Deriving constraints from function genericity

\[ f :: a \rightarrow a \]

\[ f \ x = \ldots \]
f :: a -> [a]
f x = ...
\( f :: a \rightarrow b \)
f \( x = \ldots \)
(not) Deriving constraints from function genericity

\[ \langle T \rangle \quad T \ f(T \ x) \ \{ \ \\
\quad \ldots \\
\} \]
(not) Deriving constraints from function genericity

\[
<T> \ T \ f(T \ x) \ \{ \\
    \text{return} \ x; \\
\}
\]
Deriving constraints from function genericity

```cpp
<T> T f(T x) {
    blowUpUofT();
    return x;
}
```
But what about (+)?

(+) :: Int -> Int -> Int

(+): a -> a -> a
ad hoc polymorphism

the ability of an entity to behave differently on different “input” or “contained” types
**typeclass**

a set of types defined by an *interface* (set of functions) the type must implement
Common typeclasses and typeclass constraints
Unlike types, typeclasses are open for extension
e.g. instance Eq Shape where
Higher-order typeclasses
But what about \texttt{map}?

\texttt{list-map} ::
\[
(a \to b) \to [a] \to [b]
\]

\texttt{stream-map} ::
\[
(a \to b) \to \text{Stream } a \to \text{Stream } b
\]

\texttt{vector-map} ::
\[
(a \to b) \to \text{Vector } a \to \text{Vector } b
\]
class Functor f where
    fmap :: (a -> b) -> f a -> f b
Dealing with failing computations
NoMethodError in Rms::Properties#index

Showing /home/saona/ruby/indieassets/app/views/rms/properties/index.html.erb where line #29 raised:

undefined method `name' for nil:NilClass

Extracted source (around line #29):

```ruby
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

  <td>#{property.seller_phone %}</td>

  <td>#{property.category.name %}</td>

  <td>#{property.property_name %}</td>

  <td>#{property.property_details %}</td>

  <td>#{property.location %}</td>
```
C COMPILER

Y U NO TELL ME WHERE THE SEGMENTATION FAULT IS
data Maybe a = Nothing | Just a
Chaining failing computations
s = null;
if (x != null) {
    y = x.f();
    if (y != null) {
        z = y.g();
        if (z != null) {
            for (a in z) {
                if (a != null) {
                    s = update(s, a);
                }
            }
        }
    }
}
if (s != null) { return s.h(); }
case x of
    Nothing -> Nothing
    Just x' ->
        case f x' of
            Nothing -> Nothing
            Just y ->
                case g y of
                    Nothing -> Nothing
                    Just z ->
                        foldl ___ Nothing z
case x of
  Nothing -> Nothing
  Just x' ->
    case f x' of
      Nothing -> Nothing
      Just y ->
        case g y of
          Nothing -> Nothing
          Just z ->
            foldl @&^ Nothing z
Elvis operator ?: 
  x ?: y

Safe navigation operator ? . 
  x? . y
Back to Haskell

couldFail :: _ -> Maybe _

Given \( x = \text{couldFail} \ y \), what now?
“If null then null, else do something”

absMaybe :: Maybe Int -> Maybe Int
absMaybe Nothing = Nothing
absMaybe (Just x) = Just (abs x)

lengthMaybe :: Maybe [a] -> Maybe Int
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
"If null then null, else do something"

absMaybe :: Maybe Int -> Maybe Int
absMaybe Nothing = Nothing
absMaybe (Just x) = Just (abs x)

lengthMaybe :: Maybe [a] -> Maybe Int
lengthMaybe Nothing = Nothing
lengthMaybe (Just xs) = Just (length xs)
try :: (a -> b) -> Maybe a -> Maybe b
try _ Nothing = Nothing
try f (Just x) = Just (f x)
try :: (a -> b) -> Maybe a -> Maybe b
try _ Nothing = Nothing
try f (Just x) = Just (f x)
Maybe is a functor

\[ \text{fmap} :: (a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b \]
Either String is also a functor!!

\[ \text{fmap} :: (a \to b) \to \text{Either String } a \to \text{Either String } b \]
“If null then null, else do something *that might fail*”

\[
\text{recipMaybe} :: \text{Maybe Float} \rightarrow \text{Maybe Float} \\
\text{recipMaybe} \text{ Nothing} = \text{Nothing} \\
\text{recipMaybe} \text{ (Just } x) = \begin{cases} \\
\text{Nothing} & \text{if } x = 0 \\
\text{Just } (1 / x) & \text{else} \\
\end{cases}
\]

\[
\text{headMaybe} :: \text{Maybe } [a] \rightarrow \text{Maybe } a \\
\text{headMaybe} \text{ Nothing} = \text{Nothing} \\
\text{headMaybe} \text{ (Just } xs) = \begin{cases} \\
\text{Nothing} & \text{if } \text{null } xs \\
\text{Just } (\text{head } xs) & \text{else} \\
\end{cases}
\]
“If null then null, else do something that might fail”

recipMaybe :: Maybe Float -> Maybe Float
recipMaybe Nothing = Nothing
recipMaybe (Just x) = if x == 0
                       then Nothing
                       else Just (1 / x)

headMaybe :: Maybe [a] -> Maybe a
headMaybe Nothing = Nothing
headMaybe (Just xs) = if null xs
                      then Nothing
                      Just (head xs)
tryFail :: (a -> Maybe b) -> Maybe a -> Maybe b
tryFail _ Nothing = Nothing
tryFail f (Just x) = f x
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing `andThen` _ = Nothing
(Just x) `andThen` f = f x
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
Nothing `andThen` _ = Nothing
(Just x) `andThen` f = f x
`andThen` :: `Maybe a -> (a -> Maybe b) -> Maybe b`

Nothing `andThen` _ = Nothing
(Just x) `andThen` f = f x
x `andThen` f `andThen` g `andThen` h
Maybe a -> (a -> Maybe b) -> Maybe b
(⋙⋙=) :: m a -> (a -> m b) -> m b
class Monad m where
    (>>>=) :: m a -> (a -> m b) -> m b
Either String is an instance of Monad!

(>>>=) :: Either String a
     -> (a -> Either String b)
     -> Either String b