Effectful composition in natural language semantics

Introducing functors

Dylan Bumford (UCLA)   Simon Charlow (Yale)

NASSLLI 2025 @ UW   June 23, 2025

Notes on the course

Who's this for?

- Linguists interested in what **programming language semantics** (especially, **denotational** approaches, and **pure**, **functional** languages) can teach us about meaning and compositionality in natural language
- Computer scientists interested in **programming languages**, **effects and effect systems**, **category theory**, and **compositionality**, and how these technologies and structures pop up in and inform the study of natural language meaning

We'll try hard to make the material accessible to both audiences, but you'll get the most out of the course if you've encountered compositional linguistic semantics

Resources

Follow along online:

- Slides and course materials (updated as we go): `simoncharlow.com/nasslli`
- A web-based semantic parser: `dylanbumford.com/effects.html`
- Code (Haskell, Lean, Purescript): `github.com/schar/TDParse`

This course is based on the forthcoming Bumford & Charlow (2025)

- Preprint available here: `tinyurl.com/effcomp`

Semantics (natural and unnatural)

Type-theoretic natural language semantics

Meanings are **entities** (type e), **truth values** (type t), or **functions**:

$$\tau ::= e \mid t \mid \underbrace{\tau \to \tau}_{e\to t,\ (e\to t)\to t,\ \ldots}$$

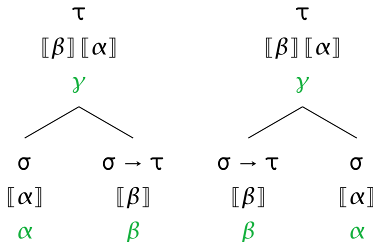
Syntactic combination is interpreted via **functional application**:

Function Application

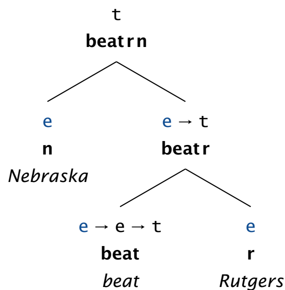
If a node γ has two daughters

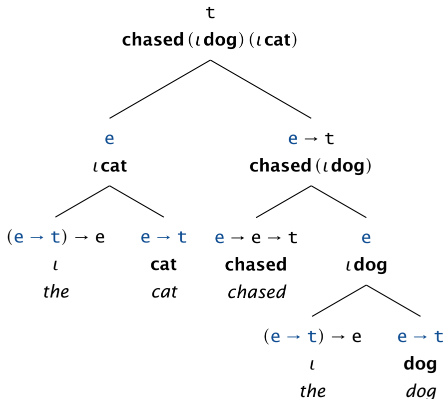
1. $\alpha : \sigma$, and
2. $\beta : \sigma \to \tau$,

then $\gamma : \tau$, and $[\![\gamma]\!] = [\![\beta]\!]\,[\![\alpha]\!]$



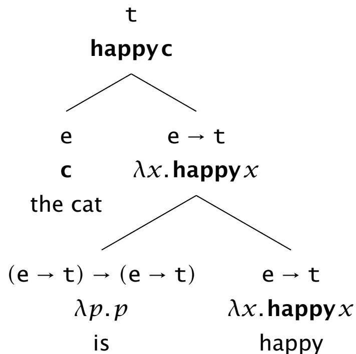
A couple examples

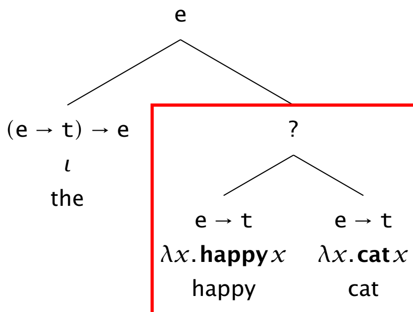




Stress test

New configurations of **already-analyzed** pieces may pose problems:





Type-driven composition

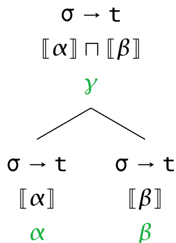
If the discrepancy is systematic enough, it can lead to proposals for additional **modes of combination**, e.g.,

