Monads

Hype for Types

March 16, 2026

The Problem

Partial Functions and Totality

Recall from 15150: a function is *total* if it returns a value for *every* input.

Partial means some inputs have no answer.

Partial Functions and Totality

Recall from 15150: a function is *total* if it returns a value for *every* input.
Partial means some inputs have no answer.

These are partial — and SML lets them lie about it

```
fun head (x :: _) = x    (* type says 'a list -> 'a   *)
                         (* but [] has no answer       *)
fun div  (x, y)   = x div y
                         (* type says int*int -> int   *)
                         (* but y=0 has no answer      *)
```

Partial Functions and Totality

Recall from 15150: a function is *total* if it returns a value for *every* input. *Partial* means some inputs have no answer.

These are partial — and SML lets them lie about it

```
fun head (x :: _) = x    (* type says 'a list -> 'a   *)
                         (* but [] has no answer       *)
fun div  (x, y)   = x div y
                         (* type says int*int -> int   *)
                         (* but y=0 has no answer      *)
```

The type `'a list -> 'a` is a promise it can't keep. `head []` breaks it at runtime with no warning in the type.

Partial Functions and Totality

Recall from 15150: a function is *total* if it returns a value for *every* input. *Partial* means some inputs have no answer.

These are partial — and SML lets them lie about it

```
fun head (x :: _) = x    (* type says 'a list -> 'a   *)
                         (* but [] has no answer       *)
fun div  (x, y)   = x div y
                         (* type says int*int -> int   *)
                         (* but y=0 has no answer      *)
```

The type `'a list -> 'a` is a promise it can't keep. `head []` breaks it at runtime with no warning in the type.

Making partiality honest: `'a option`

```
(* safe_head : 'a list -> 'a option      *)
(* SOME x if non-empty; NONE if empty    *)
```

Partial Functions and Totality

Recall from 15150: a function is *total* if it returns a value for *every* input. *Partial* means some inputs have no answer.

These are partial — and SML lets them lie about it

```
fun head (x :: _) = x    (* type says 'a list -> 'a   *)
                         (* but [] has no answer       *)
fun div  (x, y)   = x div y
                         (* type says int*int -> int   *)
                         (* but y=0 has no answer      *)
```

The type `'a list -> 'a` is a promise it can't keep. `head []` breaks it at runtime with no warning in the type.

Making partiality honest: `'a option`

```
(* safe_head : 'a list -> 'a option      *)
(* SOME x if non-empty; NONE if empty    *)
```

Now the type tells the truth. Making a partial function total means returning `'a option` instead of `'a`.

The Composition Problem

The new problem: making functions total with `option` breaks direct composition.

The Composition Problem

The new problem: making functions total with `option` breaks direct composition.

Two safe, total functions

```
(* safe_tail : 'a list -> 'a list option *)
(* safe_div  : int * int -> int option   *)
```

The Composition Problem

The new problem: making functions total with `option` breaks direct composition.

Two safe, total functions

```
(* safe_tail : 'a list -> 'a list option *)
(* safe_div  : int * int -> int option   *)
```

Each is total. Each is honest.

The Composition Problem

The new problem: making functions total with `option` breaks direct composition.

Two safe, total functions

```
(* safe_tail : 'a list -> 'a list option *)
(* safe_div  : int * int -> int option   *)
```

Each is total. Each is honest.

But you can't plug them together directly

```
safe_tail (safe_tail L)
(* TYPE ERROR: safe_tail expects 'a list
               safe_tail L returns   'a list option
               these are different types!          *)
```

The Composition Problem

The new problem: making functions total with `option` breaks direct composition.

Two safe, total functions

```
(* safe_tail : 'a list -> 'a list option *)
(* safe_div  : int * int -> int option   *)
```

Each is total. Each is honest.

But you can't plug them together directly

```
safe_tail (safe_tail L)
(* TYPE ERROR: safe_tail expects 'a list
               safe_tail L returns   'a list option
               these are different types!          *)
```

The `option` wrapper is the problem. Output type `'a list option` $\neq$ input type `'a list` — you must unwrap every result by hand before passing it on.

The Pyramid of Doom

The only way forward: manually unwrap at every step.

The Pyramid of Doom

The only way forward: manually unwrap at every step.

