Extensible
Sums and Products

Haskell Meetup 8/9/17

Jake Keuhlen https://github.com/jkeuhlen/talks

Roadmap

- Sum and Product Types
- Open and Closed Types
- What is an Extensible Type?
- extensible-sp
- Extensible Sums
- Extensible Products
- Higher Kinded Sums

What are sums and products?

Sum Types

- `type Bool = True | False`
- Sums are an OR type
  - They can only take on a single of their possible values at any time
- Either a b

Product Types

- `type Product = (String,Int)`
- Products are an AND type
  - They must contain values for all of their constituent types
- (,) a b

Closed Types vs Open Types

- Closed data types can only be extended at their declaration points
  - ADT’s in FP

- Open data types can be extended ‘on the fly’
  - Classes in OOP

Closed Types vs Open Types

- ADT’s
- “It is very cheap to add a new operation on things: you just define a new function. All the old functions on those things continue to work unchanged.”
- “It is very expensive to add a new kind of thing: you have to add a new constructor [to] an existing data type, and you have to edit and recompile every function which uses that type.”

- Classes
- “It is very cheap to add a new kind of thing: just add a new subclass, and as needed you define specialized methods, in that class, for all the existing operations. The superclass and all the other subclasses continue to work unchanged.”
- “It is very expensive to add a new operation on things: you have to add a new method declaration to the superclass and potentially add a method definition to every existing subclass. In practice, the burden varies depending on the method.”

Quotes from @NormanRamsey via SO - https://stackoverflow.com/questions/870919/why-are-haskell-algebraic-data-types-closed

What is an Extensible Type?

- “Extensible datatypes allow a type to be defined as "open", which can later be extended by disjoint union.” - wiki.haskell.org/Extensible_datatypes

- “The disjoint union of two sets A and B is a binary operator that combines all distinct elements of a pair of given sets, while retaining the original set membership as a distinguishing characteristic of the union set.” - http://mathworld.wolfram.com/DisjointUnion.html

What is an Extensible Type?

- Extensible types are not truly open
  - Type families are the only open type in Haskell
- Extensible types are an easily definable type that pulls together groups of other, fully defined types.
- Extensible constraints allow for complex type reasoning

extensible-sp

Github (most up-to-date): https://github.com/jadaska/extensible-sp

Hackage: https://hackage.haskell.org/package/extensible-sp

"The extensible-sp module provides a simple and straight-forward interface to anonymous, extensible sum types (e.g., Either) and product types (e.g., (,)). Generalizations to higher kinded types are provided as well."

extensible-sp

- Sums

```
data (a :|: b) = DataL a | DataR b deriving
(Show, Eq)

class SumClass c s where
  peek :: c -> Maybe s
  lft  :: s -> c

type (w :>|: a)  = (SumClass w a)
```

- Products

```
data (a :&: b) = Prod a b deriving Show

class ProductClass c s where
  grab :: c -> s
  stash  :: s -> c -> c

type (c :>&: a)  = (ProductClass c a)
```

Extensible Sums

- Provide a common wrapper type that makes defining heterogenous lists easy
- Allow you to easily extend type classes for any data type, even those locked away in a third party library
- Allow for powerful type programming

- Sums

```
data (a :|: b) = DataL a | DataR b deriving
(Show, Eq)

class SumClass c s where
  peek :: c -> Maybe s
  lft  :: s -> c

type (w :>|: a)  = (SumClass w a)
```

Extensible Sums - Heterogeneous Lists

- ```haskell
  {-# LANGUAGE TypeOperators #-}
  {-# LANGUAGE FlexibleContexts #-}
  ```
- ```haskell
  type StringIntChar = String :|: Int :|: Char
  xs :: [StringIntChar]
  xs = [lft (1 :: Int), lft ("2" :: String), lft 'c']
  ```

Extensible Sums - Type Class Extensions

```haskell
data OurColors = Black | Gold

data TheirColors = Green | Yellow



class ColorClass a where
  other :: a -> a
  num   :: a -> Int
  allColors :: [a]
```

```haskell
instance ColorClass OurColors where
  other Black = Gold
  other Gold = Black
  num Black = 0
  num Gold = 1
  allColors = [Black,Gold]

instance ColorClass TheirColors where
  other Green = Yellow
  other Yellow = Green
  num Green = 2
  num Yellow = 3
  allColors = [Green,Yellow]
```

