Monoid, Foldable, Traversable

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

Disclaimer

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Máquina de Estado

- Considere o seguinte problema: tenho uma árvore do tipo `Tree Char` e quero converter para uma `Tree Int` sendo que os nós folhas receberão números de `[0..]` na sequência de visita:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
                deriving Show

tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')

f tree = Node (Node (Leaf 0) (Leaf 1)) (Leaf 2)
```

Um esqueleto dessa função seria:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

rlabel :: Tree a -> Tree Int
rlabel (Leaf _)   = Leaf n
rlabel (Node l r) = Node (rlabel l) (rlabel r)
```

Estado

- Queremos que `n` seja uma variável de estado, ou seja, toda vez que a utilizarmos ela altere seu estado!
- Mas somos puros e imutáveis! Como podemos resolver isso?

UFABC

- Uma ideia é incorporar o estado atual na declaração da função:

```haskell
rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n + 1)
rlabel (Node l r) = (Node l' r', n'')
  where
    (l', n')   = rlabel l n  -- altera o estado de n
    (l'', n'') = rlabel r n' -- altera o estado de n'
```

Com isso podemos chamar:

```
> rlabel tree 0
 => (Node l' r', n'')

> (l', n') = rlabel (Node (Leaf 'a') (Leaf 'b')) 0
 => (Node l' r', n'')

> (l', n') = rlabel (Leaf 'a') 0
  => (Leaf 0, 1)
> (r', n'') = rlabel (Leaf 'b') 1
 => (Leaf 1, 2)

> (r, n'') = rlabel (Leaf 'c') 2
 => (Leaf 2, 3)

(Node (Node (Leaf 0) (Leaf 1)) (Leaf 2), 3)
```

Estado

- Vamos tentar generalizar esse padrão de programação criando um tipo estado:

```haskell
type State = Int
```

- O tipo State pode ser definido como qualquer tipo que represente o estado que queremos trabalhar.

UFABC

- Com isso queremos criar uma função que recebe um estado e retorna um novo estado, vamos chamar de transformador de estado ou *state transformer*:

```haskell
type ST = State -> State
```

- Mas como vimos no exemplo anterior, pode ser útil que além de devolver um estado novo, o transformador de estado pode retornar um valor para utilizarmos. No caso de **rlabel**:

```
ST = State -> (Tree Int, State)
```

Então podemos redefinir ST como:

```
type ST a = State -> (a, State)
```

Com isso a assinatura de `rlabel` pode se tornar:

```
rlabel :: Tree a -> ST (Tree Int)
```

- Um transformador de estado pode ser visto como uma caixa que recebe um estado e retorna um valor e um novo estado:

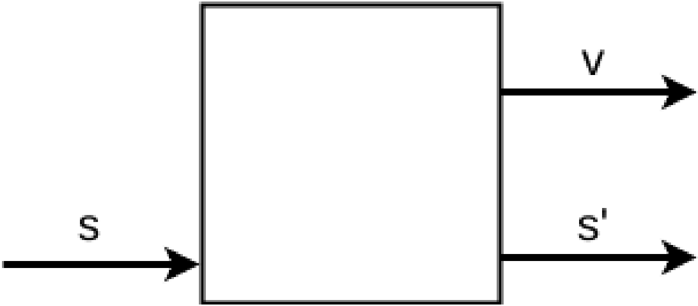

Figura 1: Transformador de estado

- Podemos pensar também em um transformador de estados que, além de um estado, recebe um valor para agir dentro do ambiente que ele vive:

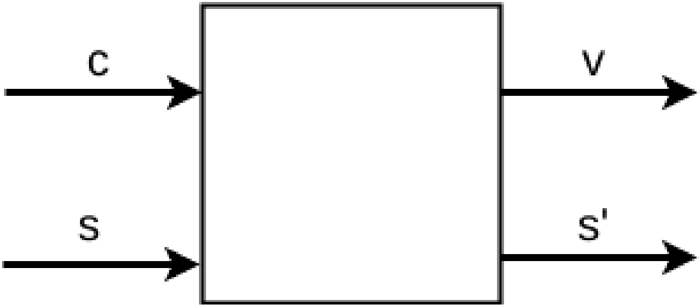

Figura 2: Transformador de estado

- Isso pode ser representado por uma função `b -> ST a`

- Agora podemos definir ST como pertencente as classes Functor, Applicative e Monads. Mas para isso ST deve ser um novo tipo e não um apelido:

```haskell
newtype ST a = S (State -> (a, State))
```

- Vamos criar uma função auxiliar para aplicar um transformador de estado em um estado (que está encapsulado no construtor S):

```haskell
type State = Int
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) s = st s
```

