_1 + \tau_2| = |\tau_1| + |\tau_2|$$

[1] In the Standard ML Basis, (almost) the `Either` structure!

Options as Sums

```
datatype ('a,'b) either = Left of 'a | Right of 'b
```

Notice:

```
type 'a option = ('a,unit) either
```

We can represent τ **option** as $\tau + \mathbf{unit}$.

Example: Distributivity

Claim

For all types A, B, C:

$$(A \times B) + (A \times C) \simeq A \times (B + C)$$

Example: Distributivity

Claim

For all types A, B, C:

$$(A \times B) + (A \times C) \simeq A \times (B + C)$$

f : `('a * 'b, 'a * 'c) either -> 'a * ('b,'c) either`

f' : `'a * ('b,'c) either -> ('a * 'b, 'a * 'c) either`

Example: Distributivity

Claim

For all types A, B, C:

$$(A \times B) + (A \times C) \simeq A \times (B + C)$$

f : `('a * 'b, 'a * 'c) either -> 'a * ('b,'c) either`

f' : `'a * ('b,'c) either -> ('a * 'b, 'a * 'c) either`

$f =$ `fn Left (a,b) => (a,Left b) | Right (a,c) => (a,Right c)`

$f' =$ `fn (a,Left b) => Left (a,b) | (a,Right c) => Right (a,c)`

Example: Distributivity

Claim

For all types A, B, C:

$$(A \times B) + (A \times C) \simeq A \times (B + C)$$

f : `('a * 'b, 'a * 'c) either -> 'a * ('b,'c) either`

f' : `'a * ('b,'c) either -> ('a * 'b, 'a * 'c) either`

$f =$ `fn Left (a,b) => (a,Left b) | Right (a,c) => (a,Right c)`

$f' =$ `fn (a,Left b) => Left (a,b) | (a,Right c) => Right (a,c)`

Practical Application

Code refactoring principle! If both cases store the same data, factor it out.

Zero to Hero

If we can add, what's 0?

Zero to Hero

If we can add, what's 0?

We call it **void**, the empty type.[2]

[2] Unlike C's `void` type, which is actually **unit**.

Zero to Hero

If we can add, what's 0?

We call it **void**, the empty type.[2]

void is a type which has no value (terminology is *uninhabited*). How do we construct a type with no value (in SML)?

$$\frac{\Gamma \vdash e : \mathbf{void}}{\Gamma \vdash \mathbf{absurd}(e) : \tau}\ (\textsc{absurd})$$

[2] Unlike C's `void` type, which is actually **unit**.

Zero to Hero

If we can add, what's 0?

We call it **void**, the empty type.[2]

void is a type which has no value (terminology is *uninhabited*). How do we construct a type with no value (in SML)?

$$\frac{\Gamma \vdash e : \mathbf{void}}{\Gamma \vdash \mathbf{absurd}(e) : \tau}\ (\textsc{absurd})$$

Implementing via SML Hacking

```
datatype void = Void of void
fun absurd (Void v) = absurd v
```

Notice: `absurd` is total!

[2] Unlike C's `void` type, which is actually **unit**.

void*

Claim

For all types τ:

$$\tau + \textbf{void} \simeq \tau$$

void*

Claim

For all types τ:

$$\tau + \textbf{void} \simeq \tau$$

f : `('tau,void) either -> 'tau`

f' : `'tau -> ('tau,void) either`

void*

Claim

For all types τ:

$$\tau + \textbf{void} \simeq \tau$$

$$f : \texttt{('tau,void) either -> 'tau}$$
$$f' : \texttt{'tau -> ('tau,void) either}$$

$$f = \texttt{fn Left x => x | Right v => absurd v}$$
$$f' = \texttt{fn x => Left x}$$
$$= \texttt{Left}$$

Functions

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A?

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A? $|B|$

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A? $|B|$
- How many choices for the output of the second?

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A? $|B|$
- How many choices for the output of the second? $|B|$

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A? $|B|$
- How many choices for the output of the second? $|B|$
- By using our cherished Multiplication Principle from concepts ...

How Many Functions?

How many (total) values are there of type $A \to B$, in terms of $|A|$ and $|B|$?

- How many choices for output of first object of type A? $|B|$
- How many choices for the output of the second? $|B|$
- By using our cherished Multiplication Principle from concepts ...

Theorem

There are $|B|^{|A|}$ total functions from type A to type B.

Example: Power of a Power

In math, it's true that:

$$(C^B)^A = C^{A \times B}$$

Example: Power of a Power

In math, it's true that:

$$(C^B)^A = C^{A \times B}$$

In terms of types, that would mean:

$$A \to (B \to C) \simeq A \times B \to C$$

Example: Power of a Power

In math, it's true that:

$$(C^B)^A = C^{A \times B}$$

In terms of types, that would mean:

$$A \to (B \to C) \simeq A \times B \to C$$

Yes!

```
f  = Fn.uncurry : ('a -> 'b -> 'c) -> ('a * 'b -> 'c)
f' = Fn.curry   : ('a * 'b -> 'c) -> ('a -> 'b -> 'c)
```

Recursive Types

Lists

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

Lists

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

```
datatype 'a list = Left of unit | Right of 'a * 'a list
```

Lists

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

```
datatype 'a list = Left of unit | Right of 'a * 'a list
```

```
type 'a list = (unit, 'a * 'a list) either
```

Lists

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

```
datatype 'a list = Left of unit | Right of 'a * 'a list
```

```
type 'a list = (unit, 'a * 'a list) either
```

$$L(\alpha) \simeq \mathbf{unit} + \alpha \times L(\alpha)$$

Lists

```
datatype 'a list = Nil | Cons of 'a * 'a list
```

```
datatype 'a list = Left of unit | Right of 'a * 'a list
```

```
type 'a list = (unit, 'a * 'a list) either
```

$$L(\alpha) \simeq \textbf{unit} + \alpha \times L(\alpha)$$

$$\begin{aligned} L(\alpha) &= 1 + \alpha \times L(\alpha) \\ &= 1 + \alpha \times (1 + \alpha \times L(\alpha)) \\ &= 1 + \alpha + \alpha \times L(\alpha) \\ &= 1 + \alpha + \alpha \times (1 + \alpha \times L(\alpha)) \\ &= 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \end{aligned}$$

Natural Numbers

How many natural numbers are there?