Predicate Modification

If a node γ has two daughters

1. $\alpha : \sigma \to t$, and
2. $\beta : \sigma \to t$,

then $\gamma : \sigma \to t$, and $[\![\gamma]\!] = [\![\beta]\!] \sqcap [\![\alpha]\!]$



This is the standard picture (Klein & Sag 1985, Heim & Kratzer 1998):

- Denotations built from a few basic kinds of objects, and functions over them
- A few basic modes of combination, with composition determined by types

Fleshed out

$(>) \quad : (\sigma \to \tau) \to \sigma \to \tau$

$f > x := f\, x$

Forward Application

$(<) \quad : \sigma \to (\sigma \to \tau) \to \tau$

$x < f := f\, x$

Backward Application

$(\circ) \quad : (\beta \to \varsigma) \to (\alpha \to \beta) \to \alpha \to \varsigma$

$f \circ g := \lambda x. f\, (g\, x)$

Function Composition

$(\sqcap) \quad : (\sigma \to t) \to (\sigma \to t) \to \sigma \to t$

$f \sqcap g := \lambda x. f\, x \wedge g\, x$

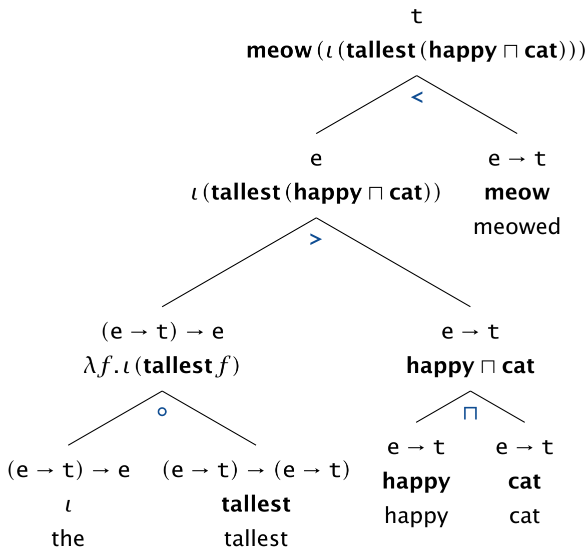
Predicate Modification

$(\upharpoonright) \quad : (\sigma \to \tau \to t) \to (\sigma \to t) \to \sigma \to \tau \to t$

$r \upharpoonright p := \lambda x. \lambda y. p\, x \wedge r\, x\, y$

Relation Restriction

D(e)riving with types



Syntax and semantics in Haskell

```haskell
data Expr = Lit Int
          | Add Expr Expr
          | Mul Expr Expr

exp1 :: Expr
exp1 = Add (Lit 1) (Mul (Lit 2) (Lit 3)) -- 1 + (2 * 3)

exp2 :: Expr
exp2 = Mul (Add (Lit 1) (Lit 2)) (Lit 3) -- (1 + 2) * 3

eval :: Expr -> Int
eval (Lit x)   = x
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)
```

Haskell types and (higher-order) function

```
ghci> :type Add
Add :: Expr -> Expr -> Expr

ghci> :type (+)
(+) :: Int -> Int -> Int
```

This is telling me that the **type** of + is such that it needs one `Int`, and then another, in order to produce an `Int` (and likewise for `Add`).

- So + is a **function**, a recipe for turning inputs to outputs, and it takes its inputs *one at a time*, making it **higher-order**.
- Underlyingly, Haskell is **polymorphic lambda calculus**

Compositional eval-uation

```
eval :: Expr -> Int
eval (Lit x)    = x
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)
```

In a real sense, Add **denotes** $\lambda x.\lambda y.x + y$, which is successively **applied** to its arguments in the course of eval-uation:

```
eval (Add (Lit 1) (Lit 2))
=> (eval (Lit 1)) + (eval (Lit 2))    -- [eval-Add]
=> 1 + (eval (Lit 2))                 -- [eval-Lit]
=> 1 + 2                              -- [eval-Lit]
=> 3                                  -- [+]
```

Note that I evaluated *left-to-right*. This is a reasonable choice, and what Haskell actually does, but the definition of eval is consistent with right-to-left.

