Category Theory

An abstract theory of functional programming

Hype for Types

Jacob Neumann

31 March 2020

Section 1

Motivation

What types does SML have?

We've seen that SML has some kinds of type constructions, and not others:

What types does SML have?

We've seen that SML has some kinds of type constructions, and not others:

- Built-in: `unit`, products, function types, lists, options

What types does SML have?

We've seen that SML has some kinds of type constructions, and not others:

- Built-in: `unit`, products, function types, lists, options
- Definable: `void`, sums, trees, streams

What types does SML have?

We've seen that SML has some kinds of type constructions, and not others:

- Built-in: `unit`, products, function types, lists, options
- Definable: `void`, sums, trees, streams
- Not definable: GADTs, dependent types, higher-inductive types

What types does SML have?

We've seen that SML has some kinds of type constructions, and not others:

- Built-in: `unit`, products, function types, lists, options
- Definable: `void`, sums, trees, streams
- Not definable: GADTs, dependent types, higher-inductive types

But what does it mean for SML to "have" a certain type?

```
type 'a list = 'a * bool
```

Think of the type system as a mathematical object

Answer: Types can be defined by their relationship to other types

Think of the type system as a mathematical object

Answer: Types can be defined by their relationship to other types

We think of the type system as a mathematical object in its own right,

Think of the type system as a mathematical object

Answer: Types can be defined by their relationship to other types

We think of the type system as a mathematical object in its own right,consisting of

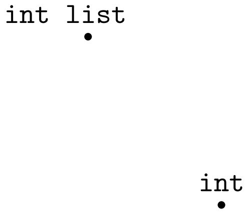
- Types
- Arrows between those types: total functions