Drop the first 3 elements, or fail

```
fun tail_3 L =
  case safe_tail L of          (* try step 1     *)
    NONE    => NONE             (* step 1 failed: stop *)
  | SOME L1 =>                  (* step 1 ok: continue *)
    (case safe_tail L1 of       (* try step 2     *)
       NONE    => NONE          (* step 2 failed: stop *)
     | SOME L2 => safe_tail L2) (* step 3     *)
```

The Pyramid of Doom

The only way forward: manually unwrap at every step.

Drop the first 3 elements, or fail

```
fun tail_3 L =
  case safe_tail L of          (* try step 1     *)
    NONE    => NONE             (* step 1 failed: stop *)
  | SOME L1 =>                  (* step 1 ok: continue *)
    (case safe_tail L1 of       (* try step 2     *)
       NONE    => NONE          (* step 2 failed: stop *)
     | SOME L2 => safe_tail L2) (* step 3     *)
```

Three applications. Three levels of nesting.

The Pyramid of Doom

The only way forward: manually unwrap at every step.

Drop the first 3 elements, or fail

```
fun tail_3 L =
  case safe_tail L of          (* try step 1     *)
    NONE    => NONE             (* step 1 failed: stop *)
  | SOME L1 =>                  (* step 1 ok: continue *)
    (case safe_tail L1 of       (* try step 2     *)
       NONE    => NONE          (* step 2 failed: stop *)
     | SOME L2 => safe_tail L2) (* step 3     *)
```

Three applications. Three levels of nesting.

`tail_5` would be *five* levels deep. Every single step is the same dance: check `NONE`, propagate, unwrap `SOME`.

The actual logic — three calls to `safe_tail` — is buried.

The Repeated Pattern

Look at what every single step does:

The Repeated Pattern

Look at what every single step does:

The shape that repeats

```
case <some 'a option> of
  NONE    => NONE              (* failure propagates *)
| SOME x => <f x>              (* success: unwrap and continue
    *)
```

The Repeated Pattern

Look at what every single step does:

The shape that repeats

```
case <some 'a option> of
  NONE    => NONE              (* failure propagates *)
| SOME x => <f x>              (* success: unwrap and continue
    *)
```

`x` is the unwrapped value. `f x` is whatever comes next.

The `NONE` branch is always identical: just pass the failure along.

The Repeated Pattern

Look at what every single step does:

The shape that repeats

```
case <some 'a option> of
  NONE    => NONE              (* failure propagates *)
| SOME x => <f x>              (* success: unwrap and continue
    *)
```

`x` is the unwrapped value. `f x` is whatever comes next.
The `NONE` branch is always identical: just pass the failure along.

The insight

This shape appears at *every step* of every option chain.

The Repeated Pattern

Look at what every single step does:

The shape that repeats

```
case <some 'a option> of
  NONE    => NONE              (* failure propagates *)
| SOME x => <f x>              (* success: unwrap and continue
    *)
```

`x` is the unwrapped value. `f x` is whatever comes next.
The `NONE` branch is always identical: just pass the failure along.

The insight

This shape appears at *every step* of every option chain.
It doesn't depend on what `f` is. It doesn't depend on the types.

The Repeated Pattern

Look at what every single step does:

The shape that repeats

```
case <some 'a option> of
  NONE    => NONE              (* failure propagates *)
| SOME x => <f x>              (* success: unwrap and continue
    *)
```

`x` is the unwrapped value. `f x` is whatever comes next.
The `NONE` branch is always identical: just pass the failure along.

The insight

This shape appears at *every step* of every option chain.
It doesn't depend on what `f` is. It doesn't depend on the types.
When you see the same pattern everywhere, you **name it and abstract it**.
That abstraction is `bind`.

bind: Sequencing Effects

`bind`: The Abstraction

Extract the pattern into a function:

bind: The Abstraction

Extract the pattern into a function:

```
(* bind : 'a option -> ('a -> 'b option) -> 'b option *)
fun bind opt f =
  case opt of
    NONE   => NONE      (* failure: skip f entirely      *)
  | SOME x => f x       (* success: unwrap x, call f     *)
```

bind: The Abstraction

Extract the pattern into a function:

```
(* bind : 'a option -> ('a -> 'b option) -> 'b option *)
fun bind opt f =
  case opt of
    NONE   => NONE     (* failure: skip f entirely      *)
  | SOME x => f x      (* success: unwrap x, call f     *)
```

opt is the result of the previous step.
f is the next step — a function that takes a plain value and returns the next option.

