Dissection of Datatypes for Generic Traversals

Kevin Ullmann (kru3@cornell.edu)

May 20, 2011

1 Introduction

The motivation for this paper is to bring together various works regarding traversals over recursive datatypes in order to offer a more cohesive presentation on the methods for achieving both genericity in their construction and fine control during their execution. We present a synthesis of four papers by three authors, Gerard Huet, Jeremy Gibbons, and Conor McBride, to demonstrate the usefulness and elegance of their approaches for implementing and making use datatype traversals. We begin by exploring a structure that allows us to represent an in progress traversal of a tree, and to move freely about that tree in any direction from our current location. This structure is Huet's Zipper [2]. We will then look at strategies for traversing generic datatypes, namely the fold operation and origami programming. We will find that these techniques, though powerful, are still lacking, in that they provide no ability to manipulate an in progress traversal. This fact will motivate us to explore the derivative, which we will see partially achieves our goal by granting us the ability to construct a one-hole context of arbitrary types. This derivative type captures the notion of a frozen-in-place traversal. Unfortunately it does not allow us to move around the structure in consideration. We will then conclude with a discussion of dissection, a powerful method that not only allows for generic calculations of the type of an in-progress traversal, but also allows us to manipulate the original data structure and modify the effects of the traversal in real time. Dissection will provide us with first class access to the layers of a stack representing the traversal, allowing us to move through recursive datatypes easily, while maintaining fine control over their progress and effects.

2 The Zipper

We begin by presenting a type-specific example, which will motivate the development of more generic methods. A programmer often would like the ability to walk over a tree structure, modifying it along the way, without having to recopy the entire structure with each edit, nor having to retrace steps to reach a previously visited location. Huet's zipper structure solves this problem by providing a representation of a tree and a subtree at the current location of a traversal, allowing for movement from this location through the tree in any direction [2].

We will consider a tree structure analogous to the structure examined by Huet in his paper on the zipper [2]. Our tree has items of type Obj only at the leaves, and each node has a variable number of children, represented as a list of subtrees.

data `tree` = `Leaf Obj | Section [tree]`

The type of a path representing an in progress traversal is

data `path` = `Top | Node [tree]` $*$ `path` $*$ `[tree]`

where a $Node(l, p, r)$ represents a traversal with its focus on a particular location in a tree, along with l a list of that location's elder siblings, r its younger siblings, and p its father path [2]. We can now define a type representing the location itself.

data `location = Loc tree * path`

See [2] for concrete examples of these types.

2.1 Moving Through the Tree

Having defined the structure of traversals, we can define primitives that allow us to move in any direction around the tree. With the following implementations, each primitive, $go\_left, go\_right, go\_down$ takes constant time, and *go_up* takes time proportional to the length of $left$ [2]. We will present all examples in Haskell.

```
go_left :: location → location
go_left (Loc t Top) = error ``left of top''
go_left (Loc t (Node (l:left) up right)) = Loc l (Node left up (t:right))
go_left (Loc t (Node [] up right)) = error ``left of first''

go_right :: location → location
go_right (Loc t Top) = error ``right of top''
go_right (Loc t (Node left up (r:right)) = Loc r (Node (t:left) up right)
go_right _ = error ``right of last''

go_up :: location → location
go_up (Loc t Top) = error ``up of top''
go_up (Loc t (Node left up right)) = Loc (Section (rev left)++(t:right)) up

go_down :: location → location
go_down (Loc (Leaf _) p) = error ``down of leaf''
go_down (Loc (Section t1:trees) p) = Loc t1 (Node [] p trees)
go_down _ = error ``down of empty''
```

Huet also gives an example function which makes use of these primitives [2]. His is example is the following function, which computes the n^{th} child of the current tree.

```
nth :: location → int → location
nth l 1 = go_down(l)
nth l n = if n > 0 then go_right(nth l (n-1))
          else error ``nth expects a positive integer''
```

2.2 Mutating the Tree

Huet describes simple functions which make use of the zipper structure to cheaply and elegantly mutate the tree. Here are his implementations of the insertion operation [2]. The insertion functions insert the given tree in in a specific direction with regard to the current location, and leave the focus at that location after the insertion is performed, except for *insert_down*, which moves the focus to the tree that was just inserted.

```
insert_right :: location → tree → location
insert_right (Loc t Top) r = error ``insert of top''
insert_right (Loc t (Node left up right)) = Loc t (Node left up (r:right))

insert_left :: location → tree → location
insert_left (Loc t Top) l = error ``insert of top''
insert_left (Loc t (Node left up right)) = Loc t (Node (l:left) up right)

insert_down :: location → tree → location
```

```
insert_down (Loc (Leaf _) p) t1 = error ‘‘down of leaf’’
insert_down (Loc (Section sons) p) t1 = Loc t1 (Node [] p sons)
```

