Zero to Zipper

Slavomir Kaslev
kaslevs@vmware.com

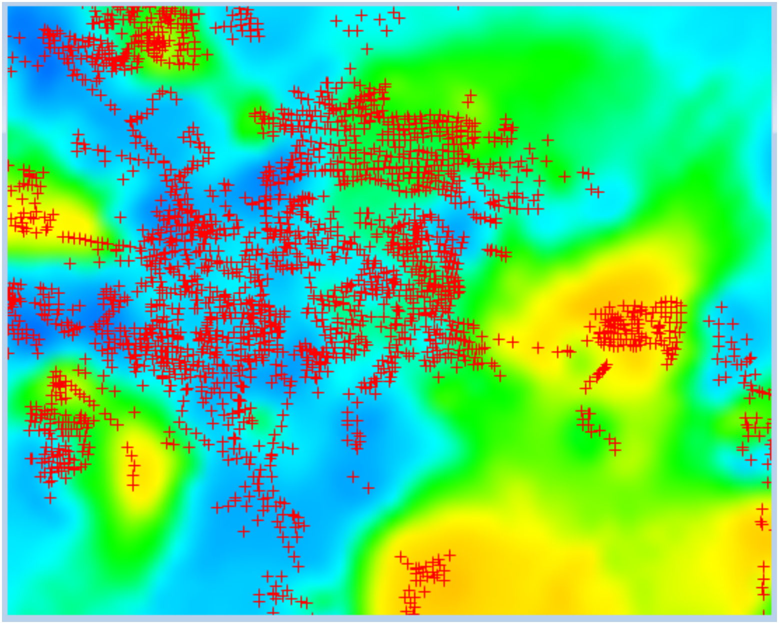
October 17, 2019

QOTD

- "The shortest path between two truths in the real domain passes through the complex domain." Jacques Hadamard

- "Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do." Michael R. Fellows and Ian Parberry

The Ocean of Programming Languages



Comparing Core Languages

C	Haskell	Lean
data Expr	data Expr	data Expr
data Type	data Type	
data Stmt		

C Core Language 1/3

```
data Expr
  = Comma       [Expr]
  | Assign      AssignOp Expr Expr
  | Cond        Expr (Maybe Expr) Expr
  | Binary      BinaryOp Expr Expr
  | Cast        Type Expr
  | Unary       UnaryOp Expr
  | SizeofExpr  Expr
  | SizeofType  Type
  | Index       Expr Expr
  | Call        Expr [Expr]
  | Member      Expr Ident Bool
  | Var         Ident
  | Const       Constant
  | CompoundLit Type InitializerList
```

C Core Language 2/3

```haskell
data Type
  = VoidType
  | BoolType
  | CharType
  | IntType
  | FloatType
  | DoubleType
  | ShortType   Type
  | LongType    Type
  | SignedType  Type
  | UnsigType   Type
  | SUType      StructureUnion
  | EnumType    Enumeration
  | FunPtr      Type [Type]
  | Ptr         Type
  | Arr         Type ArraySize
  | TypeDef     Ident
```

C Core Language 3/3

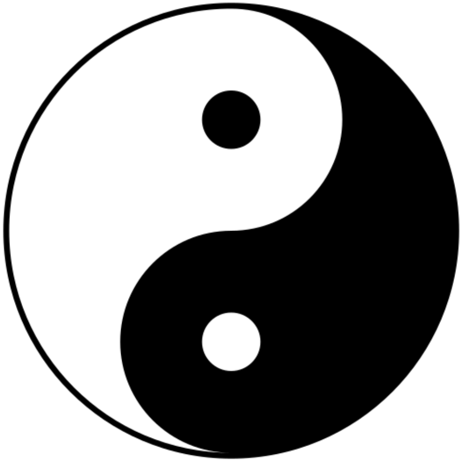
```
data Stmt
  = Label Ident Stmt [Attribute]
  | Case Expr Stmt
  | Default Stmt
  | Expr (Maybe Expr)
  | Compound [Ident] [CompoundBlockItem]
  | If Expr Stmt (Maybe Stmt)
  | Switch Expr Stmt
  | While Expr Stmt Bool
  | For (Either (Maybe Expr) Type)
        (Maybe Expr)
        (Maybe Expr)
        Stmt
  | Goto Ident
  | Cont
  | Break
  | Return (Maybe Expr)
```

