Functional stuff: GADT, Existential types, Rank-N-Types, ...

Rafał Łasocha

Wrocław, 25th March 2015

- *A* **phantom type** *is a parametrised type whose parameters do not all appear on the right-hand side of its definition*

```
data FormData a = FormData String
```

```haskell
data FormData a = FormData String

data Validated
data Unvalidated
```

```haskell
data FormData a = FormData String

data Validated
data Unvalidated

formData :: String -> FormData Unvalidated
formData str = FormData str
```

```
data FormData a = FormData String

data Validated
data Unvalidated

formData :: String -> FormData Unvalidated
formData str = FormData str

validate :: FormData Unvalidated -> Maybe (FormData Validated)
validate (FormData str) = ...
```

```haskell
data FormData a = FormData String

data Validated
data Unvalidated

formData :: String -> FormData Unvalidated
formData str = FormData str

validate :: FormData Unvalidated -> Maybe (FormData Validated)
validate (FormData str) = ...

useData :: FormData Validated -> IO ()
useData (FormData str) = ...
```

```
data FormData a = FormData String

data Validated
data Unvalidated

formData :: String -> FormData Unvalidated
formData str = FormData str

validate :: FormData Unvalidated -> Maybe (FormData Validated)
validate (FormData str) = ...

useData :: FormData Validated -> IO ()
useData (FormData str) = ...

liftStringFn :: (String -> String) -> FormData a -> FormData a
liftStringFn fn (FormData str) = FormData (fn str)
```

```haskell
data FormData a = FormData String

data Validated
data Unvalidated

formData :: String -> FormData Unvalidated
formData str = FormData str

validate :: FormData Unvalidated -> Maybe (FormData Validated)
validate (FormData str) = ...

useData :: FormData Validated -> IO ()
useData (FormData str) = ...

liftStringFn :: (String -> String) -> FormData a -> FormData a
liftStringFn fn (FormData str) = FormData (fn str)

dataToUpper :: FormData a -> FormData a
dataToUpper = liftStringFn (map toUpper)
```

- **Generalised Algebraic Datatypes** *(GADTs) are datatypes for which a constructor has a non standard type.*

- **Generalised Algebraic Datatypes** *(GADTs) are datatypes for which a constructor has a non standard type.*

```
data Empty
data NonEmpty
data List x y where
   Nil :: List a Empty
   Cons:: a -> List a b -> List a NonEmpty
```

- **Generalised Algebraic Datatypes** *(GADTs) are datatypes for which a constructor has a non standard type.*

```
data Empty
data NonEmpty
data List x y where
   Nil :: List a Empty
   Cons:: a -> List a b -> List a NonEmpty

safeHead:: List x NonEmpty -> x
safeHead (Cons a b) = a
```

- **Generalised Algebraic Datatypes** *(GADTs) are datatypes for which a constructor has a non standard type.*

```
data Empty
data NonEmpty
data List x y where
   Nil :: List a Empty
   Cons:: a -> List a b -> List a NonEmpty

safeHead:: List x NonEmpty -> x
safeHead (Cons a b) = a

silly 0 = Nil
silly 1 = Cons 1 Nil
```

- **Generalised Algebraic Datatypes** *(GADTs) are datatypes for which a constructor has a non standard type.*

```
data Empty
data NonEmpty
data List x y where
   Nil :: List a Empty
   Cons:: a -> List a b -> List a NonEmpty

safeHead:: List x NonEmpty -> x
safeHead (Cons a b) = a

silly 0 = Nil
silly 1 = Cons 1 Nil
```

- it can't infer proper type!