He further provides an implementation for the deletion operation. His implementation moves to the right if possible, otherwise left, and up if the current subtree has no siblings [2].

```
delete :: location → location
delete (Loc t Top) = error ‘‘delete of top’’
delete (Loc t (Node left up r:right)) = Loc r Node(left up right)
delete (Loc t (Node l:left up [])) = Loc l Node(left up [])
delete (Loc t (Node [] up [])) = Loc (Section []) up
```

These functions give us the ability to traverse a tree in arbitrary directions. What's more, we can modify the tree at any position that we travel to. In effect, Huet's zipper promotes the in progress traversal to a first class object which we have direct control over. We would like to define traversals like this in a generic way, to apply to any recursive structure, without having to implement functions like *go_up* for each datatype that we may wish to traverse. We would also like to maintain the fine control that is afforded us by first class access to in progress traversals.

3 Genericity

Having seen the usefulness of a readily accessible traversal for trees, we ask "Can we define a generic structure to represent traversals of arbitrary containers?" But before finding such a structure, we must be clear about what it means for a structure to be generic.

3.1 Classification of Types

In a paper exploring genericity entitled "Datatype-Generic Programming", Gibbons classifies various models for generic programs based on the parameters abstracted out of their implementations. He separates classes of genericity into those that are parameterized by value, by type, by function, by structure, by property, by stage, and by shape [1]. A brief summary of these characterizations is given here.

- Genericity by value refers to typical functions that are not normally considered to be generic at all. Take, for example, the function that raises x to the power y. This is a generic version of a function that simply squares x, since the exponent has been abstracted from the value 2 to an arbitrary value.
- Genericity by type is equivalent to what is more commonly referred to by the term polymorphism, for example an append function written for lists of arbitrary type $\boldsymbol{\alpha}$ **list**.
- Genericity by function is analogous to genericity by value except that now the parameterized values are actually functions. The map function is a perfect example of this type of genericity. Gibbons notes that genericity by function should be treated separately from genericity by value because "Among other things, it lets programmers express control structures within the language, rather than having to extend the language."[1] This concept of finer control will be an important motivation in forming the dissection operator described later in this paper.
- Genericity by structure by structure and by property are related to passing in encapsulated modules that may have hidden implementations of public interfaces. Examples of this type of genericity are functions like *sort* which operate over lists containing elements that are required to have the property of comparability.
- Genericity by stage refers to programs which construct or manipulate other programs [1], while genericity by shape "is to parametrization by type as 'by function' is to 'by value'." In other words, shape is just a higher order feature of a type, just as functions are simply higher order values.

For more precise definitions and further examples of these classifications of genericity in programs, see Gibbons paper on the subject [1].

It will be useful to explore an example of genericity by property. One such example is a function which requires its arguments to belong to the type class *Functor*, that is, they have the property of implementing a function $fmap$.

```
class Functor f where
fmap :: (a → b) → (f a → f b)
```

Note that we can easily make the *List* type class functorial:

```
instance Functor List where
fmap = mapL
```

where *mapL* is the standard map function for lists. We require that $fmap$ has the following nice properties:

$$\begin{aligned} fmap\ (f \circ g) &= fmap\ f \circ fmap\ g \\ fmap\ id &= id \end{aligned}$$

3.2 Origami

Let's take a look at how to exploit genericity to implement traversals. One group of approaches to this problem is known as "origami programming." The term refers to designing each program to reflect the structure of the datatype that it traverses [1], in order to take advantage of that structure. We have mentioned programs like this already, namely the $fmap$ operation. Note that each instance of $fmap$ is tailored to the structure being traversed, for example we defined the function for lists, but we can also define $fmap$ on binary trees, or trees of any arity. There are other, related operations, that also give us the ability to manipulate and traverse generic containers. These other operations include $fold$, $unfold$ and *build*; in fact we will see that *map* is actually a special case of $fold$. These origami operations can be used to perform what Gibbons refers to as "datatype-generic patterns of computation," or more colloquially, "origami programming."

3.2.1 Origami Examples on Lists

Here we present some of Gibbons examples of origami programming on lists. We begin by explicitly defining the types we are working with, and the $foldL$ function:

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

foldL :: b → (a → b → b) → List a → b
foldL e f Nil         = e
foldL e f (Cons x xs) = f x (foldL e f xs)
```

The fold operation traverses a datatype, in our case a list containing elements of arbitrary type, and accumulates these elements using the provided function into a single value. Take, for example, the following function, which calculates the sum of all elements.