Haskell Core Language

```haskell
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Cast  (Expr b) Coercion
  | Coercion Coercion

data Type
  = TyVarTy   Var
  | LitTy     TyLit
  | AppTy     Type Type
  | ForAllTy  !TyCoVarBinder Type
  | FunTy     Type Type
  | TyConApp  TyCon [KindOrType]
  | CastTy    Type KindCoercion
  | CoercionTy Coercion
```

The Duality of Code and Data



Lean Core Language

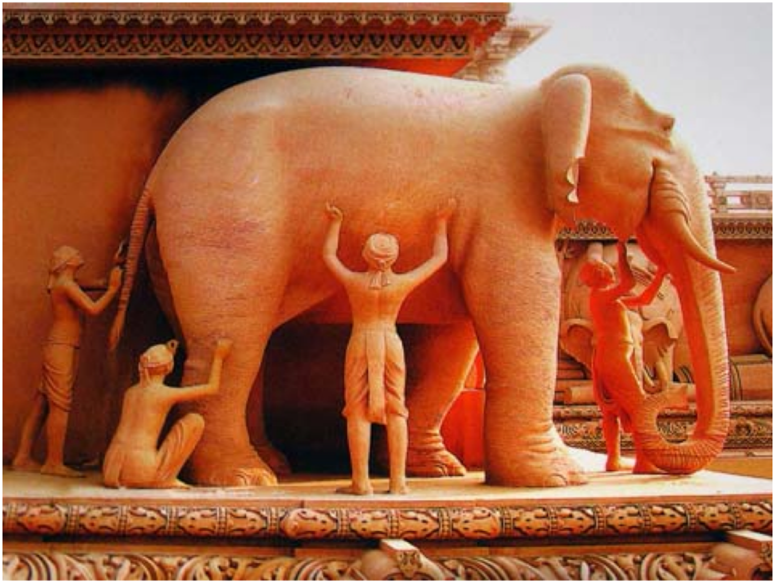
```
inductive expr
| var         : nat → expr
| sort        : level → expr
| const       : name → list level → expr
| mvar        : name → name → expr → expr
| local_const : name → name → binder_info → expr → expr
| app         : expr → expr → expr
| lam         : name → binder_info → expr → expr → expr
| pi          : name → binder_info → expr → expr → expr
| elet        : name → expr → expr → expr → expr
| macro       : macro_def → list expr → expr
```

The Curry-Howard Correspondence Extended[3]

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

[3] `http://saunders.phil.cmu.edu/book/hott-online.pdf#page=23`

The Elephant



Isomorphisms in Haskell

```haskell
data Iso a b = Iso { f :: a -> b, g :: b -> a }
-- Should satisfy the following laws:
--   ∀ x : a, g (f x) = x
--   ∀ x : b, f (g x) = x

inv :: Iso a b -> Iso b a
inv (Iso f g) = Iso g f

comp :: Iso a b -> Iso b c -> Iso a c
comp (Iso f1 g1) (Iso f2 g2) = Iso (f2 . f1) (g1 . g2)
```

Isomorphisms in Lean

```
structure iso (a b : Type) :=
(f : a → b) (g : b → a) (gf : Π x, g (f x) = x) (fg : Π x, f (g x) = x)

def inv {a b} (i : iso a b) : iso b a :=
⟨i.g, i.f, i.fg, i.gf⟩

def comp {a b c} (i : iso a b) (j : iso b c) : iso a c :=
⟨j.f ∘ i.f, i.g ∘ j.g, by simp [j.gf, i.gf], by simp [i.fg, j.fg]⟩
```

Type Sizes

- Suppose there's a size function for *finite* types $|\cdot| : \mathit{Type} \to \mathbb{N}$ and let's look at the sizes of the fundamental types namely

$$A \oplus B \qquad A \otimes B \qquad A \to B$$

- One can prove that

$$\begin{aligned} |A \oplus B| &= |A| + |B| \\ |A \otimes B| &= |A| \cdot |B| \\ |A \to B| &= |B|^{|A|} \end{aligned}$$