A Bird's-Eye View of SML

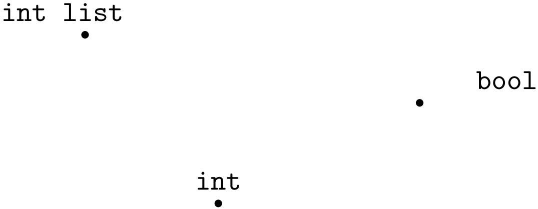
A Bird's-Eye View of SML

int list
•

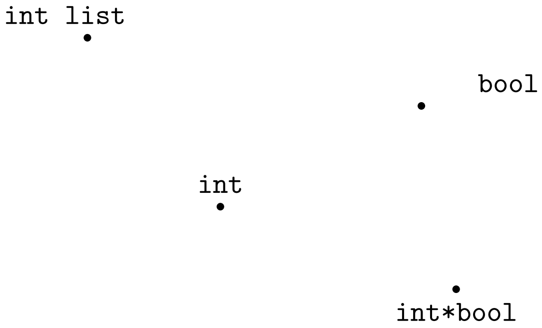
A Bird's-Eye View of SML



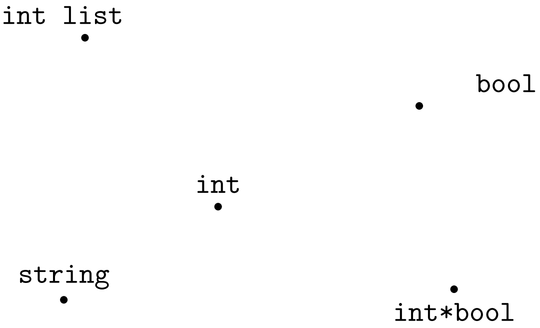
A Bird's-Eye View of SML



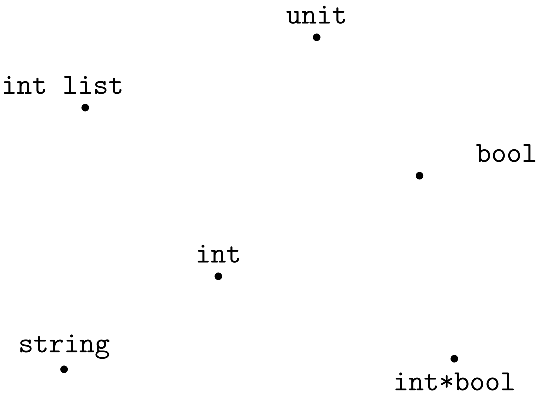
A Bird's-Eye View of SML



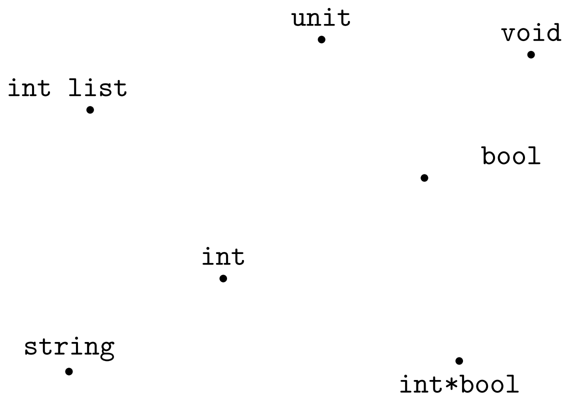
A Bird's-Eye View of SML



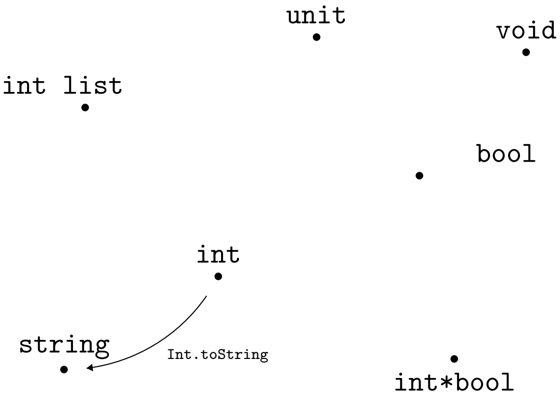
A Bird's-Eye View of SML



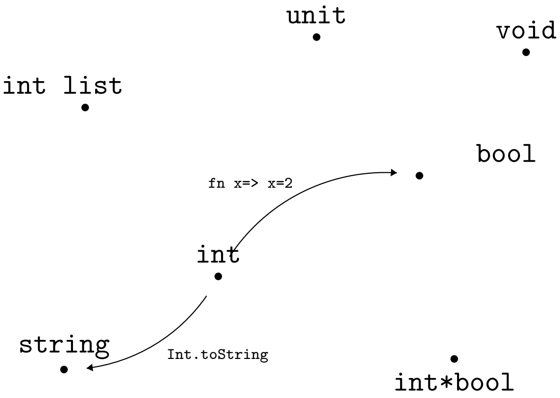
A Bird's-Eye View of SML



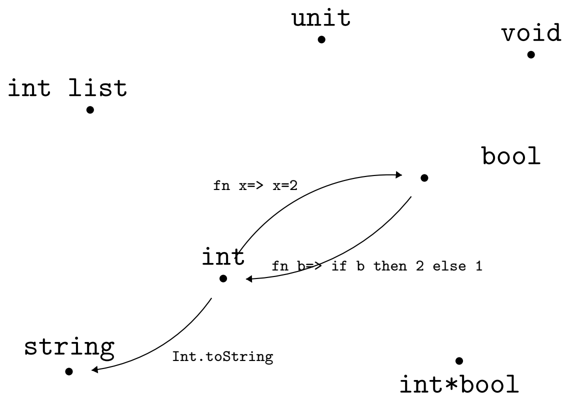
A Bird's-Eye View of SML

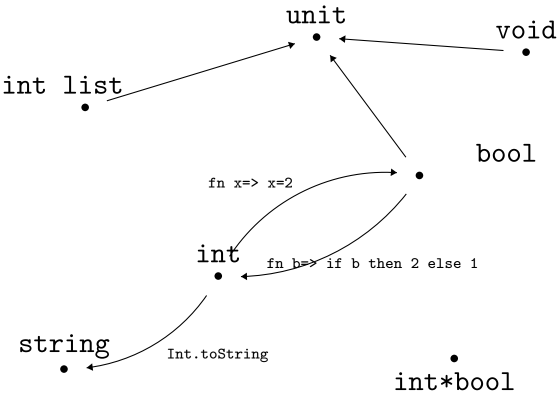


A Bird's-Eye View of SML

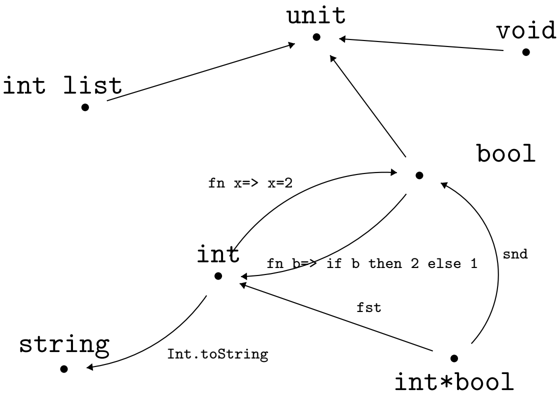


A Bird's-Eye View of SML




unit
void
int list
bool
fn x=> x=2
int
fn b=> if b then 2 else 1
string
Int.toString
int*bool

A Bird's-Eye View of SML



Theorem

There exists a total function of type $\mathit{int} \to \mathit{bool}$.

Theorem

There exists a total function of type $\mathtt{int} \to \mathtt{bool}$.

Theorem

For all types τ, τ', *there exist total functions* $\mathtt{fst} : \tau * \tau' \to \tau$ *and* $\mathtt{snd} : \tau * \tau' \to \tau'$

Theorem

There exists a total function of type $\mathtt{int} \to \mathtt{bool}$.

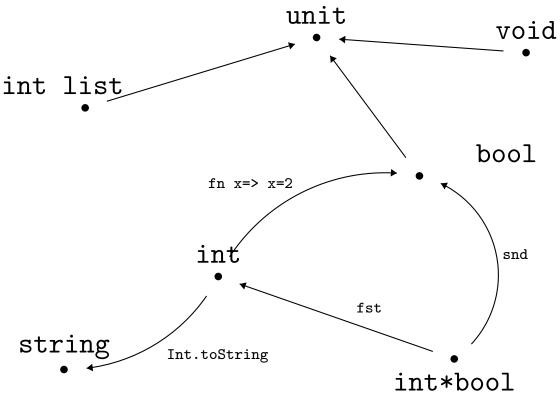
Theorem

For all types τ, τ', *there exist total functions* $\mathtt{fst} : \tau * \tau' \to \tau$ *and* $\mathtt{snd} : \tau * \tau' \to \tau'$

