Conor McBride: *The Derivative of a Regular Type is its Type of One-Hole Contexts*

Gergő Érdi

http://gergo.erdhi.hu/

Papers We Love.SG, June 2015.
Gérard Huet: *The Zipper* (Functional Pearl, 1997): functional equivalent of a pointer into a data structure: turn a tree-like structure into a subtree in context.

```hs
data Paren α = Leaf α
           | Branch (Paren α) (Paren α)

type ParenZ α = ([Either (Paren α) (Paren α)], Paren α)
```
Gérard Huet: *The Zipper* (Functional Pearl, 1997): functional equivalent of a pointer into a data structure: turn a tree-like structure into a subtree in context.

```haskell
data Paren α = Leaf α
             | Branch (Paren α) (Paren α)

type ParenZ α = ([Either (Paren α) (Paren α)], Paren α)

zip :: ParenZ α → Paren α
zip (path, t) = foldl plug t path

where
  plug t2 (Left t1) = Branch t1 t2
  plug t1 (Right t2) = Branch t1 t2
```
Introduction: Zippers

- Gérard Huet: *The Zipper* (Functional Pearl, 1997): functional equivalent of a pointer into a data structure: turn a tree-like structure into a subtree in context.

```plaintext
data Paren α = Leaf α 
  | Branch (Paren α) (Paren α)

type ParenZ α = ([Either (Paren α) (Paren α)], Paren α)

holes :: Paren α → [ParenZ α]
holes Leaf {} = []
holes (Branch t1 t2) = [([Left t1], t2), ([Right t2], t1)]
```
Is there a principled way of coming up with all this?
Is there a principled way of coming up with all this?

{-# LANGUAGE TypeFamilies, TypeOperators #-}
{-# LANGUAGE EmptyDataDecls, EmptyCase #-}
{-# LANGUAGE PatternSynonyms #-}

module ZipperDeriv where
import Prelude hiding (zip, unzip)
import Control.Arrow (first)
Sums of products

\[\text{newtype } Const \alpha = Const \{ \text{unConst :: } \alpha \}\]
\[\text{type 1 } = Const ()\]
\[\text{pattern 1 } = Const ()\]
\[\text{type 0 } = Const \text{ Void}\]
\[\text{infixl 6 } \oplus\]
\[\text{data (} \oplus \text{) } f \ g \quad = \text{InL } f \quad | \quad \text{InR } g\]
\[\text{infixl 7 } \otimes\]
\[\text{data (} \otimes \text{) } f \ g \quad = f \quad \otimes g\]
Sums of products

\textbf{newtype} $\texttt{Const} \ \alpha \ = \ \texttt{Const} \ \{ \ \texttt{unConst} :: \alpha \} \\
\textbf{type 1} = \texttt{Const} () \\
\textbf{pattern 1} = \texttt{Const} () \\
\textbf{type 0} = \texttt{Const} \ \texttt{Void} \\
\textbf{infixl} 6 \oplus \\
\textbf{data} \ (\oplus) \ f \ g \ = \texttt{InL} \ f \\
\ | \ \texttt{InR} \ g \\
\textbf{infixl} 7 \otimes \\
\textbf{data} \ (\otimes) \ f \ g \ = \ f \ \otimes \ g

Hey look, it’s a semiring! (modulo handwavy isomorphisms)
newtype Const α x = Const { unConst :: α }
type 1 = Const ()
pattern 1 = Const ()
type 0 = Const Void
infixl 6 ⊕
data (⊕) f g x = InL (f x) |
             InR (g x)

infixl 7 ⊗
data (⊗) f g x = f x ⊗ g x

newtype X x = X { unX :: x }

Hey look, it’s a semiring! (modulo handwavy isomorphisms)
So let’s build polynomials!
The original paper is formulated for polynomials, and fixed points, in many variables; but for simplicity’s sake, this presentation uses single-variable polynomials.

\[
\text{newtype } \mu f = \text{Fix } \{ \text{unFix :: } f (\mu f) \}
\]
The original paper is formulated for polynomials, and fixed points, in many variables; but for simplicity’s sake, this presentation uses single-variable polynomials.