bind: The Abstraction

Extract the pattern into a function:

```
(* bind : 'a option -> ('a -> 'b option) -> 'b option *)
fun bind opt f =
  case opt of
    NONE   => NONE     (* failure: skip f entirely      *)
  | SOME x => f x      (* success: unwrap x, call f     *)
```

opt is the result of the previous step.
f is the next step — a function that takes a plain value and returns the next option.
bind handles the case-split so you never have to write it again.

bind: The Abstraction

Extract the pattern into a function:

```
(* bind : 'a option -> ('a -> 'b option) -> 'b option *)
fun bind opt f =
  case opt of
    NONE   => NONE     (* failure: skip f entirely      *)
  | SOME x => f x      (* success: unwrap x, call f     *)
```

opt is the result of the previous step.
f is the next step — a function that takes a plain value and returns the next option.
bind handles the case-split so you never have to write it again.

What bind means in English

"If the previous step produced a value, pass it to the next step.
If it failed, stop — don't bother calling the next step at all."

A Connection You Already Know: CPS

From 15150: continuation-passing style (CPS) passes "what to do next" as an explicit argument instead of returning a value.

A Connection You Already Know: CPS

From 15150: continuation-passing style (CPS) passes "what to do next" as an explicit argument instead of returning a value.

bind is CPS for option

```
(* In direct style: *)
fun safe_head (x :: _) = SOME x
  | safe_head []       = NONE

(* bind opt f  says: "opt, and then do f" *)
(* f is the continuation --- what to do next *)
bind (safe_head L) (fn x =>    (* if head succeeds... *)
bind (safe_tail L) (fn xs =>   (* ...and tail succeeds *)
SOME (x, xs)))                 (* ...produce the pair  *)
```

A Connection You Already Know: CPS

From 15150: continuation-passing style (CPS) passes "what to do next" as an explicit argument instead of returning a value.

bind is CPS for option

```
(* In direct style: *)
fun safe_head (x :: _) = SOME x
  | safe_head []       = NONE

(* bind opt f  says: "opt, and then do f" *)
(* f is the continuation --- what to do next *)
bind (safe_head L) (fn x =>    (* if head succeeds... *)
bind (safe_tail L) (fn xs =>   (* ...and tail succeeds *)
SOME (x, xs)))                 (* ...produce the pair  *)
```

`fn x => ...` is exactly the continuation from CPS.

`bind` threads the value through, or short-circuits on `NONE`.

A Connection You Already Know: CPS

From 15150: continuation-passing style (CPS) passes "what to do next" as an explicit argument instead of returning a value.

bind is CPS for option

```
(* In direct style: *)
fun safe_head (x :: _) = SOME x
  | safe_head []       = NONE

(* bind opt f  says: "opt, and then do f" *)
(* f is the continuation --- what to do next *)
bind (safe_head L) (fn x =>    (* if head succeeds... *)
bind (safe_tail L) (fn xs =>   (* ...and tail succeeds *)
SOME (x, xs)))                 (* ...produce the pair  *)
```

`fn x => ...` is exactly the continuation from CPS.
`bind` threads the value through, or short-circuits on `NONE`.

The difference from plain CPS: in CPS, the continuation always runs. Here, `NONE` *cancels* the continuation. The effect is baked in.

The >>= Notation

Writing `bind opt f` for every step is noisy. We declare it as an infix operator:

The >>= Notation

Writing `bind opt f` for every step is noisy. We declare it as an infix operator:

Declaring >>= in SML

```
infix >>=
fun op >>= (opt, f) = bind opt f
(* Now we can write: opt >>= f
   instead of: bind opt f           *)
```

The >>= Notation

Writing `bind opt f` for every step is noisy. We declare it as an infix operator:

Declaring >>= in SML

```
infix >>=
fun op >>= (opt, f) = bind opt f
(* Now we can write: opt >>= f
   instead of: bind opt f           *)
```

>>= is called **"bind"** or **"then"**.

The arrow points right: the value on the left flows into the function on the right.

The >>= Notation

Writing `bind opt f` for every step is noisy. We declare it as an infix operator:

Declaring >>= in SML

```
infix >>=
fun op >>= (opt, f) = bind opt f
(* Now we can write: opt >>= f
   instead of: bind opt f          *)
```

>>= is called **"bind"** or **"then"**.
The arrow points right: the value on the left flows into the function on the right.

