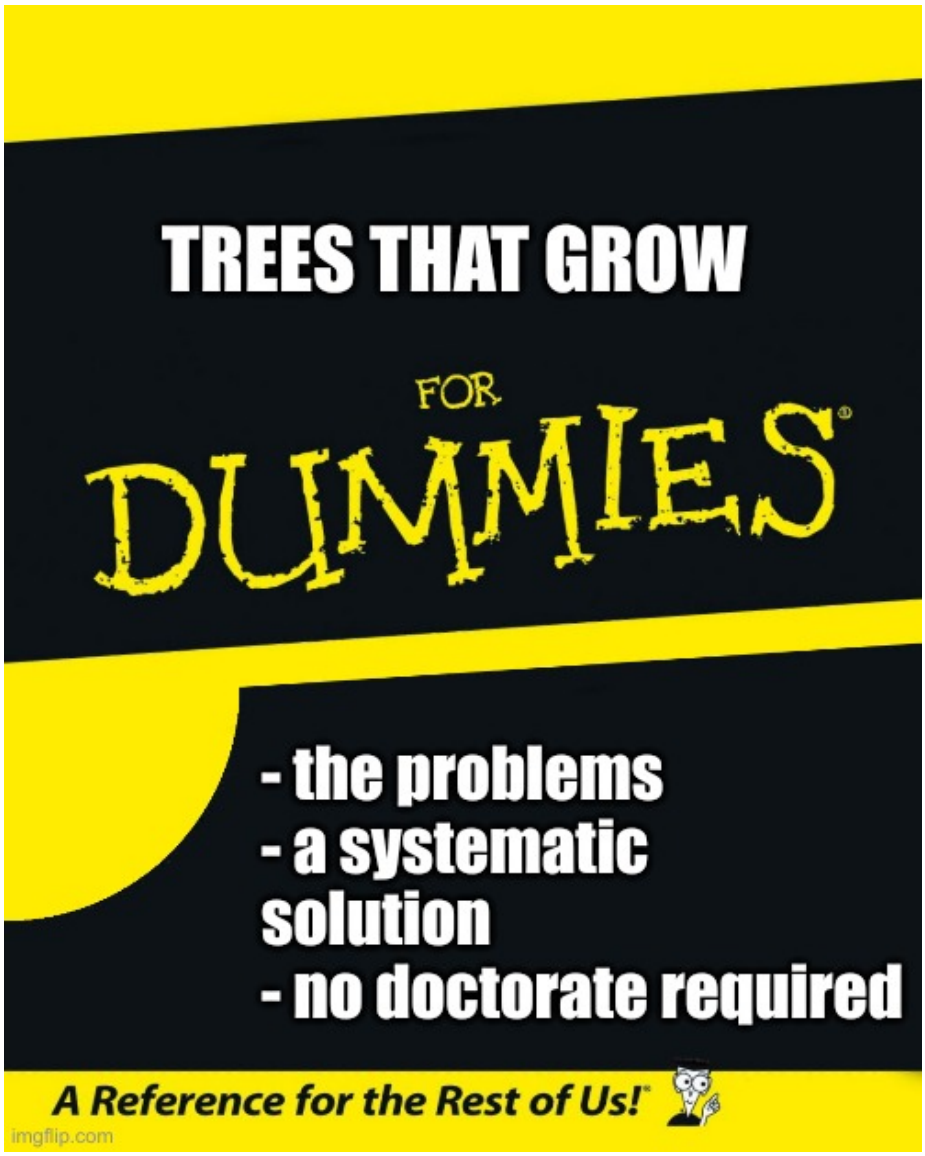
Trees That Grow for Dummies



$ whoami

- Julien Debon
- Sir4ur0n
- Java / Haskell
- Decathlon

Trees

```
data Order = Order {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map Item Quantity
}

data Item = Item {
  identifier :: ItemId
}

prepareOrder :: Order -> IO ()
```

... that grow

Price

What we have

```
-- | API Provided by Price team
loadCurrentPrice :: ItemId -> IO Price
```

What we want

```
quoteBasket :: Map Item Quantity -> Price
```

How do we implement the price feature?