Extensible Sums - Type Class Extensions

```haskell
type CollegeColors = OurColors :|: TheirColors
```

```haskell
instance (ColorClass a, ColorClass b) => ColorClass (a :|: b) where
  other (DataL x) = lft $ other x
  other (DataR y) = lft $ other y
  num (DataL x) = num x
  num (DataR y) = num y
  allColors = (DataL <$> allColors) <> (DataR <$> allColors)
```

Extensible Sums - Type Programming

```
type SumType = Int :|: String :|: Char
type OtherSum = Int :|: Bool

let sumX = lft 2 :: SumType
let sumY = lft "Foo" :: SumType
let sumZ = lft 2 :: OtherSum

incrementInt :: (a :>|: Int) => a -> a
incrementInt x = fromMaybe x $ do
  i <- peek x :: Maybe Int
  return $ lft $ i + 1
```

```
xs :: [StringIntChar]
xs = [lft (1 :: Int), lft ("2" :: String), lft
'c']

> incrementInt <$> xs
[DataL (DataR 2),DataL (DataL "2"),DataR 'c']
> incrementInt sumX
DataL (DataR 2)
> incrementInt sumY
DataL (DataL "Foo")
```

Extensible Sums - Type Programming

```
applyIntFxn = (a :>|: Int) => (Int -> Int) -> a -> a
applyIntFxn fxn x = maybe x (lft . fxn) $ peek x
```

```
xs :: [StringIntChar]
xs = [lft (1 :: Int), lft ("2" :: String), lft 'c']
```

```
> applyIntFxn (+1) <$> xs
[DataL (DataR 2),DataL (DataL "2"),DataR 'c']
```

Extensible Products

- Provide a useful extension to state and reader monads

- Products

```
data (a :&: b) = Prod a b deriving Show


class ProductClass c s where
  grab :: c -> s
  stash  :: s -> c -> c

type (c :>&: a)  = (ProductClass c a)
```

Extensible Products - State

- Simple State Example
- runGame outputs 2
- What if we need to expand `GameState`?

```haskell
module StateGame where
import Control.Monad.State
type GameValue = Int
type GameState = (Bool, Int)
playGame :: String -> State GameState GameValue
playGame []     = do
    (_, score) <- get
    return score
playGame (x:xs) = do
    (on, score) <- get
    case x of
         'a' | on -> put (on, score + 1)
         'b' | on -> put (on, score - 1)
         'c'      -> put (not on, score)
         _        -> put (on, score)
    playGame xs

startState = (False, 0)
runGame = print $ evalState (playGame "abcaaacbbcabbab" )
startState
```

*Example taken from
https://wiki.haskell.org/State_Monad#Complete_and_Concrete_Example_1

Extensible Products - State

- Switch to extensible product type for your state, instead of a normal tuple
- runGame still outputs 2
- More code for this case, but playGame now is no longer dependent on your GameState

```haskell
type GameState = Bool :&: Int
playGame :: (s :>&: Bool, s :>&: Int) => String -> State s GameValue
playGame [] = do
  score :: Int <- grab <$> get
  return score
playGame (x:xs) = do
  on :: Bool <- grab <$> get
  score :: Int <- grab <$> get
  case x of
    'a' | on -> modify $ stash $ score + 1
    'b' | on -> modify $ stash $ score - 1
    'c'      -> modify $ stash $ not on
    _        -> return ()
  playGame xs
startState = False <& 0
runGame = print $ evalState (playGame "abcaaacbbcabbab" :: State
GameState GameValue) startState
```

*Example taken from
https://wiki.haskell.org/State_Monad#Complete_and_Concrete_Example_1

Extensible Products - Reader

- Reader can benefit from extensible products in much the same way as state!
- What happens when we need to add a boolean to `Config`?

```haskell
type Config = (Int,String)

useIntConfig :: Reader Config Int
useIntConfig = do
  (i,_) <- ask
  return $ i `mod` 10

useStringConfig :: Reader Config String
useStringConfig = do
  (_,s) <- ask
  return $ reverse s

runConfig i = print $ runReader useIntConfig
(i,"foo")
runConfig2 = print $ runReader useStringConfig
(0,"foo")
```

Extensible Products - Reader

- Reader can benefit from extensible products in much the same way as state!
- What happens when we need to add a boolean to `Config`?

```
type Config = Int :&: String :&: Bool

useIntConfig :: (r :>&: Int) => Reader r Int
useIntConfig = do
  i <- grab <$> ask
  return $ i `mod` 10

useStringConfig :: (r :>&: String) => Reader r String
useStringConfig = do
  s <- grab <$> ask
  return $ reverse s

runConfig :: Int -> IO ()
runConfig i = print $ runReader useIntConfig
                 (i <& ("foo" :: String) <& True <& 10.0
runConfig2 = print $ runReader useStringConfig "foo" ::
String)
```