```
sumL :: List Int → Int
sumL l = foldL 0 (λaλb a + b) l
```

The value accumulated by a fold can itself be a type with a higher order structure, for example another list. With this notion in mind we can readily implement *mapL* from $foldL$:

```
mapL :: (a → b) → List a → List b
mapL f l = foldL Nil (λaλb Cons (f a) b) l
```

Instead of collapsing a data structure into a single value, we can also begin with a value and grow a structure. Such an operation is an unfold, and it can be thought of as a dual operation to fold. Here is Gibbons implementation of $unfoldL$, which takes a value and creates a list of values.

```
unfoldL :: (b → Bool) → (b → a) → ( b → b) → b → List a
unfoldL p f g x =
        if (p x) then Nil
                 else Cons (f x) (unfoldL p f g (g x))
```

While $fold$ is familiar to functional programmers, $unfoldL$ may seem like a strange operation. However, it is quite powerful. The following function, again from Gibbons, takes an integer and returns its predecessors.

```
preds :: Int → List Int
preds = unfoldL (0 ==) id pred where pred n = n -1
```

Combining an unfold with a fold allows for very concise definitions of powerful higher-order functions. An operation which performs an unfold that produces a list to be immediately consumed by a subsequent fold is called a hylomorphism. A hylomorphism operator for lists can be defined as follows.

```
hyloL :: (b → Bool) → (b → a) → (b → b) → c → (a → c → c) → b → c
hyloL p f g e h = (foldL e h) ∘ (unfoldL p f g)
```

Now we can write the factorial function in very compact form:

```
fact :: Int → Int
fact = hyloL (0 ==) id pred 1 (∗)
```

Gibbons notes that with lazy evaluation the intermediate data structure is not computed all at once, which suggests a better implementation for $hyloL$ that does away with the intermediate structure altogether.

```
hyloL :: ( b→ Bool) → (b → a) → (b → b) → c → ( a → c → c) → b → c
hyloL p f g e h x =
        if p x then e else h (f x) (hyloL p f g e h (g x))
```

For a more detailed discussion of these operations and of the Origami paradigms, see [1].

4 Derivatives of Types

The fold and unfold operations we have seen provide us with convenient methods for computation over generic structures. Using origami programming, we can traverse structures of arbitrary type. Unfortunately, these operations don't allow the programmer first class access to the control structure itself. That is to say, the programmer has no way to affect a fold in progress, other than to include any additional computations or side effects within the implementation of the fold itself. But what does an in progress traversal look like? In this section, we develop a method described by McBride in what may be his most influential work, "The Derivative of a Regular Type is its Type of One-Hole Contexts" [3]. This method is the calculation of partial derivatives of regular types, and provides a generic computation on a type description to construct the type of its one-hole contexts. We will see that an inhabitant of this derivative type is analogous to a frozen traversal over the original structure.

4.1 The Regular Types

Before we can begin taking partial derivatives of regular types, we must define the regular types themselves. They are the typical types defined by sums and products. We will also require a definition for the set of type variables. We will follow McBride's definition of the infinite set $Name$, which can be thought of as the set

of all strings, and the set $NmSeq$ which is a finite set of elements from $Name$ [3]. If we have $\Sigma \in NmSeq$ we can define the set of regular types over Σ, denoted $Reg\ \Sigma$, inductively as follows [3].

$$\frac{\Sigma \in NmSeq}{Reg\ \Sigma \in Set} \qquad \frac{x \in \Sigma}{x \in Reg\ \Sigma}$$

$$\frac{}{0 \in Reg\ \Sigma} \qquad \frac{}{1 \in Reg\ \Sigma} \qquad \frac{S, T \in Reg\ \Sigma}{S + T \in Reg\ \Sigma} \qquad \frac{S, T \in Reg\ \Sigma}{S \times T \in Reg\ \Sigma}$$

$$\frac{F \in Reg\ \Sigma; x}{\mu x.F \in Reg\Sigma} \qquad \frac{F \in Reg\ \Sigma; x \quad S \in Reg\ \Sigma}{F|x{=}S \in Reg\ \Sigma} \qquad \frac{T \in Reg\ \Sigma}{\underline{\mathrm{T}}_x \in Reg\ \Sigma; x}$$

We can think of the type 0 as being the empty type $\emptyset$, and 1 being the unit type $()$. The type $\mu x.F$ is a recursive type where the name x may appear in F. $F|x = S$ is simply the type described by F where all instances of x are assigned the type S. McBride calls this constructor "definition" and notes that it takes the place of substitution [3]. Finally, $\underline{\mathrm{T}}_x$ is the weakening constructor, which allows a regular type to be lifted into a larger parent set.

McBride shows us how to recover the type $Bool$ using our construction [3].

$$\begin{aligned} Bool &\equiv 1 + 1 \\ true &\equiv inl\ () \\ false &\equiv inr\ () \end{aligned}$$

4.2 Partial Differentiation

Having defined the regular types, we can now define the partial differentiation operation that will give us the type of their one-hole contexts. Let's jump right into the rules for calculating this type.

$$\frac{x \in \Sigma \quad T \in Reg\ \Sigma}{\partial_x T \in Reg\ \Sigma}$$

$$\begin{aligned}
\partial_x x &= 1 && (1)\\
\partial_x (y \backslash x) &= 0 \quad (\text{where } y\backslash x \text{ means any } y \text{ except } x) && (2)\\
\partial_x 0 &= 0 && (3)\\
\partial_x (S + T) &= \partial_x S + \partial_x T && (4)\\
\partial_x 1 &= 0 && (5)\\
\partial_x (S \times T) &= \partial_x S \times T + S \times \partial_x T && (6)\\
\partial_x (\mu y.F) &= \mu z.\underline{\partial_x F|y{=}\mu y.F}_z + \underline{\partial_y F|y{=}\mu y.F}_z \times z && (7)\\
\partial_x (F|y{=}S) &= \partial_x F|y{=}S + \partial_y F|y{=}S \times \partial_x S && (8)\\
\partial_x \underline{\mathrm{T}}_x &= 0 && (9)\\
\partial_x \underline{\mathrm{T}}_{y\backslash x} &= \underline{\partial_x T}_y && (10)
\end{aligned}$$

Many of these rules follow the very same rules for partial differentiation that one learns in any introductory calculus course. Rule 1 tells us there is only one place to find an x in an x, while rules 2,3,5, and 9 tell us there are no x's in: (Rule 2) a y that is not x, (Rule 3) the empty type, (Rule 5) unit, and (Rule 9) a type T in a context where there are no x's. Rule 4 tells us that if we are looking for an x in an $S + T$ we can find it in either the S or the T, while rule 6 tells us that if we are looking for an x in an $S \times T$, we again find it in one of them, but this time we must remember the other side to maintain the structure of the original type.

Rules 7 and 8 are analogous to the "chain rule" in calculus. Rule 7 gives us the one whole context of a recursive type. It tells us that we either find an x at the top most node in the recursive type, or we must "unroll" the type and look again, remembering the last level that we passed by. The rule for substitution, rule 8, is quite similar. We either find an x in F, or we find an x in an S that occupied a y hole in F [3].

Let's see the derivative at work. Consider the following type of binary tree.