Why this notation?

`opt >>= fn x => next`: "take opt; if it succeeded, name the result `x` and run `next`."
Chains left to right — matching the order you'd write the steps in English.

The Pyramid, Rewritten

Before and after

```
(* BEFORE: manual case analysis at every step *)
fun tail_3 L =
  case safe_tail L of
    NONE    => NONE
  | SOME L1 => case safe_tail L1 of
                 NONE    => NONE
               | SOME L2 => safe_tail L2

(* AFTER: bind handles the plumbing *)
fun tail_3 L =
  safe_tail L  >>= fn L1 =>   (* and then... *)
  safe_tail L1 >>= fn L2 =>   (* and then... *)
  safe_tail L2                (* done        *)
```

The Pyramid, Rewritten

Before and after

```
(* BEFORE: manual case analysis at every step *)
fun tail_3 L =
  case safe_tail L of
    NONE    => NONE
  | SOME L1 => case safe_tail L1 of
                 NONE    => NONE
               | SOME L2 => safe_tail L2

(* AFTER: bind handles the plumbing *)
fun tail_3 L =
  safe_tail L  >>= fn L1 =>   (* and then... *)
  safe_tail L1 >>= fn L2 =>   (* and then... *)
  safe_tail L2                (* done        *)
```

The logic is three calls to `safe_tail`. Now that's exactly what you see — nothing else.

The Pyramid, Rewritten

Before and after

```
(* BEFORE: manual case analysis at every step *)
fun tail_3 L =
  case safe_tail L of
    NONE    => NONE
  | SOME L1 => case safe_tail L1 of
                 NONE    => NONE
               | SOME L2 => safe_tail L2

(* AFTER: bind handles the plumbing *)
fun tail_3 L =
  safe_tail L  >>= fn L1 =>   (* and then... *)
  safe_tail L1 >>= fn L2 =>   (* and then... *)
  safe_tail L2                (* done        *)
```

The logic is three calls to `safe_tail`. Now that's exactly what you see — nothing else.

`tail_5`? Two more lines, no more nesting.

Monads

What Is a Monad?

The observation: `bind` and `SOME` made `option` composable. Nothing about this argument was specific to `option`.

What Is a Monad?

The observation: `bind` and `SOME` made `option` composable. Nothing about this argument was specific to `option`.

Definition

A *monad* is a type `'a t` with two operations:

- `return : 'a -> 'a t` — lift a plain value; add no effect
- `bind : 'a t -> ('a -> 'b t) -> 'b t` — "and then": run the first, pass its result to the second

subject to three laws (next slide).

What Is a Monad?

The observation: `bind` and `SOME` made `option` composable. Nothing about this argument was specific to `option`.

Definition

A *monad* is a type `'a t` with two operations:

- `return : 'a -> 'a t` — lift a plain value; add no effect
- `bind : 'a t -> ('a -> 'b t) -> 'b t` — "and then": run the first, pass its result to the second

subject to three laws (next slide).

Any type that provides `return` and `bind` satisfying the laws is a monad. `option` is one. `result` is another. There are many more.

Option as a Monad

Concretely: here is how `option` fits the definition.

Option as a Monad

Concretely: here is how `option` fits the definition.

Option satisfies the monad interface

```
type 'a t = 'a option

fun return x = SOME x          (* wrap value; no failure *)

fun bind NONE     _ = NONE     (* no value: skip f   *)
  | bind (SOME x) f = f x      (* have value: call f *)
```

Option as a Monad

Concretely: here is how `option` fits the definition.

Option satisfies the monad interface

```
type 'a t = 'a option

fun return x = SOME x          (* wrap value; no failure *)

fun bind NONE     _ = NONE     (* no value: skip f   *)
  | bind (SOME x) f = f x      (* have value: call f *)
```

`return` is SOME. `bind` is the pattern we extracted earlier.
The option monad sequences computations that might fail.

The Monad Laws

Not every `bind`/`return` **pair is a monad.** The laws ensure they behave consistently.

The Monad Laws

Not every `bind`/`return` pair is a monad. The laws ensure they behave consistently.