Some *disanalogies* between artificial and natural languages

Unlike Haskell syntax, natural language syntax is **ambiguous**:

- I saw the eagle with binoculars.

Even fixing a syntactic structure, ambiguity remains:

- A doctor examined every patient.
- Put a chicken in every pot.

And sometimes disappears:

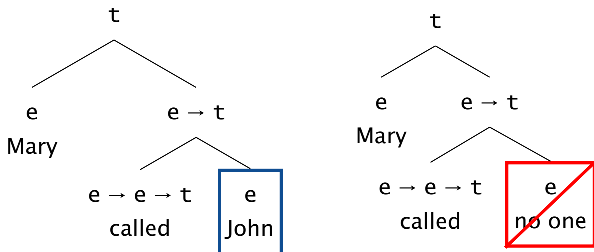
- I know a doctor who examined every patient.
- A doctor examined every patient. She was quite busy.

Effects

More than just an e

This framework is extremely flexible, but some expressions seem to have **too much meaning** to fit into the types it seems they should have

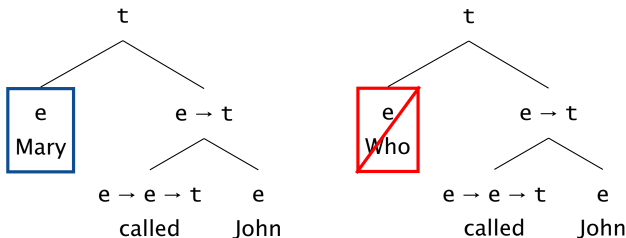
The most famous example of this comes from **quantificational noun phrases**



All reason suggests that ‘no one’ should have type e — it goes everywhere that ‘John’ goes — but there is no x s.t. $[\![\text{no one}]\!] = x$

More than just an e

The same same might be said of **interrogative noun phrases**, like ‘who’



These clearly saturate the same argument positions as ordinary names, but clearly don’t refer to any particular entity

More than just an e

Less obviously, **indefinite noun phrases** like ‘a student’ seem to play the same compositional role as ordinary NPs

But like ‘wh’-words and quantifiers, they too clearly don’t name particular entities. Yet we happily refer back to them *as if* they were names:

- Mary called. She was upset.
- Someone called. She was upset.
- Nobody called. #She was upset.

For that matter, **pronouns** are also very much like entity-like, without having stable referents. **Indexicals** too.

- I just found out they’re delighted about it.

More than just an e

Definite descriptions also seem for all intents and purposes to denote entities . . .

- The person speaking is wearing a navy shirt

. . . except when they don't.

- The only person in this room is wearing yellow
- The King of France is bald

More than just an e

With a bit of **prosodic focus**, any noun phrase can be made to contribute more to what is said than its mere referent.

- I only introduced {Jennifer, JENNIFER} to {Bill, BILL}.
- Who did you introduce Jennifer to?
  I introduced Jennifer (not JENNIFER) to BILL (not Bill).

Or you can supplement the noun phrase with an **apposition**.

- I talked to the dean, a historian.
- Jennifer hasn't read *War & Peace*, which is a classic.

These expressions again seem to compositionally function like simple entities.

Side Effects

All these expressions have an outsized semantics relative to their compositional role, which is just that of an ordinary entity, e.