Solution 1

Add a `Price` field to `Item`

```
data Order = Order {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map Item Quantity
}

data Item = Item {
  identifier :: ItemId,
  price :: Price
}
```

```haskell
data Order = Order {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map Item Quantity
}

data Item = Item {
  identifier :: ItemId,
  price :: Price
}

quoteBasket :: Map Item Quantity -> Price
```

```haskell
data Order = Order {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map Item Quantity
}

data Item = Item {
  identifier :: ItemId,
  price :: Price
}

prepareOrder :: Order -> IO ()
```

Problem #1

Unnecessary pricing

Problem #1 bis

"Special value" for Price (`0`, `-ϖ`, etc.)

Make impossible states impossible

Solution 2

Add a `Maybe Price` field to `Item`

```haskell
data Order = Order {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map Item Quantity
}

data Item = Item {
  identifier :: ItemId,
  price :: Maybe Price
}
```

Problem #1

Inconsistency

```haskell
inconsistentBasket :: Map Item Quantity
inconsistentBasket = fromList [
  (Item {identifier = id1, price = Just 42}, 1),
  (Item {identifier = id1, price = Nothing}, 1)
]
```

Problem #2

Unsafe

```haskell
quoteBasket :: Map Item Quantity -> Price

unpricedBasket :: Map Item Quantity
unpricedBasket = fromList [
  (Item {identifier = id1, price = Nothing}, 1)
]

quoteBasket unpricedBasket



quoteBasket :: Map Item Quantity -> Maybe Price

unpricedBasket :: Map Item Quantity
unpricedBasket = fromList [
  (Item {identifier = id1, price = Nothing}, 1)
]

quoteBasket unpricedBasket == Nothing
```

Constraints must be pushed upstream, not downstream

The golden rule of software quality, Gabriel Gonzalez

The golden rule of software quality, Gabriel Gonzalez

```haskell
-- | Downstream constraint
head :: [a] -> Maybe a

-- | Upstream constraint
head :: NonEmpty a -> a
```

```haskell
-- | Downstream constraint
head :: [a] -> Maybe a

myList = [1,2,3]
case head myList of
  Nothing -> ???
  Just firstElem -> myBusiness firstElem

-- | Upstream constraint
head :: NonEmpty a -> a

myList = 1 :| [2,3]
firstElem = head myList
myBusiness firstElem
```

```haskell
-- | Downstream constraint
quoteBasket :: Map Item Quantity -> Maybe Price

-- | Upstream constraint
quoteBasket :: ??? -> Price
```

Solution 3

Duplicate types

```haskell
data UnpricedOrder = UnpricedOrder {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map UnpricedItem Quantity
}

data UnpricedItem = UnpricedItem {
  identifier :: ItemId
}

data PricedOrder = PricedOrder {
  recipientName :: Text,
  recipientAddress :: Address,
  shoppingBasket :: Map PricedItem Quantity
}

data PricedItem = PricedItem {
  identifier :: ItemId,
  price :: Price
}

quoteBasket :: Map PricedItem Quantity -> Price
```

Problem #1

Extra noise/code

```haskell
prepareUnpricedOrder :: UnpricedOrder -> IO ()
preparePricedOrder :: PricedOrder -> IO ()

prepareOrder :: Either UnpricedOrder PricedOrder -> IO ()

removePrice :: PricedOrder -> UnpricedOrder
prepareOrder :: UnpricedOrder -> IO ()
```

Problem #2

Double maintenance

Problem #3

Invisible coupling => Unsafe to add fields

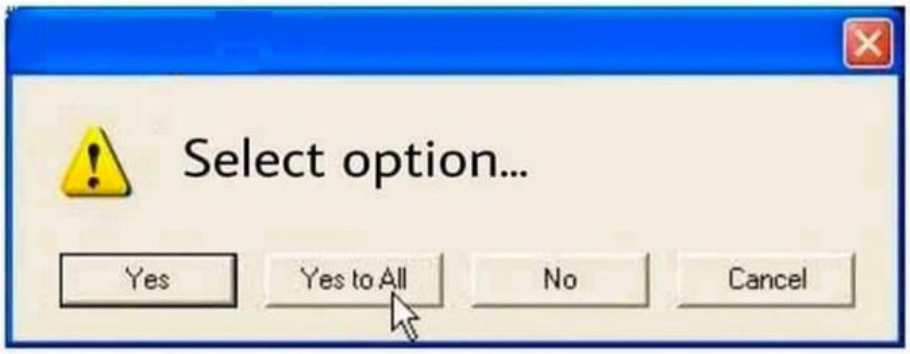
```haskell
data UnpricedItem = UnpricedItem {
  identifier :: ItemId,
```