Left identity: `return x >>= f = f x`

Wrapping `x` and immediately binding does nothing extra.
`return` introduces no effect, so binding it is the same as calling `f` directly.

The Monad Laws

Not every `bind`/`return` pair is a monad. The laws ensure they behave consistently.

Left identity: `return x >>= f = f x`

Wrapping `x` and immediately binding does nothing extra.
`return` introduces no effect, so binding it is the same as calling `f` directly.

Right identity: `m >>= return = m`

Binding and immediately wrapping does nothing extra.
`return` at the end of a chain is a no-op.

The Monad Laws

Not every `bind`/`return` pair is a monad. The laws ensure they behave consistently.

Left identity: `return x >>= f = f x`

Wrapping `x` and immediately binding does nothing extra.
`return` introduces no effect, so binding it is the same as calling `f` directly.

Right identity: `m >>= return = m`

Binding and immediately wrapping does nothing extra.
`return` at the end of a chain is a no-op.

Associativity: `(m >>= f) >>= g = m >>= (fn x => f x >>= g)`

Chaining three steps is independent of how you parenthesize them.
This is what makes >>= feel like sequential "and then" — steps can be regrouped freely.

The Monad Laws

Not every `bind`/`return` pair is a monad. The laws ensure they behave consistently.

Left identity: `return x >>= f = f x`

Wrapping `x` and immediately binding does nothing extra.
`return` introduces no effect, so binding it is the same as calling `f` directly.

Right identity: `m >>= return = m`

Binding and immediately wrapping does nothing extra.
`return` at the end of a chain is a no-op.

Associativity: `(m >>= f) >>= g = m >>= (fn x => f x >>= g)`

Chaining three steps is independent of how you parenthesize them.
This is what makes >>= feel like sequential "and then" — steps can be regrouped freely.

The laws are what make monadic code *refactorable*. Without them you couldn't safely regroup a chain of >>= steps.

More Monads

Result: Failure With a Reason

The problem: `option` is silent on failure. `NONE` tells you nothing — not which step failed, not why.

Result: Failure With a Reason

The problem: `option` is silent on failure. `NONE` tells you nothing — not which step failed, not why.

Add a message to the failure case

```
datatype 'a result = Ok of 'a | Err of string
(* success: Ok x    failure: Err "why it failed" *)

fun safe_div (_, 0) = Err "division by zero"
  | safe_div (x, y) = Ok (x div y)

fun bind (Err e) _ = Err e   (* skip f, pass error on *)
  | bind (Ok x)  f = f x     (* unwrap x, call f      *)
```

Result: Failure With a Reason

The problem: `option` is silent on failure. `NONE` tells you nothing — not which step failed, not why.

Add a message to the failure case

```
datatype 'a result = Ok of 'a | Err of string
(* success: Ok x    failure: Err "why it failed" *)

fun safe_div (_, 0) = Err "division by zero"
  | safe_div (x, y) = Ok (x div y)

fun bind (Err e) _ = Err e   (* skip f, pass error on *)
  | bind (Ok x)  f = f x     (* unwrap x, call f      *)
```

`Err e` plays the role of `NONE` — it short-circuits. The difference: the reason travels with the failure instead of being lost.

Result: Failure With a Reason

The problem: `option` is silent on failure. `NONE` tells you nothing — not which step failed, not why.

Add a message to the failure case

```
datatype 'a result = Ok of 'a | Err of string
(* success: Ok x    failure: Err "why it failed" *)

fun safe_div (_, 0) = Err "division by zero"
  | safe_div (x, y) = Ok (x div y)

fun bind (Err e) _ = Err e   (* skip f, pass error on *)
  | bind (Ok x)  f = f x     (* unwrap x, call f      *)
```

`Err e` plays the role of `NONE` — it short-circuits. The difference: the reason travels with the failure instead of being lost.

```
safe_div (10, 2) >>= fn x => safe_div (x, 0)
(* = Err "division by zero"  -- message intact *)
```

The Option–Result Comparison

Option and Result both solve the same problem. The only difference is how much information failure carries.

The Option–Result Comparison

Option and Result both solve the same problem. The only difference is how much information failure carries.

	Option	**Result**
Success	`SOME x`	`Ok x`
Failure	`NONE`	`Err "why"`
bind success	unwrap, call `f`	unwrap, call `f`
bind failure	propagate `NONE`	propagate `Err e`
Use when	failure is obvious	failure needs explanation

The Option–Result Comparison

Option and Result both solve the same problem. The only difference is how much information failure carries.