Theorem

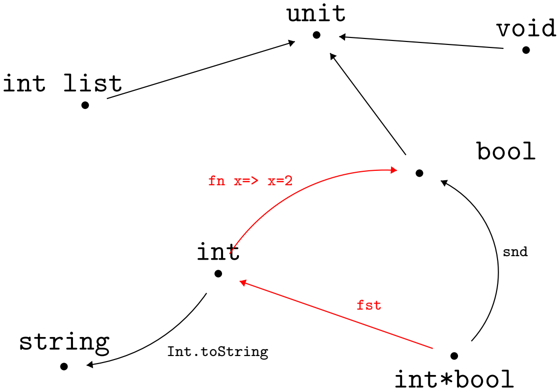
For all types τ, *there exists a unique function* $\mathtt{u}_\tau : \tau \to \mathtt{unit}$

op o is a (partial) binary operation on functions



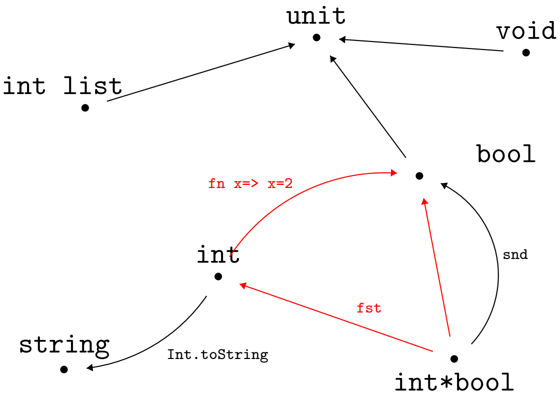
`(op o):('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`

op o is a (partial) binary operation on functions



`(op o):('b -> 'c) * ('a -> 'b) -> ('a -> 'c)`

op o is a (partial) binary operation on functions



(op o):('b -> 'c) * ('a -> 'b) -> ('a -> 'c)

op o Theory

op o Theory

Notice:

```
(fn x=> x=2) o (fn b=> if b then 2 else 1) = id_bool
```

This is an equation of functions, and it tells us information about the types `bool` and `int`.

op o Theory

Notice:

$$\texttt{(fn x=> x=2) o (fn b=> if b then 2 else 1)} = \texttt{id}_{\texttt{bool}}$$

This is an equation of functions, and it tells us information about the types `bool` and `int`.
op o is important enough that it has it's own theory.

op o Theory

Notice:

`(fn x=> x=2) o (fn b=> if b then 2 else 1)` $= \texttt{id}_{\texttt{bool}}$

This is an equation of functions, and it tells us information about the types `bool` and `int`.

`op o` is important enough that it has it's own theory.

The theory of `op o` is called **category theory**.

Section 2

Categories

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \dots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

such that:

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

such that:

1. For each object X, there exists an arrow $\mathrm{id}_X : X \to X$ called the *identity arrow* (on X)

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

such that:

1. For each object X, there exists an arrow $\mathrm{id}_X : X \to X$ called the *identity arrow* (on X)
2. For each pair of arrow $f : X \to Y$ and $g : Y \to Z$, there exists a arrow $(g \circ f) : X \to Z$, called the *composition* of g *after* f

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

such that:

1. For each object X, there exists an arrow $\mathrm{id}_X : X \to X$ called the *identity arrow* (on X)
2. For each pair of arrow $f : X \to Y$ and $g : Y \to Z$, there exists a arrow $(g \circ f) : X \to Z$, called the *composition* of g *after* f
3. Identity arrows are units under composition: for all $f : X \to Y$,

$$\mathrm{id}_Y \circ f = f = f \circ \mathrm{id}_X$$

Definition

A **category** $\mathbb{C}$ consists of:

- A collection of objects $X, Y, Z, \ldots$
- A collection of arrows f, g, h, each with a specified domain and codomain (e.g. $f : X \to Z$)

such that:

1. For each object X, there exists an arrow $\mathrm{id}_X : X \to X$ called the *identity arrow* (on X)
2. For each pair of arrow $f : X \to Y$ and $g : Y \to Z$, there exists a arrow $(g \circ f) : X \to Z$, called the *composition* of g *after* f
3. Identity arrows are units under composition: for all $f : X \to Y$,

$$\mathrm{id}_Y \circ f = f = f \circ \mathrm{id}_X$$

4. Composition is associative: for all $f : A \to B$, $g : B \to C$, $h : C \to D$,

$$(h \circ g) \circ f = h \circ (g \circ f)$$

Example 0: Types!

The type system of SML defines a category:

Example 0: Types!

The type system of SML defines a category:

- The objects are types

Example 0: Types!

The type system of SML defines a category:

- The objects are types
- The arrows are total functions

Example 0: Types!

The type system of SML defines a category:

- The objects are types
- The arrows are total functions

1. For any type τ, the identity arrow on τ is given by:

$$\texttt{val id}_\tau : \tau \to \tau \texttt{ = fn x:}\tau\texttt{ => x}$$

Example 0: Types!

The type system of SML defines a category:

- The objects are types
- The arrows are total functions

1. For any type τ, the identity arrow on τ is given by:

   $$\texttt{val id}_\tau : \tau \to \tau \texttt{ = fn x:}\tau \texttt{ => x}$$

2. For `f:X->Y`, `g:Y->Z`, `(g o f):X-> Z`

Example 0: Types!

The type system of SML defines a category:

- The objects are types
- The arrows are total functions

1. For any type τ, the identity arrow on τ is given by:

$$\texttt{val id}_\tau : \tau \to \tau \texttt{ = fn x:}\tau \texttt{ => x}$$

2. For `f:X->Y`, `g:Y->Z`, `(g o f):X-> Z`
3. $$\texttt{id}_\texttt{Y} \texttt{ o f} = \texttt{f} = \texttt{f o id}_\texttt{X}$$

Example 0: Types!