- A ideia geral de um Functor ST é que ele defina como aplicar uma função pura do tipo `a -> b` na parte do valor do resultado de um `ST a`, transformando-o efetivamente em um tipo `ST b`:

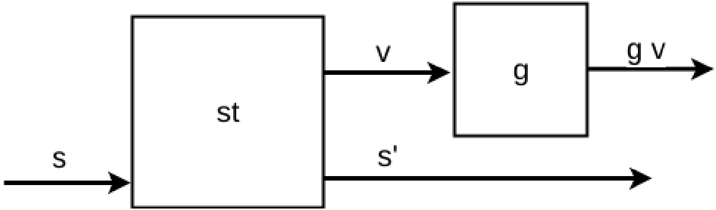

Figura 3: Functor ST

Com isso temos:

```haskell
type State = Int
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) s = st s

instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  -- x :: a , y :: b
  fmap g st = S (\s -> (y, s'))
```

Functor ST

- As definições de `y` e `s'` são obtidas da aplicação do transformador de estado `st` em um estado `s`:

```haskell
type State = Int
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) s = st s

instance Functor ST where
  -- fmap :: (a -> b) -> ST a -> ST b
  -- x :: a
  fmap g st = S stb
    where
      stb s = (g x, s')
        where (x, s') = app st s
```

Functor ST

- Esse Functor promete aplicar uma função pura apenas no valor de saída do transformador de estado, sem influenciar o estado.
- Se em `rlabel` eu quiser gerar rótulos pares, poderia aplicar `fmap (*2)` a função de estado.

- A classe Applicative define formas de combinar computações sequenciais puras dentro de computações que podem sofrer efeitos colaterais.
- Embora cada computação na sequência possa alterar o estado `s`, o valor final é a computação dos valores puros.

A definição de `pure` cria um transformador de estado puro, ou seja, que não altera o estado:

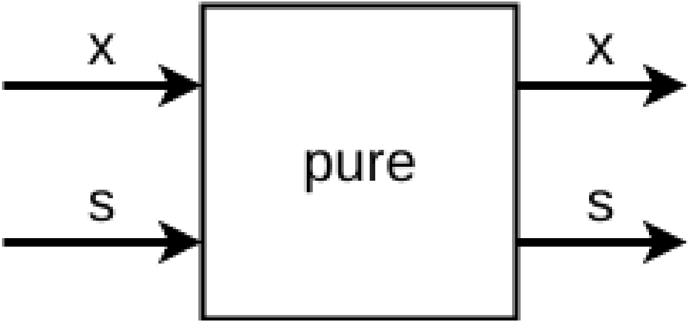

Figura 4: pure ST

Então definimos:

```haskell
newtype ST a = S (State -> (a, State))

instance Applicative ST where
  -- pure :: a -> ST a
  pure x = S (\s -> (x,s))
```

- A definição do operador `(<*>)` define a sequência de mudança de estados pelos transformadores e a combinação dos valores finais como um resultado único:

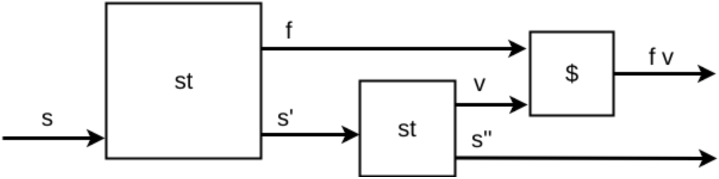

Figura 5: <*> ST

UFABC

Então definimos:

```
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) s = st s

instance Applicative ST where
  -- <*> :: ST (a -> b) -> ST a -> ST b
  stf <*> stx = S stb
    where stb s = (f x, s'')
      where (f, s')  = app stf s
            (x, s'') = app stx s'
```

No nosso exemplo de `rlabel`, podemos imaginar algo como:

```
pure Leaf <*> sInc
```

- Se `sInc` é um transformador de estados que incrementa um contador, então:
  1. `pure Leaf` é aplicado no estado atual `s` retornando ele mesmo (pois é puro)
  2. `sInc` é aplicado a `s` retornando um novo estado com o contador incrementado: `(Leaf n, n + 1)`

No caso de Monads, queremos definir um operador `(>>=)` que se comporte como:

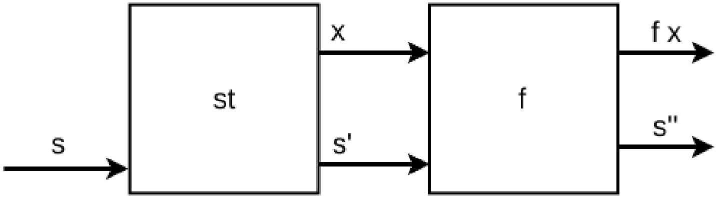

Figura 6: Monad ST

Monad ST

- Podemos observar que o operador **bind** age de forma similar a `(<*>)`, porém cada encadeamento gera um novo transformador de estado que pode depender do valor retornado pelo transformador anterior.
- Ou seja, um Monad ST pode ser usado quando queremos gerar novos transformadores dependendo do valor de retorno de outro.

