Clowns to the Left of me, Jokers to the Right

Dissecting Data Structures

Conor McBride

University of Nottingham

ctm@cs.nott.ac.uk

Abstract

This paper, submitted as a 'pearl', introduces a small but useful generalisation to the 'derivative' operation on datatypes underlying Huet's notion of 'zipper' (Huet 1997; McBride 2001; Abbott et al. 2005b), giving a concrete representation to one-hole contexts in data which is in *mid-transformation*. This operator, 'dissection', turns a container-like functor into a bifunctor representing a one-hole context in which elements to the left of the hole are distinguished in type from elements to its right.

I present dissection for polynomial functors, although it is certainly more general, preferring to concentrate here on its diverse applications. For a start, map-like operations over the functor and fold-like operations over the recursive data structure it induces can be expressed by tail recursion alone. Moreover, the derivative is readily recovered from the dissection, along with Huet's navigation operations. A further special case of dissection, 'division', captures the notion of *leftmost* hole, canonically distinguishing values with no elements from those with at least one. By way of a more practical example, division and dissection are exploited to give a relatively efficient generic algorithm for abstracting all occurrences of one term from another in a first-order syntax.

The source code for the paper is available online[1] and compiles with recent extensions to the Glasgow Haskell Compiler.

1. Introduction

There's an old *Stealer's Wheel* song with the memorable chorus:

'Clowns to the left of me, jokers to the right,
Here I am, stuck in the middle with you.'
Joe Egan, Gerry Rafferty

In this paper, I examine what it's like to be stuck in the middle of traversing and transforming a data structure. I'll show both you and the Glasgow Haskell Compiler how to calculate the datatype of a 'freezeframe' in a map- or fold-like operation from the datatype being operated on. That is, I'll explain how to compute a first-class data representation of the control structure underlying map and fold traversals, via an operator which I call *dissection*. Dissection turns out to generalise both the *derivative* operator underlying Huet's 'zippers' (Huet 1997; McBride 2001) and the notion of *division* used to calculate the non-constant part of a polynomial. Let me take you on a journey into the algebra and differential calculus of data, in search of functionality from structure.

Here's an example traversal—evaluating a very simple language of expressions:

```
data Expr = Val Int | Add Expr Expr
eval :: Expr → Int
eval (Val i)       = i
eval (Add e₁ e₂) = eval e₁ + eval e₂
```

What happens if we freeze a traversal? Typically, we shall have one piece of data 'in focus', with unprocessed data ahead of us and processed data behind. We should expect something a bit like Huet's 'zipper' representation of one-hole contexts (Huet 1997), but with different sorts of stuff either side of the hole.

In the case of our evaluator, suppose we proceed left-to-right. Whenever we face an Add, we must first go left into the first operand, recording the second Expr to process later; once we have finished with the former, we must go right into the second operand, recording the Int returned from the first; as soon as we have both values, we can add. Correspondingly, a Stack of these direction-with-cache choices completely determined where we are in the evaluation process. Let's make this structure explicit:[2]

```
type Stack = [Expr + Int]
```

Now we can implement an 'eval machine'—a tail recursion, at each stage stuck in the middle with an expression to decompose, loading the stack by going left, or a value to use, unloading the stack and moving right.

```
eval :: Expr → Int
eval e = load e []
load :: Expr → Stack → Int
load (Val i)       stk = unload i stk
load (Add e₁ e₂) stk = load e₁ (L e₂ : stk)
unload :: Int → Stack → Int
unload v  []             = v
unload v₁ (L e₂ : stk) = load e₂ (R v₁ : stk)
unload v₂ (R v₁ : stk) = unload (v₁ + v₂) stk
```

Each layer of this Stack structure is a *dissection* of Expr's recursion pattern. We have two ways to be stuck in the middle: we're either L e_2, on the left with an Exprs waiting to the right of us, or R v_1, on the right with an Int cached to the left of us. Let's find out how to do this in general, calculating the 'machine' corresponding to any old fold over finite first-order data.

[1] http://www.cs.nott.ac.uk/~ctm/CloJo/CJ.lhs

[2] For brevity, I write $\cdot + \cdot$ for Either, L for Left and R for Right

2. Polynomial Functors and Bifunctors

This section briefly recapitulates material which is quite standard. I hope to gain some generic leverage by exploiting the characterisation of recursive datatypes as fixpoints of polynomial functors. For more depth and detail, I refer the reader to the excellent *'Algebra of Programming'* (Bird and de Moor 1997).