The type system of SML defines a category:

- The objects are types
- The arrows are total functions

1. For any type τ, the identity arrow on τ is given by:

$$\texttt{val id}_\tau : \tau \to \tau \texttt{ = fn x:}\tau \texttt{ => x}$$

2. For `f:X->Y`, `g:Y->Z`, `(g o f):X-> Z`
3. $$\texttt{id}_\texttt{Y} \texttt{ o f} = \texttt{f} = \texttt{f o id}_\texttt{X}$$
4. `op o` is associative

Example 1: Set

The collection of *all sets* is a category:

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets
- The arrows are total functions

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets
- The arrows are total functions

1. For any set X, the function $\mathrm{id}_X : X \to X$ given by $\mathrm{id}_X(x) = x$ is the identity function

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets
- The arrows are total functions

1. For any set X, the function $\mathrm{id}_X : X \to X$ given by $\mathrm{id}_X(x) = x$ is the identity function
2. Composition is just the usual composition of functions.

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets
- The arrows are total functions

1. For any set X, the function $\mathrm{id}_X : X \to X$ given by $\mathrm{id}_X(x) = x$ is the identity function
2. Composition is just the usual composition of functions.
3. Identity is a unit for composition

Example 1: Set

The collection of *all sets* is a category:

- The objects are sets
- The arrows are total functions

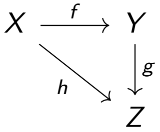
1. For any set X, the function $\mathrm{id}_X : X \to X$ given by $\mathrm{id}_X(x) = x$ is the identity function
2. Composition is just the usual composition of functions.
3. Identity is a unit for composition
4. Function composition is associative

When working in a category, the only kind of equations we can write down are equalities between arrows. So all our definitions must be phrased just using $=$, $\circ$, and arrows.

Categorical Definitions

When working in a category, the only kind of equations we can write down are equalities between arrows. So all our definitions must be phrased just using $=$, $\circ$, and arrows.
Consider the following diagram (in some category $\mathbb{C}$):

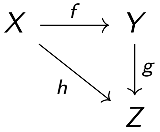
When working in a category, the only kind of equations we can write down are equalities between arrows. So all our definitions must be phrased just using $=$, $\circ$, and arrows.
Consider the following diagram (in some category $\mathbb{C}$):



Categorical Definitions

When working in a category, the only kind of equations we can write down are equalities between arrows. So all our definitions must be phrased just using $=$, $\circ$, and arrows.

Consider the following diagram (in some category $\mathbb{C}$):

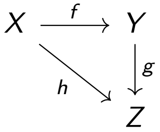


What does it mean to say that this diagram commutes?

Categorical Definitions

When working in a category, the only kind of equations we can write down are equalities between arrows. So all our definitions must be phrased just using $=$, $\circ$, and arrows.

Consider the following diagram (in some category $\mathbb{C}$):



What does it mean to say that this diagram commutes?

$$g \circ f = h$$

Example: Commutative Squares

Consider the following diagram (in some category $\mathbb{C}$):

Example: Commutative Squares

Consider the following diagram (in some category $\mathbb{C}$):

$$\begin{array}{ccc} W & \xrightarrow{f} & X \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle g} \\ Y & \xrightarrow[k]{} & Z \end{array}$$

Example: Commutative Squares

Consider the following diagram (in some category $\mathbb{C}$):

$$\begin{array}{ccc} W & \xrightarrow{f} & X \\ {\scriptstyle h}\big\downarrow & & \big\downarrow{\scriptstyle g} \\ Y & \xrightarrow[k]{} & Z \end{array}$$

What does it mean to say that this diagram commutes?

Example: Commutative Squares

Consider the following diagram (in some category $\mathbb{C}$):

$$\begin{array}{ccc} W & \xrightarrow{f} & X \\ {\scriptstyle h}\big\downarrow & & \big\downarrow{\scriptstyle g} \\ Y & \xrightarrow[k]{} & Z \end{array}$$

What does it mean to say that this diagram commutes?

$$g \circ f = k \circ h$$

Section 3

Universal Mapping Properties

Terminal Objects

Two claims:

- For every type τ, there exists a unique total function of type $\tau \to \texttt{unit}$.

Terminal Objects

Two claims:

- For every type τ, there exists a unique total function of type $\tau \rightarrow \texttt{unit}$.
- For every type τ, there is a bijection between the elements $t : \tau$ and the functions $\texttt{unit} \rightarrow \tau$.

Terminal Objects

Two claims:

- For every type τ, there exists a unique total function of type $\tau \to \texttt{unit}$.

- For every type τ, there is a bijection between the elements $t : \tau$ and the functions $\texttt{unit} \to \tau$.

$$\texttt{const} : \tau \to (\texttt{unit} \to \tau)$$
$$\texttt{ev}_{()} : (\texttt{unit} \to \tau) \to \tau$$

$\texttt{const} \circ \texttt{ev}_{()} = \texttt{id}$ and $\texttt{ev}_{()} \circ \texttt{const} = \texttt{id}$.

Terminal Objects

Two claims:

- For every type τ, there exists a unique total function of type $\tau \to \texttt{unit}$.