\[
\textbf{newtype } \mu f = \text{Fix} \{ \text{unFix :: } f (\mu f) \}
\]

This corresponds to regular data types because you can only do wholesale induction in the datatype definition: the syntax already only allows for defining non-parametric data types.
Examples!

\[
\text{type } Puzzle1F \, \alpha = 1 \oplus (\text{Const } \alpha \otimes X)
\]

\[
\text{type } Puzzle1 \, \alpha = \mu (Puzzle1F \, \alpha)
\]
Examples!

\[
\text{type } ListF \; \alpha = \mathbf{1} \oplus (\text{Const } \alpha \otimes X)
\]

\[
\text{type } List \; \alpha = \mu (ListF \; \alpha)
\]

\[
\text{pattern } Nil \quad =
\]

\[
\text{pattern } Cons \; x \; xs \quad =
\]
Examples!

\[
\text{type } \operatorname{ListF} \alpha = 1 \oplus (\operatorname{Const} \alpha \otimes X) \\
\text{type } \operatorname{List} \alpha = \mu (\operatorname{ListF} \alpha)
\]

\[
\text{pattern } \operatorname{Nil} = \operatorname{Fix} (\operatorname{InL} 1) \\
\text{pattern } \operatorname{Cons} x \; \operatorname{xs} = \operatorname{Fix} (\operatorname{InR} (\operatorname{Const} x \otimes X \; \operatorname{xs}))
\]
Examples!

\[ \text{type } \text{ListF } \alpha = 1 \oplus (\text{Const } \alpha \otimes X) \]
\[ \text{type } \text{List } \alpha = \mu (\text{ListF } \alpha) \]

\[ \text{type } \text{Puzzle2F } \alpha = \text{Const } \alpha \oplus (X \otimes X) \]
\[ \text{type } \text{Puzzle2 } \alpha = \mu (\text{Puzzle2F } \alpha) \]
Examples!

\[
\text{type } ListF \ \alpha = 1 \oplus (\text{Const } \alpha \otimes X) \\
\text{type } List \ \alpha = \mu (ListF \ \alpha)
\]

\[
\text{type } ParenF \ \alpha = \text{Const } \alpha \oplus (X \otimes X) \\
\text{type } Paren \ \alpha = \mu (ParenF \ \alpha)
\]

\[
\text{pattern } \text{Leaf } x = \\
\text{pattern } \text{Pair } t1 \ t2 =
\]
Examples!

\[\text{type } \text{ListF } \alpha = 1 \oplus (\text{Const } \alpha \otimes X)\]
\[\text{type } \text{List } \alpha = \mu (\text{ListF } \alpha)\]

\[\text{type } \text{ParenF } \alpha = \text{Const } \alpha \oplus (X \otimes X)\]
\[\text{type } \text{Paren } \alpha = \mu (\text{ParenF } \alpha)\]

\[\text{pattern } \text{Leaf } x = \text{Fix} (\text{InL} (\text{Const } x))\]
\[\text{pattern } \text{Pair } t1 \ t2 = \text{Fix} (\text{InR} (X \ t1 \otimes X \ t2))\]
Examples!

\[
\text{type } ListF \, \alpha = 1 \oplus (\text{Const} \, \alpha \otimes X) \\
\text{type } List \, \alpha = \mu (ListF \, \alpha)
\]

\[
\text{type } ParenF \, \alpha = \text{Const} \, \alpha \oplus (X \otimes X) \\
\text{type } Paren \, \alpha = \mu (ParenF \, \alpha)
\]

\[
\text{type } Puzzle3F \, \alpha = 1 \oplus (\text{Const} \, \alpha \otimes X \otimes X) \\
\text{type } Puzzle3 \, \alpha = \mu (Puzzle3F \, \alpha)
\]
Examples!