If we are to work in a generic way with data structures, we need to present them in a generic way. Rather than giving an individual **data** declaration for each type we want, let us see how to build them from a fixed repertoire of components. I'll begin with the *polynomial* type constructors in *one* parameter. These are generated by constants, the identity, sum and product. I label them with a $_1$ subscript to distinguish them their bifunctorial cousins.

```
data K1 a        x = K1 a                  -- constant
data Id          x = Id x                  -- element
data (p +1 q) x = L1 (p x) | R1 (q x)     -- choice
data (p ×1 q) x = (p x ,1 q x)            -- pairing
```

Allow me to abbreviate one of my favourite constant functors, at the same time bringing it into line with our algebraic style.

```
type 11 = K1 ()
```

Some very basic 'container' type constructors can be expressed as polynomials, with the parameter giving the type of 'elements'. For example, the Maybe type constructor gives a choice between 'Nothing', a constant, and 'Just', embedding an element.

```
type Maybe = 11 +1 Id
Nothing = L1 (K1 ())
Just x  = R1 (Id x)
```

Whenever I reconstruct a datatype from this kit, I shall make a habit of 'defining' its constructors *linearly* in terms of the kit constructors. To aid clarity, I use these *pattern synonyms* on either side of a functional equation, so that the coded type acquires the same programming interface as the original. This is not standard Haskell, but these definitions may readily be expanded to code which is fully compliant, if less readable.

The 'kit' approach allows us to establish properties of whole classes of datatype at once. For example, the polynomials are all *functorial*: we can make the standard Functor class

```
class Functor p where
  fmap :: (s → t) → p s → p t
```

respect the polynomial constructs.

```
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)
```

Our reconstructed Maybe is functorial without further ado.

2.1 Datatypes as Fixpoints of Polynomial Functors

The Expr type is not itself a polynomial, but its branching structure is readily described by a polynomial. Think of each node of an Expr as a container whose elements are the immediate sub-Exprs:

```
type ExprP = K1 Int +1 Id ×1 Id
ValP i      = L1 (K1 i)
AddP e1 e2 = R1 (Id e1 ,1 Id e2)
```

Correspondingly, we should hope to establish the isomorphism

$$\mathsf{Expr} \cong \mathsf{ExprP}\ \mathsf{Expr}$$

but we cannot achieve this just by writing

```
type Expr = ExprP Expr
```

for this creates an infinite type expression, rather than an infinite type. Rather, we must define a recursive datatype which 'ties the knot': $\mu\, p$ instantiates p's element type with $\mu\, p$ itself.

```
data μ p = In (p (μ p))
```

Now we may complete our reconstruction of Expr

```
type Expr = μ ExprP
Val i       = In (ValP i)
Add e1 e2 = In (AddP e1 e2)
```

Now, the container-like quality of polynomials allows us to define a fold-like recursion operator for them, sometimes called the *iterator* or the *catamorphism*.[3] How can we compute a t from a $\mu\, p$? Well, we can expand a $\mu\, p$ tree as a $p\ (\mu\, p)$ container of subtrees, use p's fmap to compute ts recursively for each subtree, then post-process the $p\ t$ result container to produce a final result in t. The behaviour of the recursion is thus uniquely determined by the *p-algebra* $\phi :: p\ v \to v$ which does the post-processing.

```
(| · |) :: Functor p ⇒ (p v → v) → μ p → v
(|φ|) (In p) = φ (fmap (|φ|) p)
```

For example, we can write our evaluator as a catamorphism, with an algebra which implements each construct of our language for *values* rather than expressions. The pattern synonyms for ExprP help us to see what is going on:

```
eval :: μ ExprP → Int
eval = (|φ|) where
  φ (ValP i)        = i
  φ (AddP v1 v2) = v1 + v2
```

Catamorphism may appear to have a complex higher-order recursive structure, but we shall soon see how to turn it into a first-order tail-recursion whenever p is polynomial. We shall do this by dissecting p, distinguishing the 'clown' elements left of a chosen position from the 'joker' elements to the right.

2.2 Polynomial Bifunctors

Before we can start dissecting, however, we shall need to be able to manage *two* sorts of element. To this end, we shall need to introduce the polynomial *bifunctors*, which are just like the functors, but with two parameters.

```
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 ()
```

We have the analogous notion of 'mapping', except that we must supply one function for each parameter.

[3] Terminology is a minefield here: some people think of 'fold' as threading a *binary* operator through the elements of a container, others as replacing the constructors with an alternative algebra. The confusion arises because the two coincide for *lists*. There is no resolution in sight.

```
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) ⇒
         Bifunctor (p ×2 q) where
  bimap f g (p ,2 q) = (bimap f g p ,2 bimap f g q)
```

It's certainly possible to take fixpoints of bifunctors to obtain recursively constructed container-like data: one parameter stands for elements, the other for recursive sub-containers. These structures support both fmap and a suitable notion of catamorphism. I can recommend (Gibbons 2007) as a useful tutorial for this 'origami' style of programming.