Natural Numbers

How many natural numbers are there?

```
datatype nat = Zero | Succ of nat
```

Natural Numbers

How many natural numbers are there?

```
datatype nat = Zero | Succ of nat
```

$$\textbf{nat} = \textbf{unit} + \textbf{nat}$$

Natural Numbers

How many natural numbers are there?

```
datatype nat = Zero | Succ of nat
```

$$\textbf{nat} = \textbf{unit} + \textbf{nat}$$

$$\textbf{nat} = 1 + 1 + 1 + \cdots = \infty$$

Natural Numbers

How many natural numbers are there?

```
datatype nat = Zero | Succ of nat
```

$$\textbf{nat} = \textbf{unit} + \textbf{nat}$$

$$\textbf{nat} = 1 + 1 + 1 + \cdots = \infty$$

Therefore, we would expect:

$$\infty = 1 + \infty$$

$$\textbf{nat} \simeq \textbf{nat option}$$

Natural Numbers

How many natural numbers are there?

```
datatype nat = Zero | Succ of nat
```

$$\textbf{nat} = \textbf{unit} + \textbf{nat}$$

$$\textbf{nat} = 1 + 1 + 1 + \cdots = \infty$$

Therefore, we would expect:

$$\infty = 1 + \infty$$

$$\textbf{nat} \simeq \textbf{nat option}$$

```
f  = fn Zero => NONE | Succ n => SOME n
f' = fn NONE => Zero | SOME n => Succ n
```

From Data to Proof

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

- describe infinite families of values

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

- describe infinite families of values
- define lists, trees, and numbers

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

- describe infinite families of values
- define lists, trees, and numbers
- explain why some types are infinite

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

- describe infinite families of values
- define lists, trees, and numbers
- explain why some types are infinite

Why We Lingered on Recursive Types

So far, recursive types looked like a way to:

- describe infinite families of values
- define lists, trees, and numbers
- explain why some types are infinite

But that's only half the story.

Recursive Types Imply Induction

When you define a recursive type, you are also choosing a *proof strategy*.

Recursive Types Imply Induction

When you define a recursive type, you are also choosing a *proof strategy*.

$$L(\alpha) \simeq 1 + \alpha \times L(\alpha)$$

Recursive Types Imply Induction

When you define a recursive type, you are also choosing a *proof strategy*.

$$L(\alpha) \simeq 1 + \alpha \times L(\alpha)$$

This equation doesn't just define lists — it tells us how proofs about lists must proceed.

Data Shapes Proof Shapes

Every constructor becomes a proof case:

Data Shapes Proof Shapes

Every constructor becomes a proof case:

- `Nil` $\rightarrow$ base case

Data Shapes Proof Shapes

Every constructor becomes a proof case:

- `Nil` $\rightarrow$ base case
- `Cons` $\rightarrow$ inductive step

Data Shapes Proof Shapes

Every constructor becomes a proof case:

- `Nil` $\rightarrow$ base case
- `Cons` $\rightarrow$ inductive step

Data Shapes Proof Shapes

Every constructor becomes a proof case:

- `Nil` $\rightarrow$ base case
- `Cons` $\rightarrow$ inductive step

The structure of the data *is* the structure of the proof.

This Is Exactly What Theorem Provers Need

A theorem prover must:

This Is Exactly What Theorem Provers Need

A theorem prover must:

- reason about all values of a type

This Is Exactly What Theorem Provers Need

A theorem prover must:

- reason about all values of a type
- ensure proofs terminate

This Is Exactly What Theorem Provers Need

A theorem prover must:

- reason about all values of a type
- ensure proofs terminate
- enforce that recursion makes progress

This Is Exactly What Theorem Provers Need

A theorem prover must:

- reason about all values of a type
- ensure proofs terminate
- enforce that recursion makes progress

This Is Exactly What Theorem Provers Need

A theorem prover must:

- reason about all values of a type
- ensure proofs terminate
- enforce that recursion makes progress

Recursive types give all of this for free.

From Here, the Shift Is Natural

Once types define:

- data
- recursion
- and induction

From Here, the Shift Is Natural

Once types define:

- data
- recursion
- and induction

It becomes natural to ask:

From Here, the Shift Is Natural

Once types define:

- data
- recursion
- and induction

It becomes natural to ask:

What if we used types as the foundation for proofs?

Conclusion

Conclusion

- Figured out the sizes of various types

[3] More on that later...

[4] More on that later, as well!

Conclusion

- Figured out the sizes of various types
- Generalized our type theory to include *sum types* (and **void**)

[3] More on that later...

[4] More on that later, as well!

Conclusion

- Figured out the sizes of various types
- Generalized our type theory to include *sum types* (and **void**)
- Considered *recursive types*[3]

[3] More on that later...

[4] More on that later, as well!

Conclusion

- Figured out the sizes of various types
- Generalized our type theory to include *sum types* (and **void**)
- Considered *recursive types*[3]
- Introduced *recursive type's purpose in theorem proving*[4]

[3] More on that later...

[4] More on that later, as well!