Higher Kinded Sums

- Extensible sums and products also generalize nicely to higher kinded extensions
- Further applications for Free

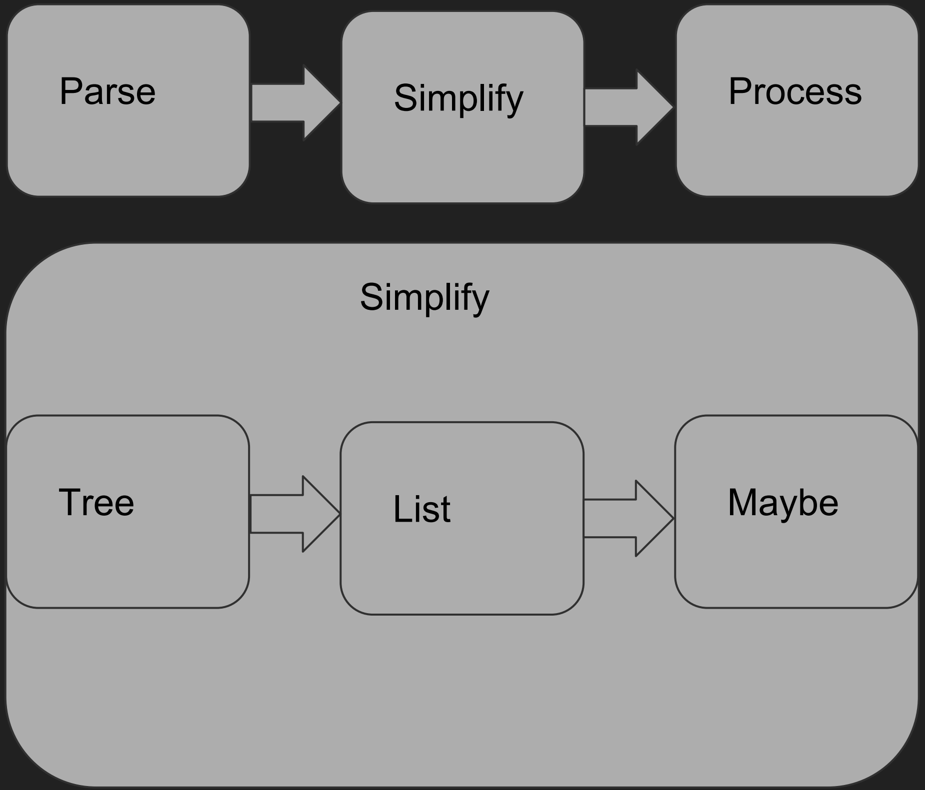
```haskell
data (f :||: g) a b = InL (f a b) | InR (g a b)
class Sum1 c s where
  peek1 :: c a b -> Maybe (s a b)
  lft1  :: s a b -> c a b
type (w :>||: a) = (Sum1 w a)
```

```haskell
data (f :+: g) a b = InL (f a b) | InR (g a b)
class Sum2 c s where
  peek2 :: c a b -> Maybe (s a b)
  lft2  :: s a b -> c a b
type (w :>+: a) = (Sum2 w a)
```

Higher Kinded Sums

- Start with a simple design pattern:
  - `Parse :: String -> [Int]`
  - `Simplify :: [Int] -> Int`
  - `Process :: Int -> IO ()`
- Ends up more complex:
  - `Parse :: String -> Either (Tree Int) [Int]`
  - `Simplify :: Either (Tree Int) [Int] -> Int`
- Ends up even more complex:
  - `Parse :: String -> Either (Either (Tree Int) (Maybe Int)) [Int]`
  - `Simplify :: Either (Either (Tree Int) (Maybe Int)) [Int] -> Int`

Higher Kinded Sums




```
simplifyTree :: Tree Int -> Tree Int
simplifyTree x = Tree (sumTree x)
simplifyList :: [Int] -> [Int]
simplifyList x = [sum x]
simplifyMaybe :: Maybe Int -> Maybe Int
simplifyMaybe = maybe (Just 0) Just

liftFxn :: (sumType :>|: f) => (f a -> f a) ->
sumType a -> sumType a
liftFxn f x = maybe x (lft1 . f) $ peek x

simplify =   liftFxn simplifyTree
           . liftFxn simplifyList
           . liftFxn simplifyMaybe
```

Questions?