\[
\begin{align*}
\text{type } & \text{ListF } \alpha = \mathbf{1} \oplus (\text{Const } \alpha \otimes X) \\
\text{type } & \text{List } \alpha = \mu (\text{ListF } \alpha)
\end{align*}
\]

\[
\begin{align*}
\text{type } & \text{ParenF } \alpha = \text{Const } \alpha \oplus (X \otimes X) \\
\text{type } & \text{Paren } \alpha = \mu (\text{ParenF } \alpha)
\end{align*}
\]

\[
\begin{align*}
\text{type } & \text{BTreeF } \alpha = \mathbf{1} \oplus (\text{Const } \alpha \otimes X \otimes X) \\
\text{type } & \text{BTree } \alpha = \mu (\text{BTreeF } \alpha)
\end{align*}
\]

\[
\begin{align*}
\text{pattern } & \text{Empty } = \\
\text{pattern } & \text{Node } x \ t1 \ t2 =
\end{align*}
\]
Examples!

\[
\text{type } \text{ListF} \ \alpha = 1 \oplus (\text{Const} \ \alpha \otimes X) \\
\text{type } \text{List} \ \alpha = \mu (\text{ListF} \ \alpha)
\]

\[
\text{type } \text{ParenF} \ \alpha = \text{Const} \ \alpha \oplus (X \otimes X) \\
\text{type } \text{Paren} \ \alpha = \mu (\text{ParenF} \ \alpha)
\]

\[
\text{type } \text{BTreeF} \ \alpha = 1 \oplus (\text{Const} \ \alpha \otimes X \otimes X) \\
\text{type } \text{BTree} \ \alpha = \mu (\text{BTreeF} \ \alpha)
\]

\[
\text{pattern } \text{Empty} = \text{Fix} (\text{InL} \ 1) \\
\text{pattern } \text{Node} \ x \ t1 \ t2 = \text{Fix} (\text{InR} (\text{Const} \ x \otimes X \ t1 \otimes X \ t2))
\]
What do we expect of a type for holes $\partial f$?

Without even defining what exactly we mean by a hole
What do we expect of a type for holes $\partial f$?

Without even defining what exactly we mean by a hole

- $\partial (f \oplus g)$: Either we have an $\text{InL}$ value with a hole, or an $\text{InR}$ value with a hole: $\partial f \oplus \partial g$

- $\partial (f \otimes g)$: Either we have a hole in the first component (and the second is untouched), or the hole is in the second component: $(\partial f \otimes g) \oplus (f \otimes \partial g)$
What do we expect of a type for holes $\partial f$?

We need to be more specific on what a hole is to continue. So let’s say we want to be able to

- Get all holes into which we can plug a subtree (of the same type as the original type)
- Plug a subtree into a hole

```haskell
class Diffable (f :: * → *) where
  type ∂ f :: * → *
  holes' :: f x → [(∂ f x, x)]
  plug' :: ∂ f x → x → f x
```
What do we expect of a type for holes $\partial f$?

We need to be more specific on what a hole is to continue. So let’s say we want to be able to

- Get all holes into which we can plug a subtree (of the same type as the original type)
- Plug a subtree into a hole

```haskell
class Diffable (f :: * -> *) where
  type ∂ f :: * -> *
  holes' :: f x -> [(∂ f x, x)]
  plug' :: ∂ f x -> x -> f x
```

This should give us an idea of what to do for $\mathsf{Const} \, \alpha$ and $X$:

- $\mathsf{Const} \, \alpha \, x$ has no possible positions (of type $x$): $\partial (\mathsf{Const} \, \alpha) = 0$
- $X \, x$ has exactly one position of type $x$ (no further labelling needed): $\partial X = 1$
Looks familiar?

\[
\begin{align*}
\text{type } \partial (\text{Const } \alpha) &= 0 \\
\text{type } \partial X &= 1 \\
\text{type } \partial (f \oplus g) &= \partial f \oplus \partial g \\
\text{type } \partial (f \otimes g) &= (\partial f \otimes g) \oplus (f \otimes \partial g)
\end{align*}
\]
type $\partial (Const \, \alpha) = 0$

$\partial X = 1$

$\partial (f \oplus g) = \partial f \oplus \partial g$

$\partial (f \otimes g) = (\partial f \otimes g) \oplus (f \otimes \partial g)$

Surprise! Turns out using the names Diffable and $\partial$ (and “Derivative” in the paper’s title) was a reasonable choice!
type \( \partial (\text{Const } \alpha) = 0 \)

\begin{align*}
\text{type } \partial X &= 1 \\
\text{type } \partial (f \oplus g) &= \partial f \oplus \partial g \\
\text{type } \partial (f \otimes g) &= (\partial f \otimes g) \oplus (f \otimes \partial g)
\end{align*}

Surprise! Turns out using the names \textit{Diffable} and \( \partial \) (and “Derivative” in the paper’s title) was a reasonable choice!