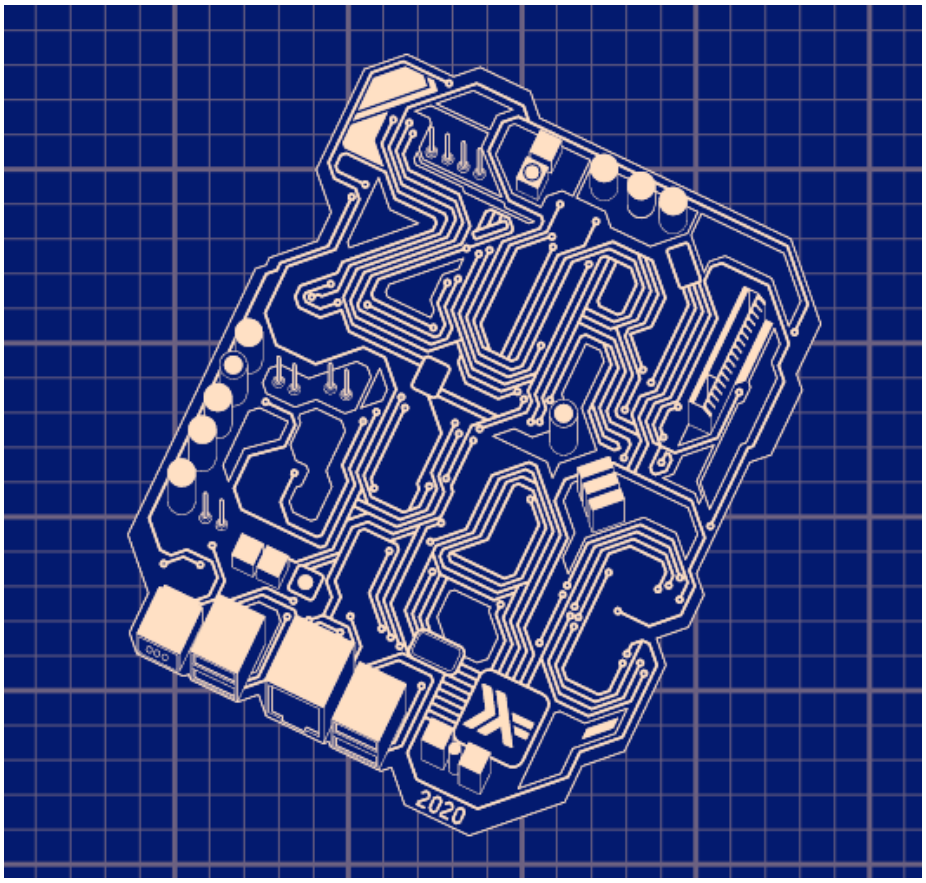
```
  description :: Text
}

data PricedItem = PricedItem {
  identifier :: ItemId,
  price :: Price
}
```