Today we'll see how to think about these extra bits of meaning as **(side) effects** of the process of interpretation

The inspiration here is from programming language theory

- Pronouns and anaphora
- Questions/'inquisitive' meanings
- Focus
- Presupposition
- Supplemental content
- Quantification and scope

- Variable management
- Nondeterministic computation
- Cellular automata
- Throwing and catching errors
- Tracing
- Control flow (jumps, aborts, loops)

Programs to the semanticist

How should we think about the meanings of expressions that have *effects*?

```
function redPlus(x,y) {
  console.log("red");
  return x + y
}
```

```
function bluePlus(x,y) {
  console.log("blue");
  return x + y
}
```

Both of these programs take two numbers and return their sum. Nevertheless, different things *happen* when they are evaluated:

```
node> redPlus(3,5) % 2 == 0
red
true
```

```
node> bluePlus(3,5) % 2 == 0;
blue
true
```

Impurity and evaluation order

Not all effects are so incidental or innocuous. Variables can cause chaos:

```
var x = 5
function f(y) {
  x = 7;
  return y;
}
```

```
node> x + f(x)
10
```

```
node> f(x) + x
12
```

If you're counting on f and x having some sort of stable function-y and argument-y denotations, such behavior might leave you at a loss

(In)direct style

```
function plusIO(x,y) {
  console.log("working");
  return x() + y()
}
```

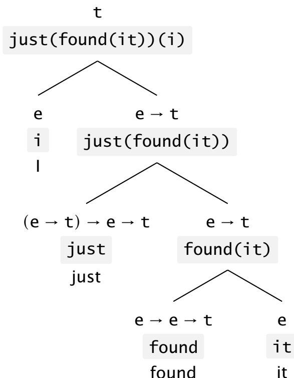
```
plusIO :: IO Int -> IO Int -> IO Int
plusIO mx my =
  mx >>= \x ->
    my >>= \y -> -- ***
      putStrLn "working"
      return (x + y)
```

The indirect (‘monadic’) style explicitly distinguishes **effectful** and **pure** code.

- The direct style just works, because effects are built into the fabric of how the language is evaluated, and hence, implicit.

In another world...

Expr	Type	Denotation
I	e	`() => os.userInfo().username`
just	$(e \to t) \to e \to t$	`p => x => p(x) + " at " + new Date.getMinutes()`
found	$e \to e \to t$	`y => x => x() + " found " + y()`
it	e	`() => readline.question("What exactly? ")`




```
> just(found(it))(i)
What exactly? the pencil
'Simon found the pencil at 11:29'
```

Effects and types

Implicit effects are at odds with the enterprises of linguistic semantics and functional programming, which seek to explicate meaning/program composition in terms of simple, straightforwardly composable mathematical primitives.

In natural language semantics, a guiding principle is that denotations should be **referentially transparent**:

- If you think a certain *behavior* is part of the **meaning** of an expression, then that behavior has to be reflected in its denotation and type

But then how can everything work together in a single grammar?

Functors

Effects and types

In both natural language semantics and functional programming, a guiding principle is that denotations should be **referentially transparent**

One facet of this is that if an expression's denotation isn't (merely) an entity, then its type can't (merely) be e

So a natural place to start is to decide what kinds of objects, and what kinds of types, these special NPs have

Algebraic Data Types

Some of these effects seem to call for denotations with multiple meanings

- Sassy, a cat : $\mathsf{e} \times \mathsf{t}$
  $[\![\text{Sassy, a cat}]\!] = \langle \mathbf{s}, \mathbf{cat}\, \mathbf{s} \rangle$

Other effects seem to call for denotations with multiple variants of meaning

- the cat : $\mathsf{e} + \bot$
  $[\![\text{the cat}]\!] = x$ if $\mathbf{cat} = \{x\}$ else $\#$

Types built from *products*, *sums*, and *functions* are called **Algebraic Data Types**