Com isso definimos:

```haskell
newtype ST a = S (State -> (a, State))

app :: ST a -> State -> (a, State)
app (S st) s = st s

instance Monad ST where
  -- (>>=) :: ST a -> (a -> ST b) -> ST b
  st >>= f = S stb
    where stb s = app (f x) s'
                  where (x, s') = app st s
```

No nosso exemplo de `rlabel`, podemos imaginar algo como:

```
do n <- sInc
   return (Leaf n)
```

para alterar o rótulo de um nó folha.

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n + 1)
rlabel (Node l r) = (Node l' r', n'')
  where
    (l', n')   = rlabel l n  -- altera o estado de n
    (l'', n'') = rlabel r n' -- altera o estado de n'
```

A versão completa do Applicative rlabel fica:

```haskell
alabel :: Tree a -> ST (Tree Int)
alabel (Leaf _)   = pure Leaf <*> sInc
alabel (Node l r) = pure Node <*> alabel l <*> alabel r
```

Monad rlabel

UFABC

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

rlabel :: Tree a -> Int -> (Tree Int, Int)
rlabel (Leaf _) n = (Leaf n, n + 1)
rlabel (Node l r) = (Node l' r', n'')
  where
    (l', n')   = rlabel l n  -- altera o estado de n
    (l'', n'') = rlabel r n' -- altera o estado de n'
```

A versão completa do Monad rlabel fica:

```haskell
mlabel :: Tree a -> ST (Tree Int)
mlabel (Leaf _)   = do n <- sInc
                       return (Leaf n)
mlabel (Node l r) = do l' <- alabel l
                          r' <- alabel r
                          return (Node l' r')
```

Gerador de estado incremental

Finalmente, a definição de `sInc` fica:

```
sInc :: ST Int
sInc = S (\n -> (n, n+1))
```

Para aplicar essas funções, devemos fazer:

```
tree :: Tree Char
tree = Node (Node (Leaf 'a') (Leaf 'b')) (Leaf 'c')

newtype ST a = S (State -> (a, State))
app :: ST a -> State -> (a, State)
app (S st) s = st s

sInc :: ST Int
sInc = S (\n -> (n, n+1))

alabel :: Tree a -> ST (Tree Int)
alabel (Leaf _)   = pure Leaf <*> sInc
alabel (Node l r) = pure Node <*> alabel l <*> alabel r

> stTree = alabel tree -- cria transformador de estado
> app stTree 0         -- começa a contar do 0
(Node (Node (Leaf 0) (Leaf 1)) (Leaf 2), 3)
```

State IO

- Conforme discutimos anteriormente, funções de **entrada e saída** de dados são **impuras** pois alteram o estado atual do sistema.
- A função `getChar` captura um caracter do teclado. Se eu executar tal função duas vezes, o valor da função não necessariamente será igual.
- A função `putChar` escreve um caracter na saída padarão (ex.: monitor). Se eu executar duas vezes seguidas com a mesma entrada, a saída será diferente.

Basicamente, as funções de entrada e saída alteram estado, ou seja:

```haskell
newtype IO a = newtype ST a = State -> (a, State)
```

com a definição de estado sendo:

```haskell
type State = Environment
```

o estado sendo o ambiente, sistema operacional, o mundo computacional que ele vive.

Com isso, tudo que fizemos até agora é suficiente para trabalharmos com IO sem afetar a pureza dos nossos programas:

```
getchar :: IO Char

putChar :: Char -> IO ()
```

Se eu fizer:

```
do putChar 'a'
   putChar 'a'
```

Na verdade ele estará fazendo algo como:

```
(_, env')  = putChar 'a' env
(_, env'') = putChar 'a' env'
```

- No Haskell chamamos as funções de entrada e saída como **ações de IO** (**IO actions**).
- As funções básicas são implementadas internamente de acordo com o Sistema Operacional

Ações básicas

Vamos trabalhar inicialmente com três ações básicas:

```haskell
-- recebe um caracter da entrada padrão
getChar :: IO Char

-- escreve um caracter na saída padrão
putChar :: Char -> IO ()

-- retorna um valor puro envolvido de uma ação IO
return :: a -> IO a
```

Funções auxiliares: `getLine`

Em vez de capturar apenas um caracter, podemos capturar uma linha inteira de informação. Podemos escrever `getLine` da seguinte maneira:

```haskell
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)
```

Atenção!