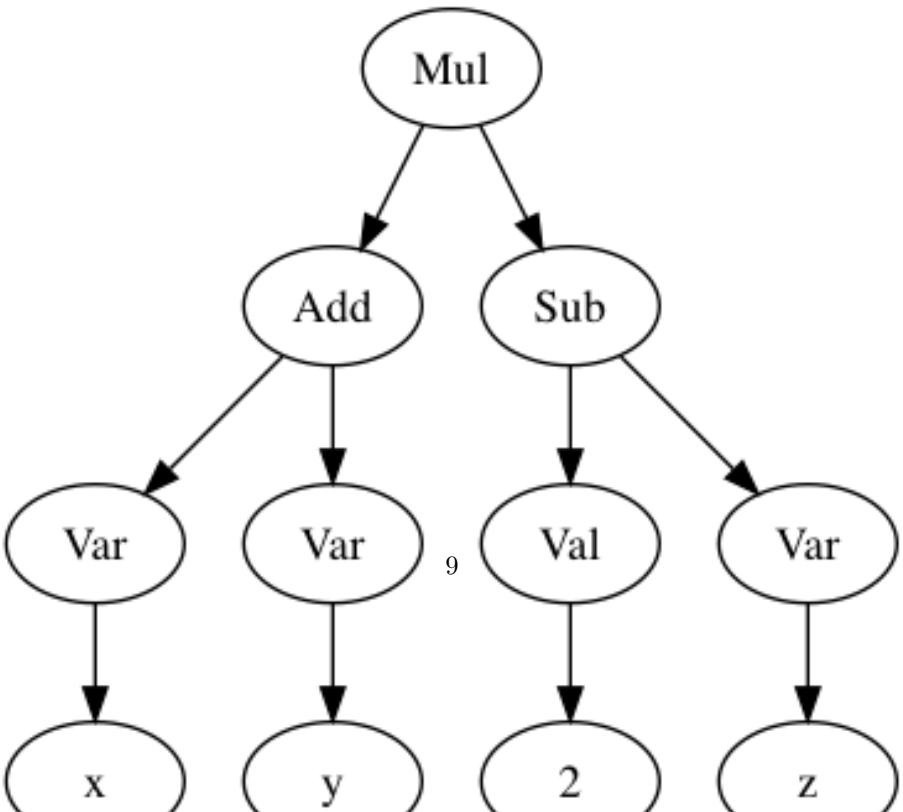
Criteria

- Full type safety
- Functional coupling is type checked
- Be able to express “Pricing is not required” in functions
- Be able to express “Pricing does matter” in functions (upstream constraint)
- Low verbosity
- Low maintenance





$(x + y) * (2 - z)$



Appeared in Journal of Universal Computer Science, vol. 23, no. 1 (2017), 42-62

Trees that Grow

Shayan Najd
(Laboratory for Foundations of Computer Science
The University of Edinburgh, Scotland, U.K.
sh.najd@gmail.com)

Simon Peyton Jones
(Microsoft Research, Cambridge, U.K.
simonpj@microsoft.com)

extensibility in functional data types

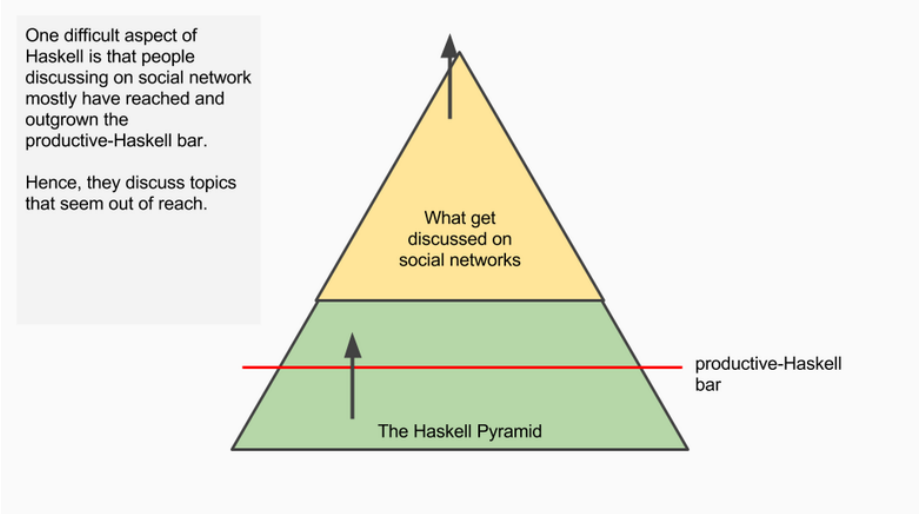
adding new data constructors, and/or [..] adding new fields to its existing data constructors