	Option	**Result**
Success	`SOME x`	`Ok x`
Failure	`NONE`	`Err "why"`
bind success	unwrap, call `f`	unwrap, call `f`
bind failure	propagate `NONE`	propagate `Err e`
Use when	failure is obvious	failure needs explanation

Same monad, different payload

The `bind` logic is structurally identical in both. Monads give you a uniform interface regardless of what failure looks like.

do-notation: >>= in Disguise

The problem: long >>= chains with many `fn` bindings get visually noisy.

do-notation: >>= in Disguise

The problem: long >>= chains with many `fn` bindings get visually noisy.

In OCaml, `let%bind` desugars to >>=

```
(* Using >>= explicitly *)
safe_tail l  >>= fun l1 =>
safe_tail l1 >>= fun l2 =>
safe_tail l2
```

do-notation: >>= in Disguise

The problem: long >>= chains with many `fn` bindings get visually noisy.

In OCaml, `let%bind` desugars to >>=

```
(* Using >>= explicitly *)
safe_tail l  >>= fun l1 =>
safe_tail l1 >>= fun l2 =>
safe_tail l2
```

```
(* The same program with let%bind *)
let%bind l1 = safe_tail l  in
let%bind l2 = safe_tail l1 in
safe_tail l2
```

do-notation: >>= in Disguise

The problem: long >>= chains with many `fn` bindings get visually noisy.

In OCaml, `let%bind` desugars to >>=

```
(* Using >>= explicitly *)
safe_tail l  >>= fun l1 =>
safe_tail l1 >>= fun l2 =>
safe_tail l2
```

```
(* The same program with let%bind *)
let%bind l1 = safe_tail l  in
let%bind l2 = safe_tail l1 in
safe_tail l2
```

`let%bind x = m in rest` means exactly `m >>= fn x => rest`. Notation only — the monad is identical.

do-notation: >>= in Disguise

The problem: long >>= chains with many `fn` bindings get visually noisy.

In OCaml, `let%bind` desugars to >>=

```
(* Using >>= explicitly *)
safe_tail l  >>= fun l1 =>
safe_tail l1 >>= fun l2 =>
safe_tail l2
```

```
(* The same program with let%bind *)
let%bind l1 = safe_tail l  in
let%bind l2 = safe_tail l1 in
safe_tail l2
```

`let%bind x = m in rest` means exactly `m >>= fn x => rest`. Notation only — the monad is identical.

The key point

do-notation works for *any* monad — option, result, state, IO, async.
It is not special I/O syntax. It is >>= in disguise.

Putting It Together

The Monad Zoo

One interface — `bind` and `return` — six different notions of "and then".

The Monad Zoo

One interface — `bind` and `return` — six different notions of "and then".

Monad	**What "and then" means**	**Seen in**
Option	only if the last step succeeded	SML `option`
Result	only if succeeded, else carry error	OCaml `Or_error`
State	pass hidden state to the next step	Compilers, parsers
List	run next step for every result	Search, nondeterminism
IO	execute this effect, then continue	Haskell `IO`
Async	wait for this, then continue	OCaml `Async`

The Monad Zoo

One interface — `bind` and `return` — six different notions of "and then".

Monad	What "and then" means	Seen in
Option	only if the last step succeeded	SML `option`
Result	only if succeeded, else carry error	OCaml `Or_error`
State	pass hidden state to the next step	Compilers, parsers
List	run next step for every result	Search, nondeterminism
IO	execute this effect, then continue	Haskell `IO`
Async	wait for this, then continue	OCaml `Async`

The payoff

A monad is a pattern for **sequencing computations that carry an effect**. The effect changes. The interface — `bind` and `return` — stays the same.

What's Next?

Monads are the step where "this computation might fail" stops being a comment you write and starts being something the type system tracks *and composes for you*.

What's Next?

Monads are the step where "this computation might fail" stops being a comment you write and starts being something the type system tracks *and composes for you*.

Vote for next week

Option A: Monads, Part 2

State monad in depth. Parser combinators. Monad transformers: stacking effects.

Option B: Parametric Polymorphism

What can a function with type `('a -> 'b) -> 'a list -> 'b list` *possibly* do?
Free theorems. What types tell you for free.