See the source code of the slides for the full implementation of the \textit{Diffable} typeclass for \textit{Const} \( \alpha \), \( X \), \( f \oplus g \) and \( f \otimes g \).
Making a zipper from holes

We now have a way of taking apart one level of an inductive data structure by having a type $\partial f$ which has a hole in it.
Making a zipper from holes

We now have a way of taking apart one level of an inductive data structure by having a type $\partial f$ which has a hole in it. We can turn this into a type $\text{Zipper } f$ (a zipper for $\mu f$) by repeatedly choosing a hole and putting either one more level of data structure into it, or finishing with a $\mu f$:

\begin{align*}
\text{type } & D f = \partial f (\mu f) \\
\text{holes } & :: (\text{Diffable } f) \Rightarrow \mu f \rightarrow [(D f, \mu f)] \\
\text{holes } & = \text{holes'} \circ \text{unFix} \\
\text{plug } & :: (\text{Diffable } f) \Rightarrow D f \rightarrow \mu f \rightarrow \mu f \\
\text{plug } df &= \text{Fix} \circ \text{plug}' df \\
\text{type } & \text{Zipper } f = ([D f], \mu f) \\
\text{zip } & :: (\text{Diffable } f) \Rightarrow \text{Zipper } f \rightarrow \mu f \\
\text{zip } (\text{path}, t) &= \text{foldl} \ (\text{flip plug}) \ t \ \text{path} \\
\text{unzip } & :: (\text{Diffable } f) \Rightarrow \text{Zipper } f \rightarrow [\text{Zipper } f] \\
\text{unzip } (dfs, t) &= \text{map} \ (\text{first} \ (\text{first} (dfs))) \ (\text{holes } t)
\end{align*}
Example: all zippers of a parenthesization

Let’s enumerate all possible zippers for a given container (by recursively calling \textit{unzip})�

\[
\text{unzips} :: (\text{Diffable } f) \Rightarrow \mu f \rightarrow [\text{Zipper } f]
\]
\[
\text{unzips } t = \text{go } ([], t)
\]
\[
\text{where}
\]
\[
\text{go } z = z : \text{concatMap } \text{go } (\text{unzip } z)
\]

We are going to use this to enumerate all zippers of this type:

\[
\text{type } \text{Paren } \alpha = \mu (\text{Const } \alpha \oplus (X \otimes X))
\]
\[
\text{pattern } \text{Leaf } x = \text{Fix } (\text{InL } (\text{Const } x))
\]
\[
\text{pattern } \text{Pair } t1 t2 = \text{Fix } (\text{InR } (X t1 \otimes X t2))
\]
Example: all zippers of a parenthesization

Replace with a marker every possible pointed subtree:

\[
\text{probe} :: \alpha \rightarrow \text{Paren} \alpha \rightarrow [\text{Paren} \alpha]
\]

\[
\text{probe marker} = \text{map} \text{ mark} \circ \text{unzips}
\]

where

\[
\text{mark} (z, t) = \text{zip} (z, \text{Leaf marker})
\]
An example parenthesization:

\[ p :: Paren \ (Maybe \ Char) \]
\[ p = (Leaf \ (Just \ 'A') \ 'Pair' \ Leaf \ (Just \ 'B')) \ 'Pair' \ (Leaf \ (Just \ 'C')) \]
Example: all zippers of a parenthesization

An example parenthesization:

\[
p :: Paren (Maybe Char) \\
p = (Leaf (Just 'A') 'Pair' Leaf (Just 'B')) 'Pair' (Leaf (Just 'C'))
\]