```
data btree = Leaf | Node btree btree
```

We can encode this type as the regular (recursive) type $\mu x.1 + x \times x$. What would a one-hole context of this type be? We can either find a subtree on the left or the right, or if we are at a leaf, there is no hole for a subtree. Thus we expect the type of the one-hole context (for a subtree) to be the choice of left or right, along with the tree that we passed along. We can represent this choice by an instance of the *Bool* type, which we have seen can be defined as $1 + 1$, so the type of one-hole contexts for *btree* would be $(1 + 1) \times btree \cong 2 \times btree$. Now let's calculate the derivative and see what we get.

$$\begin{aligned}
\partial_x(\mu x.1 + x \times x) &= \mu z.\underline{\partial_x(1 + x \times x)|x{=}\mu x.(1 + x \times x)}_z \\
&\quad + \underline{\partial_x(1 + x \times x)|x{=}\mu x.1 + x \times x}_z \times z \\
\partial_x(1 + x \times x)|x{=}\mu x.1 + x \times x &= 0 + (1 \times x + x \times 1)|x{=}\mu x.1 + x \times x \\
\partial_x(\mu x.1 + x \times x) &= \mu z.\underline{0 + (1 \times x + x \times 1)|x{=}\mu x.(1 + x \times x)}_z \\
&\quad + \underline{0 + (1 \times x + x \times 1)|x{=}\mu x.1 + x \times x}_z \times z \\
&\cong \mu z.(\underline{0}_z + \underline{2 \times x}_z + \underline{2 \times x}_z \times z)|x{=}\mu x.1 + x \times x \\
&\cong \textbf{list}\ 2 \times btree
\end{aligned}$$

Although this is a rather tedious calculation, we arrive at a wonderful result. The type of a one-hole context for *btrees* is a (potentially empty) list of choices of left or right, along with the tree passed by at each step down the tree! This structure is entirely analogous to the zipper that we've already seen, except that it was calculated generically from the type definition of the tree!

McBride notes another interesting result. If we write T^n for the n-fold product of T's and n as the sum of n 1's, then by induction on n we can show $\partial_x x^n \cong n \times x^{n-1}$, the same result we get from traditional calculus [3]. This result tells us that the one-hole context for x^n is an integer telling us which x the hole is at, along with the other $n - 1$ x's [3].

4.3 Plugging In

Given a one-hole context of some type T, $\partial_x T$, and an x, we should be able to plug the x into the hole to recover a T. Using McBride's notation, we are looking for the operation $\{T \lessdot x\}$, such that if we have a c of type T and a u of type x, then we can plug u into c, written infix as $c\ \{T \lessdot x\}\ u$, to recover a T. The bracket notation tells us the types involved in the plugging in operation. McBride gives the following rules

for plugging in [3]:

$$
\begin{array}{ll}
1\ \{x \lessdot x\}\ u & = u \\
\mathbf{inl}\ c\ \{S + T \lessdot x\}\ u & = \mathbf{inl}\ (c\ \{S \lessdot x\}\ u) \\
\mathbf{inr}\ c\ \{S + T \lessdot x\}\ u & = \mathbf{inr}\ (c\{T\ \lessdot x\}\ u) \\
\mathbf{inl}\ (c,t)\ \{S \times T \lessdot x\}\ u & = (c\{S\ \lessdot x\}\ u, t) \\
\mathbf{inr}\ (s,c)\ \{S \times T \lessdot x\}\ u & = (s, c\ \{T \lessdot x\}\ u) \\
\mathbf{con}\ (\mathbf{inl}\ c)\ \{\mu y.F \lessdot x\}\ u & = \mathbf{con}\ (c\{T \lessdot x\}u) \\
\mathbf{con}\ (\mathbf{inr}\ (c,cs))\ \{\mu y.F \lessdot x\}\ u & = \mathbf{con}\ \ (c\ \{F \lessdot y\}\ (cs\ \{\mu y.F \lessdot x\}\ u)) \\
\mathbf{inl}\ c\ \{F|y{=}S \lessdot x\}\ u & = c\ \{F \lessdot x\}\ u \\
\mathbf{inr}\ (c_y, c_x)\ \{F|y{=}S \lessdot x\}\ u & = c_y\ \{F \lessdot y\}\ (c_x\{S \lessdot x\}u) \\
c\ \{\underline{\mathrm{T}}_{y \setminus x} \lessdot x\}\ u & = c\ \{T \lessdot x\}\ u
\end{array}
$$

We have seen that c contains path information to the hole as well as the rest of the structure. Obviously, when $c = 0$ we have no where to put an x, so we omit this case.

5 Dissection

The derivative gave us a powerful method for computing the types of one-hole contexts of arbitrary types, generically from their type descriptions. We saw, by the example on *btree*, that an inhabitant of this type can be thought of as a snapshot of an in-progress traversal over the base type. We developed a method for taking a one-hole context and filling it to recover a structure of the original type, but had no ability to traverse the structures which we were creating. We now move on to an even more powerful operator than the derivative that builds on its usefulness to give us generic traversals over datatypes with first class access to each level of the traversal. This operation is known as dissection. Following McBride's approach in his paper on the topic [4], we will present the dissection of polynomials (the regular types), and will see that the operation further generalizes the notion of the derivative. From this operator we will be able not only to recover one-hole contexts from containers, but also to move through the containers, maintaining access to both the elements we have already passed over as well as to the elements we have yet to traverse.

5.1 Polynomial Functors

We will need to express the regular types from the previous section as functors. First let's define some type constructors. The following definitions are all taken from McBride [3].