A função **`return` não se comporta como em outras linguagens**!

Lembre-se: **`return`** apenas pega um valor puro e o coloca no em um contexto. Ele não interrompe a execução.

Exercício 1

Escreva as instruções do `else` como Applicative

```haskell
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
                return []
             else
                do xs <- getLine
                   return (x:xs)
```

Funções auxiliares: putStr

A função inversa escreve uma String na saída padrão:

```haskell
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr  xs

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar 'n'
```

Exercício 2

Escreva a função `putStrLn` usando Applicative.

```haskell
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar 'n'
```

Leitura de Arquivos

Imagine o seguinte arquivo de dados, `exemploData.txt`:

1.2	3.5	2.3
4.1	2.1	3.4
...	...	...

Queremos ler seu conteúdo e transformar em uma lista de listas:

```
[ [1.2, 3.5, 2.3], [4.1, 2.1, 3.4], ⋯]
```

Lendo arquivos

- A função **`readFile`** lê o arquivo em FilePath e retorna ele como String (envolvido em um IO).

```
readFile :: FilePath -> IO String
```

- Vamos criar uma função `parseFile` que fará a conversão, a assinatura dela deve ser:

```
parseFile :: String -> [[Double]]
```

- Queremos que cada linha do arquivo seja uma lista de Doubles:

```haskell
parseFile :: String -> [[Double]]
parseFile file =  map parseLine (lines file)
```

A função parseLine converte cada palavra da linha em um Double:

```haskell
parseFile :: String -> [[Double]]
parseFile file =  map parseLine (lines file)
  where
    parseLine l = map toDouble (words l)
    toDouble  w = read w :: Double
```

Nossa função `readMyFile` ficaria:

```haskell
readMyFile :: [FilePath] -> IO [[Double]]
readMyFile []     = error "./readMyFile nome-do-arquivo"
readMyFile [name] = do conteudo <- readFile name
                       return (parseFile conteudo)
```

E a main:

```haskell
main = do args <- getArgs
          conteudo <- readMyFile args
          print conteudo
```

Exercício 3

Reescreva a função `readMyFile` utilizando Applicative

```haskell
readMyFile :: [FilePath] -> IO [[Double]]
readMyFile []     = error "./readMyFile nome-do-arquivo"
readMyFile [name] = do conteudo <- readFile name
                       return (parseFile conteudo)
```

Exercício 4

Reescreva a função `readMyFile` utilizando Functor.

```
readMyFile :: [FilePath] -> IO [[Double]]
readMyFile []     = error "./readMyFile nome-do-arquivo"
readMyFile [name] = do conteudo <- readFile name
                       return (parseFile conteudo)
```

Tópicos Extras

Monoid

Um Monoid é um conjunto de valores associados a um operador binário associativo e um elemento identidade:

- Valores inteiros com o operador `+` e o elemento 0
- Valores inteiros com o operador `*` e o elemento 1
- Valores String com o operador `++` e o elemento ”

A classe Monoid é definida como:

```
class Monoid a where
   mempty  :: a
   mappend :: a -> a -> a

   mconcat :: [a] -> a
   mconcat = foldr mappend mempty
```

Monoid de Listas

Para listas temos a seguinte instância de Monoid:

```haskell
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

Para o tipo Maybe podemos definir:

```haskell
instance Monoid a => Monoid (Maybe a) where
  mempty   = Nothing

  Nothing `mappend` my      = my
  mx      `mappend` Nothing = mx
  Just x  `mappend` Just y  = Just (x `mappend` y)
```

Monoid de Monad

- Em teoria das categorias um Monad pode ser visto como um Monoid das categorias dos Functors.
  - O elemento identidade é o `return`
  - O operador associativo é uma variação de `(>>=)` com a assinatura:

```haskell
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
```

Ou seja, duas funções que transformam um valor puro em um Monad podem ser combinadas formando uma terceira função.

(>=>) também chamado de operador peixe (fish operator).

Foldable

UFABC

- A importância dos Monoids está na generalização em como combinar uma lista de valores de um tipo que pertença a essa classe.
- Sabendo que o tipo `a` é um Monoid, podemos definir:

```haskell
fold :: Monoid a => [a] -> a
fold []     = mempty
fold (x:xs) = x `mappend` fold xs
```

Essa generalização pode ser feita para outras estruturas:

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

fold :: Monoid a => Tree a -> a
fold (Leaf x)   = x
fold (Node l r) = fold l `mappend` fold r
```