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

```
data Worker b x = Worker {buffer :: b, input :: x}
```

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

```
data Worker b x = Worker {buffer :: b, input :: x}

data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}
```

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

```
data Worker b x = Worker {buffer :: b, input :: x}

data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)
```

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

```
data Worker b x = Worker {buffer :: b, input :: x}

data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)

memoryWorker :: Worker Int
```

- *An* **Existential type** *is a type which is using in constructors parameters which are not parameters of this type (they are not declared in left hand side of the type's definition).*

```
data Worker b x = Worker {buffer :: b, input :: x}

data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)

memoryWorker :: Worker Int
```

- it's impossible for function to **demand** specific Buffer
- you're more limited in what you can do with Worker like that

```
data Worker b x = Worker {buffer :: b, input :: x}
data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)
memoryWorker :: Worker Int
```

```
data Worker b x = Worker {buffer :: b, input :: x}
data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)
memoryWorker :: Worker Int

data NetBuffer = NetBuffer
netWorker = Worker NetBuffer (2 :: Int)
netWorker :: Worker Int
```

```
data Worker b x = Worker {buffer :: b, input :: x}
data Worker x =
  forall b. Buffer b => Worker {buffer :: b, input :: x}

data MemoryBuffer = MemoryBuffer
memoryWorker = Worker MemoryBuffer (1 :: Int)
memoryWorker :: Worker Int

data NetBuffer = NetBuffer
netWorker = Worker NetBuffer (2 :: Int)
netWorker :: Worker Int

workers = [netWorker, memoryWorker]
workers :: [Worker Int]
```

- Rank-N types are types, which are using “forall” keyword in their’s definition, and it cannot be moved above (in AST of type sense).

- Rank-N types are types, which are using “forall” keyword in their’s definition, and it cannot be moved above (in AST of type sense).

```
ghci> let putInList x = [x]
```

- Rank-N types are types, which are using “forall” keyword in their's definition, and it cannot be moved above (in AST of type sense).

```
ghci> let putInList x = [x]
ghci> liftTup putInList (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
ghci> let liftTup liftFunc (a, b) = (liftFunc a, liftFunc b)
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
ghci> let liftTup liftFunc (a, b) = (liftFunc a, liftFunc b)
ghci> liftTup (\x -> [x]) (5, "Hello")
  No instance for (Num [Char]) arising from literal '5'
  ...
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
ghci> let liftTup liftFunc (a, b) = (liftFunc a, liftFunc b)
ghci> liftTup (\x -> [x]) (5, "Hello")
  No instance for (Num [Char]) arising from literal '5'
  ...
ghci> :t liftTup
liftTup :: (t -> t1) -> (t, t) -> (t1, t1)
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
...
ghci> :t liftTup
liftTup :: (t -> t1) -> (t, t) -> (t1, t1)
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
...
ghci> :t liftTup
liftTup :: (t -> t1) -> (t, t) -> (t1, t1)

-- Second try
-- test.hs:
liftTup :: (x -> f x) -> (a, b) -> (f a, f b)
liftTup liftFunc (t, v) = (liftFunc t, liftFunc v)
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- First try:
...
ghci> :t liftTup
liftTup :: (t -> t1) -> (t, t) -> (t1, t1)

-- Second try
-- test.hs:
liftTup :: (x -> f x) -> (a, b) -> (f a, f b)
liftTup liftFunc (t, v) = (liftFunc t, liftFunc v)

ghci> :l test.hs
  Couldnt match expected type 'x' against inferred type 'a'
  ...
```

```
ghci> liftTup (\x -> [x]) (5, "Blah") # We want to achieve this!
  ([5], ["Blah"])

-- Second try
liftTup :: (x -> f x) -> (a, b) -> (f a, f b)
...

-- Third try
liftTup :: (forall x. x -> f x) -> (a, b) -> (f a, f b)

ghci> liftTup putInList (5, "Hello")
([5], ["Hello"])
```

- It uses RankNTypes and GADTs

- It uses RankNTypes and GADTs
- It adds abstract layer above IO

- It uses RankNTypes and GADTs
- It adds abstract layer above IO
- It describes how our program want to use IO
- ...but it doesn't show to the user internals of the communication