```
data K₁ a x = K₁ a
data Id x = Id x
data (p +₁ q) x = L₁ (p x) | R₁ (q x)
data (p ×₁ q) x = (p x,₁ q x)
```

These constructors give us constants, single elements, sums, and products, respectively. We define the unit type to be

```
type 1₁ = K₁ ()
```

and the empty type simply as

```
data Zero
```

The subscripts indicate that these are different constructors from the bifunctors that we will see soon. The polynomial constructors we have just defined are all functorial, since we can define the requisite $fmap$ function for each instance as follows:

```
instance Functor (K₁ a) where fmap f (K₁ a) = K₁ a
instance Functor Id where fmap f (Id s) = Id (f s)
instance (Functor p, Functor q) ⇒ Functor (p +₁ q) where
        fmap f (L₁ p) = L₁ (fmap f p)
        fmap f (R₁ q) = R₁ (fmap f q)
instance (Functor p, Functor q) ⇒ Functor (p ×₁ q) where
        fmap f (p,₁ q) = (fmap f p,₁ fmap f q)
```

These definitions are sufficient to ensure that any polynomial built out of the given constructors is automatically functorial as well.

Before moving on to the polynomial bifunctors, which will be necessary for distinguishing elements that have been passed over from those which are yet to come, let's consider an example datatype that can be expressed as a fixpoint of the polynomial functors. The following datatype can be used to represent arbitrary logical propositions.

```
data prop = true | false | Not prop | Or prop prop
```

We can express this datatype using the μ notation from the previous section, but instead we will use our polynomial functors. We construct the datatype in a similar manner to McBride's construction of the *Expr* type [4]. First we define a type *PropP* that is a container for the immediate subterms of the proposition.