- For every type τ, there is a bijection between the elements $t : \tau$ and the functions $\texttt{unit} \to \tau$.

$$\texttt{const} : \tau \to (\texttt{unit} \to \tau)$$
$$\texttt{ev}_{()} : (\texttt{unit} \to \tau) \to \tau$$

  $\texttt{const} \circ \texttt{ev}_{()} = \texttt{id}$ and $\texttt{ev}_{()} \circ \texttt{const} = \texttt{id}$.

An object which has these properties is called a *terminal object*.

Initial Objects

An object V in a category is called *initial* if for every other object X, there exists a unique arrow $V \to X$. Does the SML type system have an initial type?

Initial Objects

An object V in a category is called *initial* if for every other object X, there exists a unique arrow $V \to X$. Does the SML type system have an initial type?

Yes! The type `datatype void = Void of void` is initial, because, given any other type τ, the function

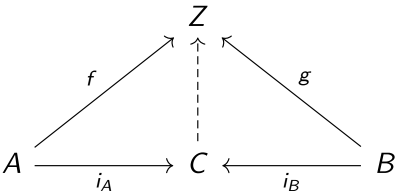
$$\texttt{(fn \_ => raise Fail "Won't happen")} : \texttt{void} \to \tau$$

Coproducts

Given objects A, B of a category, we say that the object C is a *coproduct* of A and B if there exist arrows $i_A : A \to C$ and $i_B : B \to P$ such that

Coproducts

Given objects A, B of a category, we say that the object C is a *coproduct* of A and B if there exist arrows $i_A : A \to C$ and $i_B : B \to P$ such that



For every object Z and arrows $f : A \to Z$ and $g : B \to Z$, there exists a unique arrow $h : C \to Z$ such that

$$f = h \circ i_A \qquad \text{and} \qquad g = h \circ i_B$$

Sum Types are Coproducts

If τ, σ are types, let $\tau + \sigma$ denote the type `(τ,σ) either`, where

```
datatype ('a,'b) either = inL of 'a | inR of 'b
```

Sum Types are Coproducts

If τ, σ are types, let $\tau + \sigma$ denote the type `(`τ`,`σ`) either`, where

```
datatype ('a,'b) either = inL of 'a | inR of 'b
```

Then $\tau + \sigma$ is a coproduct of τ and σ:

$$\begin{aligned} C &= \tau + \sigma \\ i_\tau &= \texttt{inL} \\ i_\sigma &= \texttt{inR} \end{aligned}$$

Sum Types are Coproducts

If τ, σ are types, let $\tau + \sigma$ denote the type `(τ,σ) either`, where

```
datatype ('a,'b) either = inL of 'a | inR of 'b
```

Then $\tau + \sigma$ is a coproduct of τ and σ:

$$C = \tau + \sigma$$
$$i_\tau = \texttt{inL}$$
$$i_\sigma = \texttt{inR}$$

and for any type ρ and any `f : ` $\tau \to \rho$, `g : ` $\sigma \to \rho$,

$$\texttt{h = (fn (inL x) => f(x) | (inR y) => g(y))} : \tau + \sigma \to \rho$$

Sum Types are Coproducts

If τ, σ are types, let $\tau + \sigma$ denote the type `(`τ`,`σ`) either`, where

```
datatype ('a,'b) either = inL of 'a | inR of 'b
```

Then $\tau + \sigma$ is a coproduct of τ and σ:

$$C = \tau + \sigma$$
$$i_\tau = \mathtt{inL}$$
$$i_\sigma = \mathtt{inR}$$

and for any type ρ and any `f :` $\tau \to \rho$, `g :` $\sigma \to \rho$,

`h = (fn (inL x) => f(x) | (inR y) => g(y))` $: \tau + \sigma \to \rho$

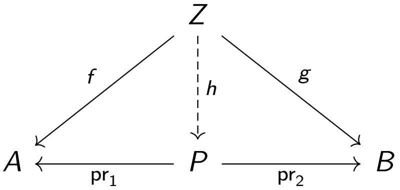
You can check: `h o inL` $=$ `f` and `h o inR` $=$ `g`.

Products

Given objects A, B of a category, we say that the object P is a *product* of A and B if there exist arrows $\mathrm{pr}_1 : A \to C$ and $\mathrm{pr}_2 : B \to P$ such that

Products

Given objects A, B of a category, we say that the object P is a *product* of A and B if there exist arrows $\mathrm{pr}_1 : A \to C$ and $\mathrm{pr}_2 : B \to P$ such that



For every object Z and arrows $f : Z \to A$ and $g : Z \to B$, there exists a unique arrow $h : Z \to C$ such that

$$f = \mathrm{pr}_1 \circ h \qquad \text{and} \qquad g = \mathrm{pr}_2 \circ h$$

So do we have products in SML?