- It uses RankNTypes and GADTs
- It adds abstract layer above IO
- It describes how our program want to use IO
- ...but it doesn't show to the user internals of the communication
- It's pure!

```
class Monad m => MonadPrompt p m where
  prompt :: p a -> m a
```

```
class Monad m => MonadPrompt p m where
  prompt :: p a -> m a

data Prompt p r
instance MonadPrompt p (Prompt p)
instance MonadPrompt p (PromptT p m)
```

```
class Monad m => MonadPrompt p m where
  prompt :: p a -> m a

data Prompt p r
instance MonadPrompt p (Prompt p)
instance MonadPrompt p (PromptT p m)

runPrompt :: (forall a. p a -> a) -> Prompt p r -> r
runPromptM :: Monad m => (forall a. p a -> m a) -> Prompt p r ->
```

```
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime
```

```haskell
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime

handleIO :: Request a -> IO a
```

```haskell
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime

handleIO :: Request a -> IO a
handleIO (Echo s) = putStrLn s
handleIO GetLine = catchJust
  (guard . isEOFError)
  (Just <$> getLine)
  (const (return Nothing))
handleIO GetTime = getCurrentTime
```

```
handleIO :: Request a -> IO a
```

```
handleIO :: Request a -> IO a

cat :: Prompt Request ()
cat = do
  line <- prompt GetLine
  maybe (return ()) (\x -> prompt (Echo x) >> cat) line
```

```
handleIO :: Request a -> IO a

cat :: Prompt Request ()
cat = do
  line <- prompt GetLine
  maybe (return ()) (\x -> prompt (Echo x) >> cat) line

runPromptM :: Monad m =>
   (forall a. p a -> m a) ->
   Prompt p r ->
   m r
```

```haskell
handleIO :: Request a -> IO a

cat :: Prompt Request ()
cat = do
  line <- prompt GetLine
  maybe (return ()) (\x -> prompt (Echo x) >> cat) line

runPromptM :: Monad m =>
   (forall a. p a -> m a) ->
   Prompt p r ->
   m r

runCat :: IO ()
runCat = runPromptM handleIO cat
```

```haskell
handleIO :: Request a -> IO a

cat :: Prompt Request ()
cat = do
  line <- prompt GetLine
  maybe (return ()) (\x -> prompt (Echo x) >> cat) line

runPromptM :: Monad m =>
   (forall a. p a -> m a) ->
   Prompt p r ->
   m r

runCat :: IO ()
runCat = runPromptM handleIO cat
```

- Why `runPromptM` needs to be Rank2Type function?

```
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime
```

- Why `runPromptM` needs to be Rank2Type function?

```
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime

handleIO :: Request a -> IO a
```

- Why `runPromptM` needs to be Rank2Type function?

```
data Request a where
  Echo :: String -> Request ()
  GetLine :: Request (Maybe String)
  GetTime :: Request UTCTime

handleIO :: Request a -> IO a

runPromptM :: Monad m =>
   (forall a. p a -> m a) ->
   Prompt p r ->
   m r
```

- What we achieved?

- What we achieved?
- Why is that cool?

RWS monad

- RWS (Reader, Writer, State) Monad

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)
  - We won't use Reader monad from RWS.

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)
  - We won't use Reader monad from RWS.
- Writer: Something, to which we can write data (for example, logging to file)

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)
  - We won't use Reader monad from RWS.
- Writer: Something, to which we can write data (for example, logging to file)
  - `tell` function is writing to the Writer monad

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)
  - We won't use Reader monad from RWS.
- Writer: Something, to which we can write data (for example, logging to file)
  - `tell` function is writing to the Writer monad
- State we already know.