Podemos então criar a classe dos ”dobráveis”:

```haskell
class Foldable t where
   fold    :: Monoid a => t a -> a
   foldMap :: Monoid b => (a -> b) -> t a -> b
   foldr   :: (a -> b -> b) -> b -> t a -> b
   foldl   :: (a -> b -> a) -> a -> t b -> a
```

UFABC

Considere os seguintes tipos:

```haskell
newtype Sum a = Sum a
   deriving (Eq, Ord, Show, Read)

newtype Prod a = Ord a
   deriving (Eq, Ord, Show, Read)

getSum :: Sum a -> a
getSum (Sum x) = x

getProd :: Prod a -> a
getProd (Prod x) = x
```

Exemplo prático

Considere os seguintes Monoids:

```haskell
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x+y)

instance Num a => Monoid (Prod a) where
    mempty = Prod 1
    Prod x `mappend` Prod y = Prod (x*y)
```

Exemplo prático

UFABC

Para efetuar a somatória e produtória de uma lista de números basta fazer:

```
> getSum (foldMap Sum [1..10])
55

> getProd (foldMap Prod [1..10])
3628800
```

Se definirmos a instância de `Foldable` para o tipo `Tree`, bastaria fazer:

```
> getSum (foldMap Sum arvore)
> getProd (foldMap Prod arvore)
```

As funções são as mesmas!!!

Outras funções Foldable

A classe Foldable também define por padrão diversas funções auxiliares:

```haskell
null    :: t a -> Bool
length  :: t a -> Int
elem    :: Eq a => a -> t a -> Bool
maximum :: Ord a => t a -> a
minimum :: Ord a => t a -> a
sum     :: Num a => t a -> a
product :: Num a => t a -> a
toList  :: t a -> [a]
```

- O MapReduce[1] é um modelo de programação utilizado largamente em clusters.
  - Google, Amazon, Yahoo! e outros estão entre seus grandes utilizadores
- Muitas ferramentas como Apache Hadoop e Apache Spark são baseadas neste modelo
- O **MapReduce nada mais é que a função `foldMap`**
  - O *Map* é feito em paralelo e de maneira distribuída (o que é fácil já que o map é puro)
  - O *Reduce* corresponde ao *fold*
  - Como a redução ou folding é feito em uma ordem arbitrária é imprescindível que se trabalhe utilizando um monoid[2] para que o resultado seja consistente!

[1] https://en.wikipedia.org/wiki/MapReduce
[2] https://en.wikipedia.org/wiki/Monoid#MapReduce

Exercício 5

Implemente `toList` utilizando `foldMap`.

```
toList  :: t a -> [a]
```

- Ela transforma qualquer estrutura `Foldable` em uma lista!
- Uma vez que já temos todas as funções anteriores para Listas, basta fornecer a definição de `foldMap` para a instância e todo o resto vem por padrão (mas nem sempre eficiente).

Considere a função:

```haskell
average :: [Int] -> Int
average ns = sum ns `div` length ns
```

Ela agora pode ser generalizada para:

```haskell
average :: Foldable t => t Int -> Int
average ns = sum ns `div` length ns
```

UFABC

E agora podemos fazer:

```
> average (Node (Leaf 1) (Leaf 3))
2
```

UFABC

Uma última classe que veremos no curso é a `Traversable` ou seja, tipos que podem ser mapeados:

```haskell
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f =>
                              (a -> f b) -> t a -> f (t b)
```

Traversable

- Essa classe é útil quando, por exemplo, temos uma função que mapeia um tipo `a` para `Maybe b` e temos uma lista de `a`.
- Nesse caso queremos retornar um `Maybe [b]` ao invés de `[Maybe b]`. Isso dá para ser feito utilizando o `Applicative` para listas:

```haskell
class (Functor t, Foldable t) => Traversable t where
  traverse :: Applicative f =>
                              (a -> f b) -> t a -> f (t b)

instance Traversable [] where
  traverse g []     = pure []
  traverse g (x:xs) = pure (:) <*> g x <*> traverse g xs
```

Supondo a função:

```haskell
dec :: Int -> Maybe Int
dec x | x <= 0     = Nothing
      | otherwise = Just (x - 1)

> traverse dec [1,2,3]
Just [0,1,2]
> traverse dec [2,1,0]
Nothing
```

Exercício 6

Escreva a instância de `Traversable` para `Tree`.

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
              deriving Show

class (Functor t, Foldable t) => Traversable t where
   traverse :: Applicative f =>
                             (a -> f b) -> t a -> f (t b)
```