```
data PropP = K₁ Bool +₁ (Id +₁ Id × Id)
ValP b = L₁ (K₁ b)
NotP p = R₁ (L₁ (Id p))
OrP p₁ p₂ = R₁ (R₁ (Id p₁ ,₁ Id p₂))
```

We then use the μ constructor to tie the recursive not, giving us the true type for *prop*

```
data Prop = μ PropP
Val b = In (ValP b)
Not p = In (NotP p)
Or p₁ p₂ = In (OrP p₁ p₂)
```

where the μ constructor is a recursive type defined as

```
data μ p = In (p (μ p))
```

Again following McBride's *Expr* example [4], we can build a machine to evaluate the truth assignment of a given proposition. Our machine will traverse the structure defined by *Prop*, at each step considering either an expression to decompose, or a value to use.

```
eval :: Prop → Bool
eval p = load p []

load :: Prop → Stack → Bool
load (Val b) stk = unload b stk
load (Not p) stk = load p (L (L 1) : stk)
load (Or p₁ p₂) stk = load p₁ (L (R p₂) : stk)

unload :: Bool → Stack → Bool
unload b [] = b
unload b (L (L 1) : stk) = unload (-b) stk
unload b (L (R p₂) : stk) = load p₂ (R b : stk)
unload b₁ (R b₂ : stk) = unload (b1 || b2) stk
```

Each layer of the stack is a dissection of a *Prop*. The layer can be one of three things. It can be an $R\ b$, in which case we are to the right of a value, an $L\ (L\ 1)$ which means that we are exploring the proposition

inside of a Not, or an $L\ (R\ p)$, which means that we went to the left of an Or, which had a proposition p on the right. We will now see how to construct this sort of traversal for generic types.

5.2 Polynomial Bifunctors

The polynomial bifunctors are analogous to the polynomial functors, except they have two parameters instead of one.

```
data K₂ a x y = K₂ a
data Fst x y = Fst x
data Snd x y = Snd y
data (p +₂ q) x y = L₂ (p x y) | R₂ (q x y)
data (p ×₂ q) x y = (p x y,₂ q x y)
type 1₂ = K₂ ()
type 0₂ = K₂ Zero
```

Now instead of $fmap$, we have *bimap*, which requires one function for each parameter in the bifunctor.

```
class Bifunctor p where bimap :: (s₁ → t₁) → (s₂ → t₂) → p s₁ s₂ → p t₁ t₂
instance Bifunctor (K₂ a) where bimap f g (K₂ a) = K₂ a
instance Bifunctor Fst where bimap f g (Fst x) = Fst (f x)
instance Bifunctor Snd where bimap f g (Snd y) = Snd (g y)
instance (Bifunctor p, Bifunctor q) ⇒ Bifunctor (p +₂ q) where
        bimap f g (L₂ p) = L₂ (bimap f g p)
        bimap f g (R₂ q) = R₂ (bimap f g q)
instance (Bifunctor p, Bifunctor q) ⇒ Bifuntor (p ×₂ q) where
        bimap f g (p,₂ q) = (bimap f g p,₂ bimap f g q)
```

The polynomial functors are containers of a single type of element, while the polynomial bifunctors contain two types of elements. McBride refers to these as "clowns" and "jokers" respectively [4], but we shall refer to them as "formers" and "latters", since in an in order traversal all of the "formers" come before the current location, while the "latters" have yet to be inspected. McBride defines two operators for raising functors to bifunctors. We include them here for completeness but will not dwell on their implementation. The first, which McBride calls 'all clowns' [4] lifts a functor to a bifunctor whose elements are all formers, and the second lifts a functor to a bifunctor with all latters. These are defined as follows:

```
data Fp c j = F(p c)
instance Functor f ⇒ Bifunctor Ff where bimap f g (F pc) = F(fmap f pc)