Some natural choices for effect types

Expression	Type	Denotation
it	$\mathtt{i} \to \mathtt{e}$	$\lambda i.\pi\, i$ (e.g., $\lambda i.i$)
the planet	$\mathtt{e} + \bot$	x if $\mathbf{planet} = \{x\}$ else #
Jupiter, a planet	$\mathtt{e} \times \mathtt{t}$	$\langle \mathbf{j}, \mathbf{planet}\,\mathbf{j} \rangle$
which planet	$\{\mathtt{e}\}$	$\{x \mid \mathbf{planet}\,x\}$
no planet	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	$\lambda Q.\neg\exists x.\mathbf{planet}\,x \wedge Q\,x$
JUPITER	$\mathtt{e} \times \{\mathtt{e}\}$	$\langle \mathbf{j}, \{x \mid x \sim \mathbf{j}\} \rangle$
as for Jupiter	$\mathtt{s} \to (\mathtt{e} \times \mathtt{s})$	$\lambda s.\langle \mathbf{j}, \mathbf{j} \mathbin{+\!\!+} s \rangle$
a planet	$\mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	$\lambda s.\{\langle x, x \mathbin{+\!\!+} s \rangle \mid \mathbf{planet}\,x\}$

It's worth considering what it would take to develop a standard grammar by hand to deal with all these effects. It would be complex, inflexible, brittle, and arbitrary.

Functors

In each case, we have an e situated in some kind of **structural context**

it	$i \to e$
the planet	$e + \bot$
Jupiter, a planet	$e \times t$
which planet	$\{e\}$
no planet	$(e \to t) \to t$
JUPITER	$e \times \{e\}$
as for Jupiter	$s \to e \times s$
a planet	$s \to \{e \times s\}$

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

Functors

In each case, we have an $\texttt{e}$ situated in some kind of **structural context**

it	$\texttt{R}\ \texttt{e} ::= \texttt{i} \to \texttt{e}$
the planet	$\texttt{M}\ \texttt{e} ::= \texttt{e} + \bot$
Jupiter, a planet	$\texttt{W}\ \texttt{e} ::= \texttt{e} \times \texttt{t}$
which planet	$\texttt{S}\ \texttt{e} ::= \{\texttt{e}\}$
no planet	$\texttt{C}\ \texttt{e} ::= (\texttt{e} \to \texttt{t}) \to \texttt{t}$
JUPITER	$\texttt{F}\ \texttt{e} ::= \texttt{e} \times \{\texttt{e}\}$
as for Jupiter	$\texttt{T}\ \texttt{e} ::= \texttt{s} \to \texttt{e} \times \texttt{s}$
a planet	$\texttt{D}\ \texttt{e} ::= \texttt{s} \to \{\texttt{e} \times \texttt{s}\}$

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

Functors

In each case, we have an **e** situated in some kind of **structural context**

it	$R\,\square ::= i \to \square$
the planet	$M\,\square ::= \square + \bot$
Jupiter, a planet	$W\,\square ::= \square \times t$
which planet	$S\,\square ::= \{\square\}$
no planet	$C\,\square ::= (\square \to t) \to t$
JUPITER	$F\,\square ::= \square \times \{\square\}$
as for Jupiter	$T\,\square ::= s \to \square \times s$
a planet	$D\,\square ::= s \to \{\square \times s\}$

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

Functors as **boxes**

Functors are boxes. We can open the box and mess with the contents. Given some $f : \mathsf{e} \to \tau$, we can readily derive apply f "inside" our enriched e's:

- $\{x \mid \textbf{planet}\, x\} \xrightarrow{\bullet f} \{f\, x \mid \textbf{planet}\, x\}$ $\qquad \mathsf{S}\,\tau$

Functors as **boxes**

Functors are boxes. We can open the box and mess with the contents. Given some $f : \mathsf{e} \to \tau$, we can readily derive apply f "inside" our enriched e's:

- $\{x \mid \textbf{planet}\, x\} \xrightarrow{\bullet f} \{f\, x \mid \textbf{planet}\, x\}$ $\quad \mathtt{S}\,\tau$
- $\langle \textbf{j}, \textbf{planet}\, \textbf{j} \rangle \xrightarrow{\bullet f} \langle f\, \textbf{j}, \textbf{planet}\, \textbf{j} \rangle$ $\quad \mathtt{W}\,\tau$

Functors as **boxes**

Functors are boxes. We can open the box and mess with the contents. Given some $f : \mathrm{e} \to \tau$, we can readily derive apply f "inside" our enriched e's:

- $\{x \mid \mathbf{planet}\, x\} \xrightarrow{\bullet f} \{f\, x \mid \mathbf{planet}\, x\}$ $\quad \mathtt{S}\,\tau$
- $\langle \mathbf{j}, \mathbf{planet}\, \mathbf{j}\rangle \xrightarrow{\bullet f} \langle f\, \mathbf{j}, \mathbf{planet}\, \mathbf{j}\rangle$ $\quad \mathtt{W}\,\tau$
- $\langle \mathbf{j}, \{x \mid x \sim \mathbf{j}\}\rangle \xrightarrow{\bullet f} \langle f\, \mathbf{j}, \{f\, x \mid x \sim \mathbf{j}\}\rangle$ $\quad \mathtt{F}\,\tau$
- $\lambda s.\, \{\langle x, s + x\rangle \mid \mathbf{planet}\, x\} \xrightarrow{\bullet f} \lambda s.\, \{\langle f\, x, s + x\rangle \mid \mathbf{planet}\, x\}$ $\quad \mathtt{D}\,\tau$
- x if $\mathbf{planet} = \{x\}$ else $\# \xrightarrow{\bullet f} f\, x$ if $\mathbf{planet} = \{x\}$ else $\#$ $\quad \mathtt{M}\,\tau$
- $\lambda x.\, x \xrightarrow{\bullet f} \lambda x.\, f\, x$ $\quad \mathtt{R}\,\tau$
- $\lambda c.\, \neg\exists x.\, \mathbf{planet}\, x \wedge c\, x \xrightarrow{\bullet f} \lambda c.\, \neg\exists x.\, \mathbf{planet}\, x \wedge c\,(f\, x)$ $\quad \mathtt{C}\,\tau$
- ...

For any functor F, we have

$\bullet : (\sigma \to \tau) \to (F\,\sigma \to F\,\tau)$

Functor examples

A **functor** is a particular sort of **type constructor**, a function from types to types:

$$W\,\alpha ::= \alpha \times t$$

As it happens, many of the particular Functors in our table already have idiosyncratic names (or at least close approximations) in Haskell:

Mathematical type	**Haskell type**
$(e \to t) \to t$	`data Cont t a = Cont ((a -> t) -> t)`
$\{e\}$	`data [] a = [a]`
$s \to \{e \times s\}$	`data State s a = State (s -> (a,s))`
$e + \bot$	`data Maybe a = Just a | Nothing`
$e \times t$	`data Writer t a = Writer a t`
$i \to e$	`data Reader i a = Reader (i -> a)`
$e \times \{e\}$	`--`

Functorial operations

Not every type constructor is a functor. The value(s) of type α hiding in the structure $\mathsf{F}\,\alpha$ must be, intuitively speaking, accessible to other operations.

For instance, say you have a set of numbers, and a function to update those numbers

$$S = \{1, 2, 3\} \qquad f = \lambda n.\, n + 1$$

We can modify the numbers in S by **mapping** f over the contents of S

$$S' = \{f\,n \mid n \in S\} = \{f\,1,\ f\,2,\ f\,3\} = \{2, 3, 4\}$$

Or we can start with a set of strings and update them by adding some text:

$$S = \{\text{"a"}, \text{"b"}, \text{"c"}\} \qquad f = \lambda m.\, m \mathbin{+\!\!+} \text{"d"}$$

$$S' = \{f\,m \mid m \in S\} = \{f\,\text{"a"},\ f\,\text{"b"}, f\,\text{"c"}\} = \{\text{"ad"}, \text{"bd"}, \text{"cd"}\}$$

Functorial operations

Similarly if we have a number paired with a message, we can still easily modify the number by projecting it out and then pairing it back up

$$P = \langle 7, \text{"hello"} \rangle \qquad f = \lambda n.\, n + 1$$

$$P' = \langle f\, P_0, P_1 \rangle = \langle f\, 7, \text{"hello"} \rangle = \langle 8, \text{"hello"} \rangle$$

And again, we would do the same thing no matter what kind of data was stored in P

$$P = \langle \texttt{true}, \text{"hello"} \rangle \qquad f = \lambda b.\, \neg b$$

$$P' = \langle f\, P_0, P_1 \rangle = \langle f\, \texttt{true}, \text{"hello"} \rangle = \langle \texttt{false}, \text{"hello"} \rangle$$

No functorial operations

Not every **structural context** guarantees accessibility like this, e.g.