So do we have products in SML?
Idea: $\tau * \sigma$ is a product of τ and σ:

$$\begin{aligned} P &= \tau * \sigma \\ \mathsf{pr}_1 &= \texttt{fn (x,y) => x} \\ \mathsf{pr}_2 &= \texttt{fn (x,y) => y} \end{aligned}$$

So do we have products in SML?
Idea: $\tau * \sigma$ is a product of τ and σ:

$$\begin{aligned} P &= \tau * \sigma \\ \mathsf{pr}_1 &= \texttt{fn (x,y) => x} \\ \mathsf{pr}_2 &= \texttt{fn (x,y) => y} \end{aligned}$$

and for any type ρ and any `f : ` $\rho \to \tau$, `g : ` $\rho \to \sigma$,

$$\texttt{h = fn z => (f z, g z)} : \rho \to \tau * \sigma$$

So do we have products in SML?
Idea: $\tau*\sigma$ is a product of τ and σ:

$$\begin{aligned} P &= \tau*\sigma \\ \mathsf{pr}_1 &= \texttt{fn (x,y) => x} \\ \mathsf{pr}_2 &= \texttt{fn (x,y) => y} \end{aligned}$$

and for any type ρ and any `f :` $\rho \to \tau$, `g :` $\rho \to \sigma$,

$$\texttt{h = fn z => (f z, g z)} : \rho \to \tau*\sigma$$

So then...

$$\texttt{f} = \mathsf{pr}_1\texttt{o h} \qquad \text{and} \qquad \texttt{g} = \mathsf{pr}_2\texttt{o h}$$

sectionpage

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

- Group Theory: Groups and group homomorphisms

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

- Group Theory: Groups and group homomorphisms
- Linear Algebra: Vector spaces and linear transformations

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

- Group Theory: Groups and group homomorphisms
- Linear Algebra: Vector spaces and linear transformations
- Topology: Topological spaces and continuous functions between them

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

- Group Theory: Groups and group homomorphisms
- Linear Algebra: Vector spaces and linear transformations
- Topology: Topological spaces and continuous functions between them
- Category Theory: Categories and ...

Mathematics studies *transformations* between objects

Mathematical fields are generally concerned with certain kinds of objects and functions between them:

- Group Theory: Groups and group homomorphisms
- Linear Algebra: Vector spaces and linear transformations
- Topology: Topological spaces and continuous functions between them
- Category Theory: Categories and ...FUNCTORS!

An unfortunate naming collision

In SML, we already have a notion called *functors*, which are things that map between *structures*.

An unfortunate naming collision

In SML, we already have a notion called *functors*, which are things that map between *structures*.

This usage is related, but not the same. We'll use *functor* to mean a kind of "function between categories". In this lecture, we'll focus on "endofunctors": functors from the SML type system to itself.

Defn: An *endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.map :  ('a -> 'b) -> 'a F.t -> 'b F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

```
F.map (g o f) = (F.map g) o (F.map f)
```

Defn: An *endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.map :  ('a -> 'b) -> 'a F.t -> 'b F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

`F.map (g o f)` = `(F.map g) o (F.map f)`

We think of `F.t` as being a *function on types*, the type-level component of `F`. We think of `F.map` as being a *function on functions*, the function component of `F`.

Examples

- Options: `'a F.t = 'a option` and

```
fun map f NONE = NONE | map f (SOME x) = SOME(f x)
```

Examples

- Options: `'a F.t = 'a option` and

```
fun map f NONE = NONE | map f (SOME x) = SOME(f x)
```

- Lists: `'a F.t = 'a list` and

```
fun map f [] = [] | map f (x::xs) = (f x)::map f xs
```

Examples

- Options: `'a F.t = 'a option` and

```
fun map f NONE = NONE | map f (SOME x) = SOME(f x)
```

- Lists: `'a F.t = 'a list` and

```
fun map f [] = [] | map f (x::xs) = (f x)::map f xs
```

- Fixed Products: For some type `t1`, let `'a F.t = 'a * t1` and

```
fun map f (x,z) = (f x,z)
```

Examples

- Options: `'a F.t = 'a option` and

```
fun map f NONE = NONE | map f (SOME x) = SOME(f x)
```

- Lists: `'a F.t = 'a list` and

```
fun map f [] = [] | map f (x::xs) = (f x)::map f xs
```

- Fixed Products: For some type `t1`, let `'a F.t = 'a * t1` and

```
fun map f (x,z) = (f x,z)
```

  Check that:

```
map (g o f) (x,z) = (g(f(x)),z)
                  = map g (f(x),z)
                  = map g (map f (x,z))
```

Examples

- If `F` and `G` are endpfunctors, then `G o F` is an endofunctor with `'a (G o F).t = 'a F.t G.t`
- For some type `t1`, let `'a F.t = t1 -> 'a`, we need

```
F.map :  ('a -> 'b) -> (t1 -> 'a) -> (t1 -> 'b)
```

Examples

- If `F` and `G` are endpfunctors, then `G o F` is an endofunctor with `'a (G o F).t = 'a F.t G.t`
- For some type `t1`, let `'a F.t = t1 -> 'a`, we need

```
F.map :  ('a -> 'b) -> (t1 -> 'a) -> (t1 -> 'b)
```

```
fun map (f:'a -> 'b) (x :  t1 -> 'a) :  t1 -> 'b
      = fn z => f(x(z))
```

Examples

- If `F` and `G` are endpfunctors, then `G o F` is an endofunctor with `'a (G o F).t = 'a F.t G.t`
- For some type `t1`, let `'a F.t = t1 -> 'a`, we need

```
F.map : ('a -> 'b) -> (t1 -> 'a) -> (t1 -> 'b)
```

```
fun map (f:'a -> 'b) (x : t1 -> 'a) : t1 -> 'b
      = fn z => f(x(z))
```

```
val map = curry (op o)
```

Examples

- If `F` and `G` are endpfunctors, then `G o F` is an endofunctor with `'a (G o F).t = 'a F.t G.t`
- For some type `t1`, let `'a F.t = t1 -> 'a`, we need

```
F.map : ('a -> 'b) -> (t1 -> 'a) -> (t1 -> 'b)
```

```
fun map (f:'a -> 'b) (x : t1 -> 'a) : t1 -> 'b
      = fn z => f(x(z))
```

```
val map = curry (op o)
```

- Can we define an endofunctor `F` with `'a F.t = 'a -> t1`?