RWS monad

- RWS (Reader, Writer, State) Monad
- Reader: Something, from which we can read data (for example, configuration datatype)
    - We won't use Reader monad from RWS.
- Writer: Something, to which we can write data (for example, logging to file)
    - `tell` function is writing to the Writer monad
- State we already know.
    - `get` gets the state
    - `put` sets the state

```
type Input = [String]
type Output = [String]
```

```
type Input = [String]
type Output = [String]

handleRWS :: Request a -> RWS r Output Input a
handleRWS (Echo s) = tell (return s)
handleRWS GetLine = do
  lines <- get
  if null lines
    then return Nothing
    else do
      put (tail lines)
      return (Just (head lines))
```

```
type Input = [String]
type Output = [String]

handleRWS :: Request a -> RWS r Output Input a
handleRWS (Echo s) = tell (return s)
handleRWS GetLine = do
  lines <- get
  if null lines
    then return Nothing
    else do
      put (tail lines)
      return (Just (head lines))

rwsCat :: RWS r Output Input ()
rwsCat = runPromptM handleRWS cat
```

```haskell
type Input = [String]
type Output = [String]

handleRWS :: Request a -> RWS r Output Input a
handleRWS (Echo s) = tell (return s)
handleRWS GetLine = do
  lines <- get
  if null lines
    then return Nothing
    else do
      put (tail lines)
      return (Just (head lines))

rwsCat :: RWS r Output Input ()
rwsCat = runPromptM handleRWS cat

simulateCat :: Input -> Output
simulateCat input = snd $ evalRWS rwsCat undefined input
```

- Red-black trees properties:
  - every black node has two black children
  - every path from the root to an empty tree passes through the same number of black nodes

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

- Red is really MaybeRed

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

- `Red` is really `MaybeRed`
- `C (...)` constructor means that there's Black node inside

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

- Red is really `MaybeRed`
- `C (...)` constructor means that there's Black node inside
- When there's Red node inside of Red node, we need to do something. Thus, type `Red (Red Black) a b` means dangerous situation.

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

- Red is really `MaybeRed`
- `C (...)` constructor means that there's Black node inside
- When there's Red node inside of Red node, we need to do something. Thus, type `Red (Red Black) a b` means dangerous situation.
- Note that Black nodes are always increasing their depth, by passing it down increased by one (look at `[b]` at Black constructor)

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] -> a -> Red Black a [b] -> Red Bl
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] -> a -> Red Black a [b] -> Red Bl
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] -> a -> Red Black a [b] -> Red Bl
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
balanceL (R(a,x,R(b,y,c))) z d = R(B(a,x,C b),y,B(C c,z,d))
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] -> a -> Red Black a [b] -> Red Bl
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
balanceL (R(a,x,R(b,y,c))) z d = R(B(a,x,C b),y,B(C c,z,d))
balanceL (R(C a,x,C b)) z d = C(B(R(a,x,b),z,d))
```

```
type Tr t a b = (t a b,a,t a b)
data Red t a b = C (t a b) | R (Tr t a b)
data Black a b = E | B(Tr (Red Black) a [b])

balanceL :: Red (Red Black) a [b] -> a -> Red Black a [b] -> Red Bl
balanceL (R(R(a,x,b),y,c)) z d = R(B(C a,x,C b),y,B(c,z,d))
balanceL (R(a,x,R(b,y,c))) z d = R(B(a,x,C b),y,B(C c,z,d))
balanceL (R(C a,x,C b)) z d = C(B(R(a,x,b),z,d))
balanceL (C a) x b = C(B(a,x,b))
```

```
insB :: Ord a => a -> Black a b -> Red Black a b
insB x E = R(E,x,E)
insB x t@(B(a,y,b))
  | x<y = balanceL (insR x a) y b
  | x>y = balanceR a y (insR x b)
  | otherwise = C t
```

```
insB :: Ord a => a -> Black a b -> Red Black a b
insB x E = R(E,x,E)
insB x t@(B(a,y,b))
  | x<y = balanceL (insR x a) y b
  | x>y = balanceR a y (insR x b)
  | otherwise = C t

insR :: Ord a => a -> Red Black a b -> RR a b
insR x (C t) = C(insB x t)
insR x t@(R(a,y,b))
  | x<y = R(insB x a,y,C b)
  | x>y = R(C a,y,insB x b)
  | otherwise = C t
```

```
tickB :: Black a b -> Black a c
tickB E = E
tickB (B(a,x,b)) = B(tickR a,x,tickR b)
```

```
tickB :: Black a b -> Black a c
tickB E = E
tickB (B(a,x,b)) = B(tickR a,x,tickR b)

tickR :: Red Black a b -> Red Black a c
tickR (C t) = C(tickB t)
tickR (R(a,x,b)) = R(tickB a,x,tickB b)
```