$$
\begin{array}{l}
\textbf{data}\ Exp_X\ \xi = Lit_X\ \ (X_{Lit}\ \ \xi)\ Integer \\
\qquad\qquad\ \ \mid Var_X\ \ (X_{Var}\ \ \xi)\ Var \\
\qquad\qquad\ \ \mid Ann_X\ \ (X_{Ann}\ \ \xi)\ (Exp_X\ \xi)\ Typ \\
\qquad\qquad\ \ \mid Abs_X\ \ (X_{Abs}\ \ \xi)\ Var\ \ \ (Exp_X\ \xi) \\
\qquad\qquad\ \ \mid App_X\ \ (X_{App}\ \ \xi)\ (Exp_X\ \xi)\ (Exp_X\ \xi) \\
\qquad\qquad\ \ \mid Exp_X\ \ (X_{Exp}\ \ \xi)
\end{array}
\qquad
\begin{array}{l}
\textbf{type family}\ X_{Lit}\ \ \xi \\
\textbf{type family}\ X_{Var}\ \ \xi \\
\textbf{type family}\ X_{Ann}\ \ \xi \\
\textbf{type family}\ X_{Abs}\ \ \xi \\
\textbf{type family}\ X_{App}\ \ \xi \\
\textbf{type family}\ X_{Exp}\ \ \xi
\end{array}
$$

$$
\boxed{\Gamma \vdash M : A}
\qquad
\frac{}{\Gamma \vdash i : \mathbf{Int}}
\qquad
\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}
\qquad
\frac{\Gamma \vdash M : A}{\Gamma \vdash M :: A : A}
$$

$$
\frac{x : A, \Gamma \vdash N : B}{\Gamma \vdash \boldsymbol{\lambda} x.\ N : A \to B}
\qquad
\frac{\Gamma \vdash L : A \to B \quad \Gamma \vdash M : A}{\Gamma \vdash L\ M : B}
$$

$$
\begin{array}{l}
\textbf{type}\ Exp^{TC} = Exp_X\ TC \\
\textbf{data}\ TC \\
\textbf{type instance}\ X_{Lit}\ \ TC = Void \\
\textbf{type instance}\ X_{Var}\ \ TC = Void
\end{array}
\qquad
\begin{array}{l}
\textbf{type instance}\ X_{Ann}\ \ TC = Void \\
\textbf{type instance}\ X_{Abs}\ \ TC = Void \\
\textbf{type instance}\ X_{App}\ \ TC = Typ \\
\textbf{type instance}\ X_{Exp}\ \ TC = Void
\end{array}
$$



Demo time

Extension descriptor

$$
\begin{aligned}
\textbf{data}\ Exp_X\ \xi = \ & Lit_X\ (X_{Lit}\ \xi)\ Integer \\
| \ & Var_X\ (X_{Var}\ \xi)\ Var \\
| \ & Ann_X\ (X_{Ann}\ \xi)\ (Exp_X\ \xi)\ Typ \\
| \ & Abs_X\ (X_{Abs}\ \xi)\ Var\ (Exp_X\ \xi) \\
| \ & App_X\ (X_{App}\ \xi)\ (Exp_X\ \xi)\ (Exp_X\ \xi) \\
| \ & Exp_X\ (X_{Exp}\ \xi)
\end{aligned}
$$

– ξ is a type index to Exp_X. We call ξ the *extension descriptor*, because it describes which extension is in use. For example $Exp_X\ TC$ might be a variant of Exp_X for the type checker for a language; we will see many examples shortly.

Extra field

$$\begin{array}{lll} \textbf{data}\ Exp_X\ \xi = Lit_X & (X_{Lit}\ \xi) & Integer \\ \mid Var_X & (X_{Var}\ \xi) & Var \\ \mid Ann_X & (X_{Ann}\ \xi) & (Exp_X\ \xi)\ Typ \\ \mid Abs_X & (X_{Abs}\ \xi) & Var\ \ (Exp_X\ \xi) \\ \mid App_X & (X_{App}\ \xi) & (Exp_X\ \xi)\ (Exp_X\ \xi) \\ \mid Exp_X & (X_{Exp}\ \xi) & \end{array} \qquad \begin{array}{l} \textbf{type family}\ X_{Lit}\ \xi \\ \textbf{type family}\ X_{Var}\ \xi \\ \textbf{type family}\ X_{Ann}\ \xi \\ \textbf{type family}\ X_{Abs}\ \xi \\ \textbf{type family}\ X_{App}\ \xi \\ \textbf{type family}\ X_{Exp}\ \xi \end{array}$$