2.3 Nothing is Missing

We are still short of one basic component: Nothing. We shall be constructing types which organise 'the ways to split at a position', but what if there are *no* ways to split at a position (because there are no positions)? We need a datatype to represent impossibility and here it is:

```
data Zero
```

Elements of Zero are hard to come by—elements worth speaking of, that is. Correspondingly, if you have one, you can exchange it for anything you want.

```
magic :: Zero → a
magic x = x `seq` error "we never get this far"
```

I have used Haskell's seq operator to insist that magic evaluate its argument. This is necessarily $\bot$, hence the error clause can never be executed. In effect magic *refutes its input*.

We can use p Zero to represent 'ps with no elements'. For example, the only inhabitant of [Zero] mentionable in polite society is [ ]. Zero gives us a convenient way to get our hands on exactly the constants, common to every instance of p. Accordingly, we should be able to embed these constants into any other instance:

```
inflate :: Functor p ⇒ p Zero → p x
inflate = fmap magic
```

However, it's rather a lot of work traversing a container just to transform all of its nonexistent elements. If we cheat a little, we can do nothing much more quickly, and just as safely!

```
inflate :: Functor p ⇒ p Zero → p x
inflate = unsafeCoerce♯
```

This unsafeCoerce♯ function behaves operationally like $\lambda x \to x$, but its type, $a \to b$, allows the programmer to intimidate the typechecker into submission. It is usually present but well hidden in the libraries distributed with Haskell compilers, and its use requires extreme caution. Here we are sure that the only Zero computations mistaken for xs will fail to evaluate, so our optimisation is safe.

Now that we have Zero, allow me to abbreviate

```
type 01 = K1 Zero
type 02 = K2 Zero
```

3. Clowns, Jokers and Dissection

We shall need three operators which take polynomial functors to bifunctors. Let me illustrate them: consider functors parametrised by elements (depicted •) and bifunctors are parametrised by clowns (◀) to the left and jokers (▶) to the right. I show a typical $p\ x$ as a container of •s

{–•–•–•–•–•–}

Firstly, 'all clowns' ∠p lifts p uniformly to the bifunctor which uses its left parameter for the elements of p.

{–◀–◀–◀–◀–◀–}

We can define this uniformly:

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

Note that ∠Id ≅ Fst.

Secondly, 'all jokers' ⦣p is the analogue for the right parameter.

{–▶–▶–▶–▶–▶–}

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

Note that ⦣Id ≅ Snd.

Thirdly, 'dissected' ◭p takes p to the bifunctor which chooses a position in a p and stores clowns to the left of it and jokers to the right.

{–◀–◀–○–▶–▶–}

We must clearly define this case by case. Let us work informally and think through what to do each polynomial type constructor. Constants have no positions for elements,

{–}

so there is no way to dissect them:

```
◭(K1 a) = 02
```

The Id functor has just one position, so there is just one way to dissect it, and no room for clowns or jokers, left or right.

{–•–} ⟶ {–○–}

```
◭Id = 12
```

Dissecting a $p +_1 q$, we get either a dissected p or a dissected q.

L1 {–•–•–•–} ⟶ L2 {–◀–○–▶–}
R1 {–•–•–•–} ⟶ R2 {–◀–○–▶–}

```
◭(p +1 q) = ◭p +2 ◭q
```

So far, these have just followed Leibniz's rules for the derivative, but for pairs $p \times_1 q$ we see the new twist. When dissecting a pair, we choose to dissect either the left component (in which case the right component is all jokers) or the right component (in which case the left component is all clowns).

({–•–•–•–} ,1 {–•–•–•–}) ⟶ { L2 ({–◀–○–▶–} ,2 {–▶–▶–▶–}) / R2 ({–◀–◀–◀–} ,2 {–◀–○–▶–})

```
◭(p ×1 q) = ◭p ×2 ⦣q +2 ∠p ×2 ◭q
```

Now, in Haskell, this kind of type-directed definition can be done with type-class programming (Hallgren 2001; McBride 2002). Allow me to abuse notation very slightly, giving dissection constraints a slightly more functional notation, after the manner of (Neubauer et al. 2001):

```
class (Functor p, Bifunctor p̂) ⇒ Δp ↦ p̂ | p → p̂ where
    -- methods to follow
```

In ASCII, $\Delta p \mapsto \hat{p}$ is rendered relationally as `Diss p p''`, but the annotation $|p \rightarrow \hat{p}$ is a *functional dependency*, indicating that p determines $\hat{p}$, so it is appropriate to think of $\Delta\cdot$ as a functional operator, even if we can't quite treat it as such in practice.

