type inference

the act of determining the type of expressions in a program, statically and without annotations
Type-checking allows the compiler to check for the validity of a program before it is executed.

But even stronger: when it comes to ad hoc polymorphism, type-checking determines what code is executed.
Recall: in Haskell, a single function identifier can refer to different implementations, depending on the typeclass.

\[
x \;==\; y \\
f \;x \;>>=\; \lambda y \rightarrow g \;y
\]
Consider method overloading in Java

class Point {
    void move(int dx, int dy) {
        // move implementation
    }
    int move(float dx, float dy) {
        // move implementation
    }
    String move() {
        // move implementation
    }
}

Point p = Point();
p.move(1, 2);
p.move(1.5, 3.5);
p.move();
In Haskell, we’re actually a bit more constrained:

(==) :: Eq a => a -> a -> Bool

1 == 2
1.5 == 2.5
"Hello" == "Goodbye"
But now consider `read`

```haskell
read :: Read a => String -> a
```

`read` is ad-hoc polymorphic in its return type! This is known as *return type polymorphism*.
This is not allowed in Java!

class Point {
    void move(int dx, int dy) {
        ... }
    int move(int dx, int dy) {
        ... }
    String move(int dx, int dy) {
        ... }
}
In Java, the compiler chooses which implementation of `move` to use based on its arguments, but not its return type. This makes a lot of sense if we regularly call a function for its side effects:

```
p.move(1, 2);
```
In Haskell, this is much rarer, so **type inference** can (mostly) be used to determine the return type.

```haskell
read :: Read a => String -> a
```

True && (read "False")
3 + (read "6")
4.5 - (read "2.3")
In Haskell, this is much rarer, so type inference can (mostly) be used to determine the return type.

return :: Monad m => a -> m a
In Haskell, this is much rarer, so **type inference** can (mostly) be used to determine the return type.

\[
\text{return} :: \text{Monad } a \Rightarrow a \to m a
\]

\[
f :: \text{Maybe } \text{Int} \to \text{Either } \text{String } \text{Int} \to \text{IO } \text{Int} \to ___
f = \ldots
\]

\[
f \text{(return 1)} \text{(return 1)} \text{(return 1)}
\]
But sometimes a concrete type cannot be inferred when the function is called.

read "3"

f :: String -> String
f x = show (read x)
We fix this by providing explicit type annotations.

```haskell
read "3" :: Int

f :: String -> String
f x = show (read x :: Int)
```
Combining monads, monad transformers
We've seen two monads representing different kinds of effects: **Maybe** (failing computations) and **State** (stateful computations).

How do we express a computation that has a *combination* of effects?
Goal: label each node with its position in the tree’s postorder traversal, but fail if see “David”.

\[ \text{postOrderLabelM :: BTree String} \rightarrow \text{State Int (Maybe (BTree Int))} \]
Three implementations:

1. Manually expanding Maybes.
2. Writing a new "State + Maybe" monad.
3. Using *monad transformers*. 
Haskell uses monad transformers to represent *combinations of effects*. Is this the best approach?

Designing and implementing **effect systems** is an active area of research in programming languages!