From Isomorphisms to Equations and Back

- Let $|A| = a$, $|B| = b$ and $|C| = c$
- Distributive Law

$$A \otimes (B \oplus C) = (A \otimes B) \oplus (A \otimes C)$$
$$a(b + c) = ab + ac$$

- Pattern matching on disjoint union

$$A \oplus B \to C = (A \to C) \otimes (B \to C)$$
$$c^{a+b} = c^a c^b$$

- Function currying

$$A \to B \to C = A \otimes B \to C$$
$$c^{b^a} = c^{ab}$$

Analytic Combinatorics

- Analytic combinatorics deals with counting combinatorial objects by means of their generating functions
- What is a generating function?
- Given a type A and a size function $|\cdot| : A \to \mathbb{N}$, A's ordinary generating function (OGF) is defined as

$$A(x) = \sum_{a:A} x^{|a|} = \sum_{n=0}^{\infty} a_n x^n$$

- The numbers a_n tell us how many objects in A are of size n

Symbolic Method: Finding generating functions

- Flajolet and Sedgewick propose a simple method of finding equation for the OGF of a given combinatorial construction expressed in their specification language
- In the special case of algebraic data types, the symbolic method uses the fact that if A, B, C are types and $A(x), B(x), C(x)$ are the corresponding OGFs then

$$C = A \oplus B \implies C(x) = A(x) + B(x)$$

and

$$C = A \otimes B \implies C(x) = A(x)B(x)$$

Symbolic Method: Examples 1/2

```
data Maybe x = None | Just x
```

$$M(x) = 1 + x$$

```
data List x = Nil | Cons x (List x)
```

$$L(x) = 1 + xL(x)$$

$$L(x) = \frac{1}{1-x}$$

$$L(x) = 1 + x + x^2 + x^3 + x^4 + x^5 + \dots$$

Symbolic Method: Examples 2/2

```haskell
data C x = Single x | Pair x x
type F x = [C x]
```

$$F(x) = \frac{1}{1 - x - x^2}$$

$$F(x) = 1 + x + 2x^2 + 3x^3 + 5x^4 + 8x^5 + 13x^6 + 21x^7 + \dots$$

```haskell
data BinTree x = Leaf | Branch x (BinTree x) (BinTree x)
```

$$B(x) = 1 + xB(x)^2$$

$$B(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

$$B(x) = 1 + x + 2x^2 + 5x^3 + 14x^4 + 42x^5 + 132x^6 + 429x^7 + \dots$$

Zipper

- Zipper of a data structure is another data structure that provides iteration and modification in $O(1)$ time complexity
- OGF for the zipper over any data structure with OGF $F(x)$ is defined as

$$Z_F(x) = x\frac{\partial}{\partial x}F(x)$$

- The derivative of an OGF $\frac{\partial}{\partial x}F(x)$ gives the OGF of the same structure with one hole in it, e.g.

$$\frac{\partial}{\partial x}x^n = nx^{n-1}$$

- The x in the right hand side is called focus and holds what was initially in that hole

List Zipper 1/2

```
data List x = Nil | Cons x (List x)
```

$$L(x) = 1 + xL(x)$$

$$L(x) = \frac{1}{1-x}$$

$$\frac{\partial}{\partial x} L(x) = \frac{1}{(1-x)^2} = L(x)^2$$

$$Z_L(x) = xL(x)^2$$

```
data ZList x = Focus x (List x) (List x)
```

List Zipper 2/2

```haskell
data ZList a = Focus a [a] [a]

toZipper :: [a] -> Maybe (ZList a)
toZipper [] = Nothing
toZipper (x:xs) = Just $ Focus x xs []

fromZipper :: ZList a -> [a]
fromZipper   (Focus x r []) = x:r
fromZipper z@(Focus x r (y:p)) = fromZipper $ left z

set :: ZList a -> a -> ZList a
set (Focus x r p) y = Focus y r p

left :: ZList a -> ZList a
left z@(Focus x r []) = z
left   (Focus x r (y:p)) = Focus y (x:r) p

right :: ZList a -> ZList a
right z@(Focus x [] p) = z
right   (Focus x (y:r) p) = Focus y r (x:p)

main = do
  let z  = fromJust $ toZipper [1,2,3,4,5]
      z1 = set (right $ right z) 42
      z2 = set (left z1) 0
  print $ fromZipper z2
-- prints [1,0,42,4,5]
```