Examples

- If `F` and `G` are endpfunctors, then `G o F` is an endofunctor with `'a (G o F).t = 'a F.t G.t`
- For some type `t1`, let `'a F.t = t1 -> 'a`, we need

```
F.map : ('a -> 'b) -> (t1 -> 'a) -> (t1 -> 'b)
```

```
fun map (f:'a -> 'b) (x : t1 -> 'a) : t1 -> 'b
      = fn z => f(x(z))
```

```
val map = curry (op o)
```

- Can we define an endofunctor `F` with `'a F.t = 'a -> t1`?
  Answer: No. If we did, we would need `F.map` to be of type

```
F.map : ('a -> 'b) -> ('a -> t1) -> ('b -> t1)
```

Defn: A *contravariant endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.comap :  ('a -> 'b) -> 'b F.t -> 'a F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

```
F.comap (g o f) = (F.comap f) o (F.comap g)
```

Defn: A *contravariant endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.comap :  ('a -> 'b) -> 'b F.t -> 'a F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

```
F.comap (g o f) = (F.comap f) o (F.comap g)
```

Example:

- For some type `t1`, let `'a F.t = 'a -> t1`, we need

```
F.comap :  ('a -> 'b) -> ('b -> t1) -> ('a -> t1)
```

Defn: A *contravariant endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.comap :  ('a -> 'b) -> 'b F.t -> 'a F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

```
F.comap (g o f) = (F.comap f) o (F.comap g)
```

Example:

- For some type `t1`, let `'a F.t = 'a -> t1`, we need

```
F.comap :  ('a -> 'b) -> ('b -> t1) -> ('a -> t1)
```

```
fun comap (f:'a -> 'b) (w :  'b -> t1) :  'a -> t1
      = fn x => w(f(x))
```

Defn: A *contravariant endofunctor* `F` on the SML type system consists of

- A polymorphic type constructor `'a F.t`
- A polymorphic function

```
F.comap :  ('a -> 'b) -> 'b F.t -> 'a F.t
```

such that, for all `f:t1 -> t2` and all `g:t2 -> t3`,

```
F.comap (g o f) = (F.comap f) o (F.comap g)
```

Example:

- For some type `t1`, let `'a F.t = 'a -> t1`, we need

```
F.comap :  ('a -> 'b) -> ('b -> t1) -> ('a -> t1)
```

```
fun comap (f:'a -> 'b) (w :  'b -> t1) :  'a -> t1
     = fn x => w(f(x))
```

Section 5

Natural Transformations

Our progress so far:

Our progress so far:

- Start with objects (types)

Our progress so far:

- Start with objects (types)
- Consider arrows between those objects, yielding categories.

Our progress so far:

- Start with objects (types)
- Consider arrows between those objects, yielding categories.
- Consider arrows between categories: functors

Our progress so far:

- Start with objects (types)
- Consider arrows between those objects, yielding categories.
- Consider arrows between categories: functors
- Consider arrows between functors?

Defn: Given two endofunctors `F` and `G` on the SML type system, a *natural transformation* $\mathtt{E} : \mathtt{F} \to \mathtt{G}$ consists of

- A polymorphic function `E : 'a F.t -> 'a G.t`.

such that for all functions `f : t1 -> t2`,

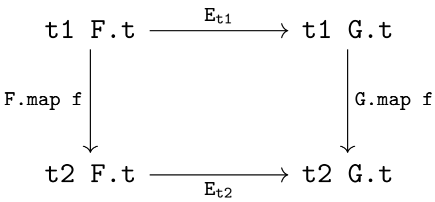
$$\mathtt{E_{t2}\ o\ (F.map\ f)} = \mathtt{(G.map\ f)\ o\ E_{t1}}$$

Defn: Given two endofunctors `F` and `G` on the SML type system, a *natural transformation* `E` : `F` → `G` consists of

- A polymorphic function `E : 'a F.t -> 'a G.t`.

such that for all functions `f : t1 -> t2`,

$$\mathtt{E}_{\mathtt{t2}} \ \mathtt{o} \ (\mathtt{F.map\ f}) = (\mathtt{G.map\ f}) \ \mathtt{o} \ \mathtt{E}_{\mathtt{t1}}$$



We write $\mathtt{E}_{\mathtt{t1}}$ to denote `E`, instantiated at type `t1`, i.e. $\mathtt{E}_{\mathtt{t1}}$ `: t1 F.t -> t1 G.t`.

Examples

- The function `hd: 'a list -> 'a option` is a natural transformation from the `list` endofunctor to the `option` endofunctor

Examples

- The function `hd: 'a list -> 'a option` is a natural transformation from the `list` endofunctor to the `option` endofunctor
- The concat function `concat: 'a list list -> 'a list` is a natural transformation `list o list` $\to$ `list`.

Examples

- The function `hd: 'a list -> 'a option` is a natural transformation from the `list` endofunctor to the `option` endofunctor
- The concat function `concat: 'a list list -> 'a list` is a natural transformation `list o list` $\to$ `list`.
- The function `SOME : 'a -> 'a option` is a natural transformation from the identity functor `Id` (`'a Id.t = 'a` and `Id.map f = f`) to the `option` endofunctor.