– Each data constructor C has an extra field of type $X_C\ \xi$, where X_C is a type family, or type-level function [Chakravarty et al., 2005]. We can use this field to extend a data constructor with extra fields (Section 3.3). For example, if we define $X_{App}\ TC$ to be Typ, the App constructor of a tree of type $Exp_X\ TC$ will have a Typ field.

$$\begin{array}{l} \textbf{type}\ Exp^{TC} = Exp_X\ TC \\ \textbf{data}\ TC \\ \textbf{type instance}\ X_{Lit}\ TC = Void \\ \textbf{type instance}\ X_{Var}\ TC = Void \end{array} \qquad \begin{array}{l} \textbf{type instance}\ X_{Ann}\ TC = Void \\ \textbf{type instance}\ X_{Abs}\ TC = Void \\ \textbf{type instance}\ X_{App}\ TC = Typ \\ \textbf{type instance}\ X_{Exp}\ TC = Void \end{array}$$

Extension descriptor Extra field

Add a constructor

Criteria

- Full type safety
- Functional coupling is type checked
- Be able to express "Pricing is not required" in functions
- Be able to express "Pricing does matter" in functions (upstream constraint)
- Low verbosity
- Low maintenance

Caveat #1

- More type machinery/astronomy
- More indirection

Caveat #2

Typeclass instances boilerplate

```haskell
class Foo a where
  foo :: a -> Text

instance Foo (Item 'Unpriced) where
  foo _ = "unpriced"

instance Foo (Item 'Priced) where
  foo _ = "priced"

doesNotCompile :: Item isPriced -> Text
doesNotCompile item = foo item -- Compilation failure
-- No instance for (Foo (Item isPriced)) arising from a use of ‘foo’

doesCompile :: Foo (Item isPriced) => Item isPriced -> Text
doesCompile item = foo item
```

Caveat #3

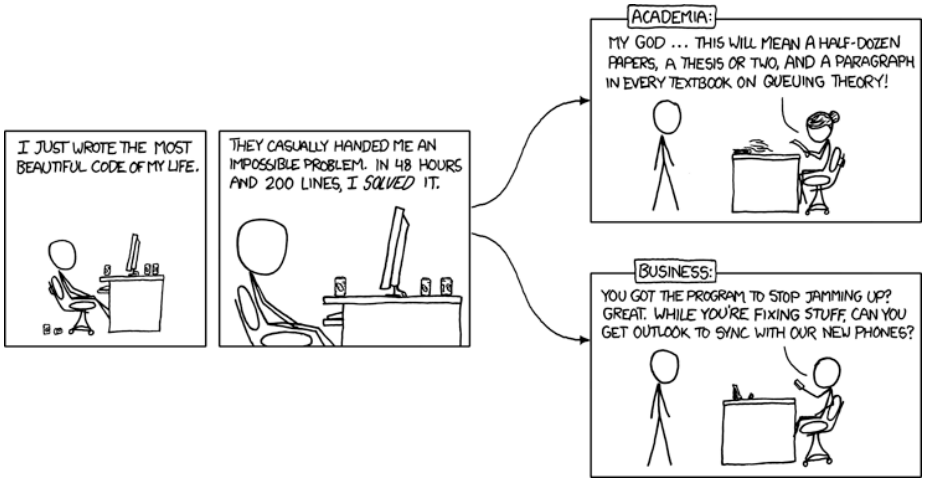
`()` and `Void` don't fare well with `Generic`

Going further

- Type parameters
- Existential types
- GADTs
- Type synonyms for comfort
- Hierarchy of extension descriptors

Conclusion

Questions



References

- https://www.microsoft.com/en-us/research/uploads/prod/2016/11/trees-that-grow.pdf
- http://www.haskellforall.com/2020/07/the-golden-rule-of-software-quality.html
- https://www.youtube.com/watch?v=bhhE2DxbrJM&ab_channel=Z%C3%BCrichFriendsofHaskell
- https://xkcd.com/664/