data Tp c j = T(p j)
instance Functor f ⇒ Bifunctor (Tf) where bimap f g (T pj) = T(fmap g pj)
```

We have that $FId \cong Fst$ and $TId \cong Snd$.

The dissection of a functor p, Δp, lifts p to the bifunctor which represents both a position in p and the formers and latters with regard to this position. We can define Δp case by case. Constants have no positions for elements of any type, and so there is no way to dissect them [4].

$\Delta(\mathtt{K_1\ a}) = \mathtt{0_2}$

The Id functor can store only one element at one position.

$\Delta\mathtt{Id} = \mathtt{1_2}$

Dissecting a sum $p +_1 q$ gives us either a dissected p or dissected q [4].

$\Delta(\mathtt{p} +_1 \mathtt{q}) = \Delta\mathtt{p} +_2 \Delta\mathtt{q}$

Dissecting a pair gives us either a dissected left component with all latters to the right, or a dissected right component with all formers to the left [4].

$\Delta(\texttt{p} \times_1 \texttt{q}) = \Delta\texttt{p} \times_2 \texttt{Tq} +_2 \texttt{Fp} \times_2 \Delta\texttt{q}$

Let's dissect our proposition type:

$$\begin{aligned}\Delta(PropP) &= \Delta(K_1\ Bool +_1 (Id +_1 Id \times Id)) \\ &= 0_2 +_2 1_2 +_2 1_2 \times TId +_2 FId \times 1_2\end{aligned}$$

We see that applying this bifunctor to $Bool$ and $Prop$ gives us

$$\Delta(PropP)\ Bool\ Prop \cong unit + Prop + Bool$$

thus this expression tells us that dissecting a $PropP$ gives us either a unit, a bool, or a proposition. We have recovered the exact type of the stack layers in the evaluation machine!

5.3 Moving Around a Dissection

So far, the dissection operator doesn't seem much more powerful than the derivative operator from earlier. However, unlike with the derivative, we can easily define operations that use dissections to move throughout a datatype. Consider the function *right* which takes either a container or dissected container, and moves one position to the right. The function would have the signature

```
right :: p j + (Δp c j, c) → (j, Δp c j) + p c
```

This signature represents the different situations we can encounter when attempting to move right. We can either start all the way at left of the structure, in which case the elements are all latters, or we are already in the middle of a traversal. If we are in the middle of a traversal, then we we have a dissected p, and a former to place in the hole when we move on. After the move, we are either left with a dissected p and a latter that came out of the hole at our current location, or we have reached the end of the structure, and all the elements are formers. For our purposes, it is sufficient to note that this function (and similarly the $left$ function) can be implemented. For a detailed description of the implementation see McBride's paper on dissection [4].

We can use the dissection of p to make its $fmap$ tail-recursive. [4]

```
tmap :: Δp ↦ q ⇒ (s → t) → p s → p t
tmap f ps = continue (right (L ps)) where
        continue (L (s,pd)) = continue (right (R (pd, f s)))
        continue (R pt) = pt
```

where the requirement $\Delta p \mapsto q$ stipulates that we can take a derivate of p and $\Delta p \cong q$.

5.4 Tail Recursive Catamorphism

We now have the tools necessary to implement a tail recursive catamorphism, like our evaluation machine, generically for any polynomial functor. Given a p-algebra, $\phi :: p\ v \to v$, we traverse a μp depth first, left to right, to compute a final value v. At any point in the traversal, we'll be processing a node in the middle of processing its parent node, in the middle of processing its parent node, and so on. Because we're moving left to right, all the nodes to the left will have been transformed into values, while the nodes on the right still contain unprocessed μp's. Thus, our stack is a list of dissections $[\Delta p\ v\ (\mu\ p)]$ [4]. We can begin the procedure just like we began the evaluation machine for propositions:

```
tcata :: Δp ↦ q ⇒ (p v → v) → μ p → v
tcata φ t = load φ t []
```

To load a node we must dissect it to get at its subnodes, we then enter the node from the far left to begin processing it.

```
load :: Δp ↦ q ⇒ (p v → v) → μ p → [q v (μ p)] → v
load φ (In pt) stk = next φ (right (L pt)) stk
```

We may step into another subnode, which means we must dissect this one, pushing the previous dissection onto the stack to be continued later in the traversal. If we've reached the end of the node, we have all the values for its subnode, so we can apply ϕ and unload.

```
next :: Δp ↦ q ⇒ (p v → v) → (μ p, q v (μ p)) + p v → [q v (μ p)] → v
next φ (L (t,pd)) stk = load φ t (pd : stk)
next φ (R pv) stk = unload φ (φ pv) stk
```

After obtaining the value for a node, we can push the value onto the stack and resume our traversal of the parent node. If the stack is empty, we have the value for the root, so we can stop.

```
unload :: Δp ↦ q ⇒ (p v → v) → v → [q v (μ p)] → v
unload φ v (pd : stk) = next φ (right (R (pd,v))) stk
unload φ v [] = v
```

(Note that the definitions and implementations of *load*, *next*, and *unload* are all thanks to McBride as well [4].)

We can now easily implement our previous eval machine

```
eval :: μ PropP → Bool
eval = tcata φ where
        φ (ValP b) = b
        φ (NotP b) = -b
        φ (OrP b1 b2) = b1 || b2
```

5.5 Diagonal Dissection and the Zipper

We can recover the derivative of a datatype using its dissection via the method of diagonal dissection. That is to say, performing a dissection where there is no distinction between formers and latters: $\partial p\ x = \Delta p\ x\ x$. We can also recover the the plugging in operation. The following implementation is due to McBride [4], the operative functor is shown in a comment.