```
tickB :: Black a b -> Black a c
tickB E = E
tickB (B(a,x,b)) = B(tickR a,x,tickR b)

tickR :: Red Black a b -> Red Black a c
tickR (C t) = C(tickB t)
tickR (R(a,x,b)) = R(tickB a,x,tickB b)

inc :: Black a b -> Black a [b]
inc = tickB
```

```
ghci> (E :: Black a [[[b]]])
E
it :: Black a [[[b]]]
```

```
ghci> (E :: Black a [[[b]]])
E
it :: Black a [[[b]]]

newtype Tree a = forall b . ENC (Black a b)
```

```
ghci> (E :: Black a [[[b]]])
E
it :: Black a [[[b]]]

newtype Tree a = forall b . ENC (Black a b)

empty :: Tree a
empty = ENC E
```

```
ghci> (E :: Black a [[[b]]])
E
it :: Black a [[[b]]]

newtype Tree a = forall b . ENC (Black a b)

empty :: Tree a
empty = ENC E

insert :: Ord a => a -> Tree a -> Tree a
insert x (ENC t) = ENC(blacken (insB x t))
```

```
ghci> (E :: Black a [[[b]]])
E
it :: Black a [[[b]]]

newtype Tree a = forall b . ENC (Black a b)

empty :: Tree a
empty = ENC E

insert :: Ord a => a -> Tree a -> Tree a
insert x (ENC t) = ENC(blacken (insB x t))

blacken :: Red Black a b -> Black a b
blacken (C u) = u
blacken (R(a,x,b)) = B(C(inc a),x,C(inc b))
```

```
class Renderable a where
  boundingSphere :: a -> Sphere
  hit :: a -> [Fragment] -- returns the "fragments" of all hits with ray
```

```
class Renderable a where
  boundingSphere :: a -> Sphere
  hit :: a -> [Fragment] -- returns the "fragments" of all hits with ray

hits :: Renderable a => [a] -> [Fragment]
hits xs = sortByDistance $ concatMap hit xs
```

```haskell
class Renderable a where
  boundingSphere :: a -> Sphere
  hit :: a -> [Fragment] -- returns the "fragments" of all hits with ray

hits :: Renderable a => [a] -> [Fragment]
hits xs = sortByDistance $ concatMap hit xs

data AnyRenderable = forall a. Renderable a => AnyRenderable a
```

```
class Renderable a where
  boundingSphere :: a -> Sphere
  hit :: a -> [Fragment] -- returns the "fragments" of all hits with ray

hits :: Renderable a => [a] -> [Fragment]
hits xs = sortByDistance $ concatMap hit xs

data AnyRenderable = forall a. Renderable a => AnyRenderable a

instance Renderable AnyRenderable where
    boundingSphere (AnyRenderable a) = boundingSphere a
    hit (AnyRenderable a) = hit a
```

```
class Renderable a where
  boundingSphere :: a -> Sphere
  hit :: a -> [Fragment] -- returns the "fragments" of all hits with ray

hits :: Renderable a => [a] -> [Fragment]
hits xs = sortByDistance $ concatMap hit xs

data AnyRenderable = forall a. Renderable a => AnyRenderable a

instance Renderable AnyRenderable where
    boundingSphere (AnyRenderable a) = boundingSphere a
    hit (AnyRenderable a) = hit a

[ AnyRenderable x
, AnyRenderable y
, AnyRenderable z ]
```

References

- https://wiki.haskell.org/Phantom_type
- https://wiki.haskell.org/Generalised_algebraic_datatype
- https://wiki.haskell.org/Existential_type
- http://stackoverflow.com/questions/3071136/what-does-the-forall-keyword-in-haskell-ghc-do
- https://://themonadreader.files.wordpress.com/2010/01/issue15.pdf
- Red-black trees with types. Stefan Kahrs, 2001.
- "Adventures in Three Monads", Edward Z. Yang. The Monad Reader, issue 15.