Mathematical type	Haskell type
$\mathsf{E}\sigma = \sigma \to \sigma$	`data Endo s = Endo (s -> s)`

There's no obvious sense in which $\mathsf{E}\sigma$ is "storing" any σ's in an accessible way

- This would amount to a way of composing a $\sigma \to \tau$ function with a $\sigma \to \sigma$ function to get a $\tau \to \tau$ function
- What's wrong with $f \bullet_{\mathsf{E}} M := \lambda t.t$?

Functor laws

Technically, a type constructor F is a **Functor** if there is some operation that will map a function $k : \sigma \to \tau$ over a structure $F\,\sigma$, yielding an $F\,\tau$

$$\bullet : (\sigma \to \tau) \to F\,\sigma \to F\,\tau$$

Moreover, the function should be 'well-behaved':

Identity:	$\mathbf{id} \bullet M = M$
Composition	$(f \circ g) \bullet M = (f \bullet (g \bullet M))$

- +tri When these laws are satisfied, they guarantee that $\bullet$ doesn't interact with the shape, or structure, that F embodies
- +tri All it does, all it can do, is use f to adjust the value(s) contained inside

Functors in Haskell

A Haskell **functor** is a type constructor with a lawful `fmap`:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
-- 'point-free' versions of the Functor laws:
-- fmap id == id
-- fmap (f . g) == fmap f . fmap g
```

Functor instances

For most Functors, these instances pretty much write themselves

```
data Writer a = Writer (a, Bool)
instance Functor Writer where
--   fmap :: (a -> b) -> Writer a -> Writer b
  fmap k (Writer (a, p)) = Writer (k a, p)
```

$$\mathsf{W}\,\alpha ::= \alpha \times \mathsf{t}$$
$$k \bullet \langle a, b \rangle = \langle k\, a, b \rangle$$

```
data [] a = [a]
instance Functor [] where
-- fmap :: (a -> b) -> [a] -> [b]
  fmap k s = [k x | x <- s]
```

$$\mathsf{S}\,\alpha ::= \{\alpha\}$$
$$k \bullet S = \{k\, a \mid a \in S\}$$

In fact, it is literally impossible to write a well-typed instance of `fmap` that does not satisfy the **Composition** law (Wadler 1989)

(You can still write an instance that violates **Identity**; e.g., `fmap k s = []`)

More Functor instances

```haskell
data Maybe a = Just a | Nothing
instance Functor Maybe where
-- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap k m = case m of
    Nothing -> Nothing
    Just a  -> Just (k a)
```

More Functor instances

```haskell
data Maybe a = Just a | Nothing
instance Functor Maybe where
-- fmap :: (a -> b) -> Maybe a -> Maybe b
  fmap k m = case m of
    Nothing -> Nothing
    Just a  -> Just (k a)
```

```haskell
data Tree a = Leaf a | Branch Label [Tree a]
instance Functor Tree where
-- fmap :: (a -> b) -> Tree a -> Tree b
  fmap k m = case m of
    Leaf a = Leaf (k a)
    Branch l daughters = Branch l [fmap k t | t <- daughters]
```

Bumford, Dylan & Simon Charlow. 2025. Effect-driven interpretation: Functors for natural language composition. Submitted to Cambridge University Press, *Elements in Semantics*.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar*. Oxford: Blackwell.

Klein, Ewan & Ivan A. Sag. 1985. Type-driven translation. *Linguistics and Philosophy* 8(2). 163–201. https://doi.org/10.1007/BF00632365.

Wadler, Philip. 1989. Theorems for free! In *Proceedings of the fourth international conference on functional programming languages and computer architecture*, 347–359.