```
plug :: x → Δp x x → p x
plug{-K1 a-} x (K2 z) = error ``constants have no holes''
plug{-Id-} x (K2 ()) = Id x
plug{-p +1 q-} x (L2 pd) = L1 (plug{-p-} x pd)
plug{-p +1 q-} x (R2 qd) = R1 (plug{-q-} x qd)
plug{-p ×1 q} x (L2 (pd,2 Tqx)) = (plug{-p-} x pd,1 qx)
plug{-p ×1 q} x (R2 (Fpx,2 qd)) = (px,1 plug{-q-} x qd)
```

McBride also shows us how to recover Huet's zipper operations for a tree of type μp where p is a dissectable functor [4].

```
zUp,zDown,zLeft,zRight::Δp ↦ q ⇒ (μ p, [q (μ p) (μ p)]) → Maybe (μ p, [q (μ p) (μ p)])
```

where $Maybe\ x = Nothing \mid Just\ x$.

```
zUp (t,[]) = Nothing
zUp (t,pd:pds) = Just (In (plug{-p-} t pd), pds)
```

```
zDown (In pt,pds) = case right{-p-} (L pt) of
        L (t,pd) → Just (t,pd:pds)
        R _ → Nothing
zRight (t,[]) = Nothing
zRight (t::μ p,pd:pds) = case right{-p-} (R (pd,t)) of
        L (t',pd') → Just (t', pd':pds)
        R (_:: p (μ p)) → Nothing
```

McBride notes that Haskell's type checker needs assistance in the definition of $zRight$, for his explanation of this issue see his paper on dissection [4]. For us this peculiarity in the implementation is not important.

6 Discussion

In attempting to construct generic, manipulable traversals for arbitrary datatypes, we have seen four techniques with different advantages and disadvantages. Thanks to Huet [2], we found a structure perfectly suited to traversing tree structures. In effect, Huet had found one case of McBride's dissection operation. The obvious disadvantage of the zipper is that it lacks genericity. If one wanted to follow Huet's example for other structures, they would be forced to reimplement something similar to the zipper with regard to the structure in consideration. However, the zipper has the advantage of being efficient and easy to understand and extend. Future work on defining structures to represent traversals for specific datatypes could be useful for developing software, but we think that the generic approach is more interesting and will yield richer results.

For more generic traversals, we turned to Gibbons and origami programming. The techniques and functions associated with origami programming give us the ability to perform most of the transformations on typical datatypes like trees and lists that we might want. Many common patterns of computation on these structures can be expressed in a fold, or captured by a hylomorphism. Furthermore, although we did not discuss this point in much detail, most of these operations can also be defined in a generic way, obviating the need to reimplement functions for various datatypes. These approaches are quite useful, and in fact in his paper, Gibbons goes into detail on four different styles of origami programming and their uses [1]. Our main concern with origami programming and generic programming of a similar nature was that we have no access to the in-progress traversals. We are able to efficiently walk over structures to perform computations on them, but we cannot easily stop the traversals at arbitrary points to inspect or modify the data structure.

To begin tackling the issue of manipulating in-progress traversals, we examined the derivative of a datatype. The derivative gives us the type of one-hole contexts for its parent type. We found that this type exactly corresponded to the type of a single layer of the stack representing an in progress traversal. We were able to use this technique to mimic some of the functionality provided by the zipper; the plugging-in operation gave us a generic analog to Huet's insertion functions [2, 3]. However, the fact that we could not easily move around a data structure using the derivative lead us to its more powerful brother, dissection.

Dissection affords us nearly everything that we were looking for. Using dissection we can traverse arbitrary datatypes, but we also have first class access to the in-progress traversal. This allows us to modify the structure and the computation at any point. What's more, the generic nature of the computation of the dissection means that we can do this for any container types that can be expressed as polynomial bifunctors. This includes quite a large class of types. Further work in this area could yield a similar technique for even more complex types, such as trees containing lists, or other nested container types. More examples of the uses of dissection are also worth exploring, as we have seen the construction in theory but not in practice. McBride gives a few nice examples in his paper on the topic [4] but they provide only a quick look at what dissection can offer us. It would also be interesting to see how these techniques could be applied to the problem of data synchronization. Data synchronization often requires application specific methods, but dissection and even origami programming may help to generalize these techniques. It is also likely that more analogues from traditional calculus that apply to type theory could be discovered. McBride mentions this possibility, saying "one only has to open one's old school textbooks almost at random and ask 'what

does this mean for datatypes?' " [3]. Topics like the second derivative and the total (as opposed to partial derivative) may provide further insight into type theory and generic traversals.

We have seen that by exploiting types and manipulating type descriptions, we can create very powerful and structured generic programs, and we hope to have stimulated an interest in generic programming and calculations on types.

References

[1] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[2] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[3] Conor McBride. The derivative of a regular type is its type of one-hole contexts.

[4] Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *POPL*, 2008.