Binary Tree Zipper 1/3

```
data BinTree x = Leaf | Branch x (BinTree x) (BinTree x)
```

$$B(x) = 1 + xB(x)^2$$

$$\frac{\partial}{\partial x}B(x) = B(x)^2 + 2xB(x)\frac{\partial}{\partial x}B(x)$$

$$\frac{\partial}{\partial x}B(x) = \frac{B(x)^2}{1 - 2xB(x)}$$

$$Z_B(x) = xB(x)^2\frac{1}{1 - 2xB(x)}$$

```
data Segment x = SLeft x (BinTree x) | SRight x (BinTree x)
data ZBinTree x = Focus x (BinTree x) (BinTree x) [Segment x]
```

Binary Tree Zipper 2/3

```
data BinTree a = Leaf | Branch a (BinTree a) (BinTree a)

data Segment a = SLeft a (BinTree a) | SRight a (BinTree a)
data ZBinTree a = Focus a (BinTree a) (BinTree a) [Segment a]

toZipper :: BinTree a -> Maybe (ZBinTree a)
toZipper Leaf = Nothing
toZipper (Branch x l r) = Just $ Focus x l r []

fromZipper :: ZBinTree a -> BinTree a
fromZipper   (Focus x l r []) = Branch x l r
fromZipper z@(Focus x l r (s:p)) = fromZipper $ up z

set :: ZBinTree a -> a -> ZBinTree a
set (Focus x l r p) y = Focus y l r p

left :: ZBinTree a -> ZBinTree a
left z@(Focus x Leaf r p) = z
left   (Focus x (Branch y ll lr) r p) = Focus y ll lr (SLeft x r:p)

right :: ZBinTree a -> ZBinTree a
right z@(Focus x l Leaf p) = z
right   (Focus x l (Branch y rl rr) p) = Focus y rl rr (SRight x l:p)

up :: ZBinTree a -> ZBinTree a
up z@(Focus x l r []) = z
up   (Focus x l r (SLeft y ur:p))  = Focus y (Branch x l r) ur p
up   (Focus x l r (SRight y ul:p)) = Focus y ul (Branch x l r) p
```

Binary Tree Zipper 3/3

```haskell
t :: BinTree Int
t = Branch 1 (Branch 2 Leaf (Branch 3 Leaf (Branch 4 Leaf Leaf)))
             (Branch 5 Leaf Leaf)

main = do
  let z  = fromJust $ toZipper t
      z1 = set (right $ left z) 42
      z2 = set (up z1) 0
  print $ fromZipper z2

-- prints
-- Branch 1 (Branch 0 Leaf (Branch 42 Leaf (Branch 4 Leaf Leaf)))
--          (Branch 5 Leaf Leaf)
```

Challenge

- Write a library that automatically derives the zipper and its operations for any given data structure

```
data RoseTree a = Node a [RoseTree a]
$(mkZipper ''RoseTree)
```

  - Bonus points if it also derives a Comonad instance

- To handle multiple type variables the zipper can be generalized

$$Z_F(\boldsymbol{x}) = \boldsymbol{x} \cdot \nabla F(\boldsymbol{x})$$

Further Reading

- "Functional Pearl: The Zipper" by Gérard Huet
- "The Derivative of a Regular Type is its Type of One-Hole Contexts" by Conor McBride
- "Analytic Combinatorics" course on Coursera
  `https://www.coursera.org/learn/analytic-combinatorics`
- "An Introduction to the Analysis of Algorithms" by Philippe Flajolet and Robert Sedgewick
- "Analytic Combinatorics" by Philippe Flajolet and Robert Sedgewick
- "Homotopy Type Theory: Univalent Foundations of Mathematics" by The Univalent Foundations Program
- "Constructive Mathematics and Computer Programming" by Per Martin-Löf
- Tangent bundle
  `https://en.wikipedia.org/wiki/Tangent_bundle`
- Code and slides from this talk
  `https://github.com/skaslev/zero-to-zipper`

Thank you