Examples

- The function `hd: 'a list -> 'a option` is a natural transformation from the `list` endofunctor to the `option` endofunctor
- The concat function `concat: 'a list list -> 'a list` is a natural transformation `list o list` $\to$ `list`.
- The function `SOME : 'a -> 'a option` is a natural transformation from the identity functor `Id` (`'a Id.t = 'a` and `Id.map f = f`) to the `option` endofunctor.
- For any endofunctor `F`, the identity function `I: 'a F.t -> 'a F.t` given by `I(x)=x` is a natural transformation `F` $\to$ `F`.

The category of endofunctors

We can form a category `Fun(SML,SML)` of endofunctors and natural transformations, where

- The objects are endofunctors on the SML type system
- The arrows are natural transformations

The category of endofunctors

We can form a category `Fun(SML,SML)` of endofunctors and natural transformations, where

- The objects are endofunctors on the SML type system
- The arrows are natural transformations

We can check that the composition of endofunctors defined earlier is associative, that the identity transformation works as an identity arrow, etc.

Monoid in the category of endofunctors

Defn: An endofunctor `T` is said to constitute a *monad* if it comes equipped with two natural transformations

- `eta` : `Id` $\rightarrow$ `T`
- `mu` : `T o T` $\rightarrow$ `T`

Monoid in the category of endofunctors

Defn: An endofunctor `T` is said to constitute a *monad* if it comes equipped with two natural transformations

- $\texttt{eta} : \texttt{Id} \to \texttt{T}$
- $\texttt{mu} : \texttt{T o T} \to \texttt{T}$

such that, for all types τ,

- "`mu` is associative":

$$\texttt{mu}_\tau \texttt{ o (T.map mu}_\tau\texttt{)} = \texttt{mu}_\tau \texttt{ o mu}_{(\tau \texttt{ T.t})}$$

- "`eta` is a unit for `mu`":

$$\texttt{mu}_\tau \texttt{ o eta}_{(\tau \texttt{ T.t})} = \texttt{mu}_\tau \texttt{ o T.map eta}_\tau$$

Example

- `list` is a monad. `eta` is the singleton function

```
fun eta (x : 'a) : 'a list = [x]
```

and `mu` is `concat`:

```
fun mu (L : 'a list list):'a list = foldr (op @) [] L
```

Example

- `list` is a monad. `eta` is the singleton function

```
fun eta (x : 'a) : 'a list = [x]
```

and `mu` is `concat`:

```
fun mu (L : 'a list list):'a list = foldr (op @) [] L
```

The associativity condition says that if we have any `L : t1 list list list`,

$$\texttt{mu (map mu L)} = \texttt{mu (mu L)}$$

and unit says that for all `xs : t1 list`,

Example

- `list` is a monad. `eta` is the singleton function

```
fun eta (x : 'a) : 'a list = [x]
```

  and `mu` is `concat`:

```
fun mu (L : 'a list list):'a list = foldr (op @) [] L
```

  The associativity condition says that if we have any `L : t1 list list list`,

$$\texttt{mu (map mu L)} = \texttt{mu (mu L)}$$

  and unit says that for all `xs : t1 list`,

$$\texttt{mu [xs]} = \texttt{mu (map eta xs)}$$

  which are both true.

Example

- `list` is a monad. `eta` is the singleton function

```
fun eta (x : 'a) : 'a list = [x]
```

  and `mu` is `concat`:

```
fun mu (L : 'a list list):'a list = foldr (op @) [] L
```

  The associativity condition says that if we have any `L : t1 list list list`,

$$\texttt{mu (map mu L)} = \texttt{mu (mu L)}$$

  and unit says that for all `xs : t1 list`,

$$\texttt{mu [xs]} = \texttt{mu (map eta xs)}$$

  which are both true.
- Options are monads

Example

- `list` is a monad. `eta` is the singleton function

```
fun eta (x : 'a) : 'a list = [x]
```

  and `mu` is `concat`:

```
fun mu (L : 'a list list):'a list = foldr (op @) [] L
```

  The associativity condition says that if we have any `L : t1 list list list`,

$$\texttt{mu (map mu L)} = \texttt{mu (mu L)}$$

  and unit says that for all `xs : t1 list`,

$$\texttt{mu [xs]} = \texttt{mu (map eta xs)}$$

  which are both true.
- Options are monads
- The identity functor is a monad

Example

- `list` is a monad. `eta` is the singleton function

  ```
  fun eta (x : 'a) : 'a list = [x]
  ```

  and `mu` is `concat`:

  ```
  fun mu (L : 'a list list):'a list = foldr (op @) [] L
  ```

  The associativity condition says that if we have any `L : t1 list list list`,

  $$\texttt{mu (map mu L)} = \texttt{mu (mu L)}$$

  and unit says that for all `xs : t1 list`,

  $$\texttt{mu [xs]} = \texttt{mu (map eta xs)}$$

  which are both true.
- Options are monads
- The identity functor is a monad
- The functor `'a F.t = ('a -> void) -> void` is a monad

Next Time

Next Time

- Multi-variable functors

Next Time

- Multi-variable functors
- Currying and higher-order functors

Next Time

- Multi-variable functors
- Currying and higher-order functors
- The Yoneda Embedding and Continuation-Passing Style

Next Time

- Multi-variable functors
- Currying and higher-order functors
- The Yoneda Embedding and Continuation-Passing Style
- Other fun stuff?

Thank you!