I shall extend this definition and its instances with operations shortly, but let's start by translating our informal program into type-class Prolog:

```
instance Δ(K1 a) ↦ 02

instance Δ Id ↦ 12

instance (Δp ↦ p̂, Δq ↦ q̂) ⇒
    Δp +1 q ↦ p̂ +2 q̂

instance (Δp ↦ p̂, Δq ↦ q̂) ⇒
    Δp ×1 q ↦ p̂ ×2 ⌟q +2 ⌞p ×2 q̂
```

Before we move on, let us just check that we get the answer we expect for our expression example.

$$\Delta \mathsf{K}_1\ \mathsf{Int} +_1 \mathsf{Id} \times_1 \mathsf{Id} \mapsto 0_2 +_2 1_2 \times_2 \lrcorner\mathsf{Id} +_2 \llcorner\mathsf{Id} \times_2 1_2$$

A bit of simplification tells us:

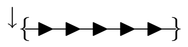
$$\Delta \mathsf{ExprP}\ \mathsf{Int}\ \mathsf{Expr} \cong \mathsf{Expr} + \mathsf{Int}$$

Dissection (with values to the left and expressions to the right) has calculated the type of layers of our stack!

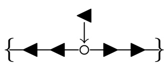
4. How to Creep Gradually to the Right

If we're serious about representing the state of a traversal by a dissection, we had better make sure that we have some means to move from one position to the next. In this section, we'll develop a method for the $\Delta p \mapsto \hat{p}$ class which lets us move rightward one position at a time. I encourage you to move leftward yourselves.

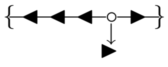
What should be the type of this operation? Consider, firstly, where our step might start. If we follow the usual trajectory, we'll start at the far left—and to our right, all jokers.



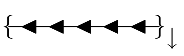
Once we've started our traversal, we'll be in a dissection. To be *ready* to move, we we must have a clown to put into the hole.



Now, think about where our step might take us. If we end up at the next position, out will pop the next joker, leaving the new hole.



But if there are no more positions, we'll emerge at the far right, all clowns.



Putting this together, we add to **class** $\Delta p \mapsto \hat{p}$ the method

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

Let me show you how to implement the instances by pretending to write a *polytypic* function after the manner of (Jansson and Jeuring 1997), showing the operative functor in a comment.

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

You can paste each clause of right {-p-} into the corresponding $\Delta p \mapsto \cdot$ instance.

For constants, we jump all the way from far left to far right in one go; we cannot be in the middle, so we refute that case.

```
right{-K1 a-} x = case x of
  L (K1 a)      → R (K1 a)
  R (K2 z, c)   → magic z
```

We can step into a single element, or step out.

```
right{-Id x-} x = case x of
  L (Id j)      → L (j, K2 ())
  R (K2 (), c)  → R (Id c)
```

For sums, we make use of the instance for whichever branch is appropriate, being careful to strip tags beforehand and replace them afterwards.

```
right{-p +1 q-} x = case x of
  L (L1 pj)      → mindp (right{-p-} (L pj))
  L (R1 qj)      → mindq (right{-q-} (L qj))
  R (L2 pd, c)   → mindp (right{-p-} (R (pd, c)))
  R (R2 qd, c)   → mindq (right{-q-} (R (qd, c)))
  where
    mindp (L (j, pd)) = L (j, L2 pd)
    mindp (R pc)      = R (L1 pc)
    mindq (L (j, qd)) = L (j, R2 qd)
    mindq (R qc)      = R (R1 qc)
```

For products, we must start at the left of the first component and end at the right of the second, but we also need to make things join up in the middle. When we reach the far right of the first component, we must continue from the far left of the second.

```
right{-p ×1 q-} x = case x of
  L (pj ,1 qj)              → mindp (right{-p-} (L pj))        qj
  R (L2 (pd ,2 ⌟qj), c)     → mindp (right{-p-} (R (pd, c))) qj
  R (R2 (⌞pc ,2 qd), c)     → mindq pc (right{-q-} (R (qd, c)))
  where
    mindp (L (j, pd)) qj = L (j, L2 (pd ,2 ⌟qj))
    mindp (R pc)      qj = mindq pc (right{-q-} (L qj))
    mindq pc (L (j, qd)) = L (j, R2 (⌞pc ,2 qd))
    mindq pc (R qc)      = R (pc ,1 qc)
```

Let's put this operation straight to work. If we can dissect p, then we can make its fmap operation tail recursive. Here, the jokers are the source elements and the clowns are the target elements.

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

4.1 Tail-Recursive Catamorphism

If we want to define the catamorphism via dissection, we could just replace fmap by tmap in the definition of $(\!|\cdot|\!)$, but that would be cheating! The point, after all, is to turn a higher-order recursive program into a tail-recursive machine. We need some kind of *stack*.

Suppose we have a p-algebra, $\phi{::}p\ v \rightarrow v$, and we're traversing a $\mu\, p$ depth-first, left-to-right, in order to compute a 'value' in v. At any given stage, we'll be processing a given node, in the middle of traversing her mother, in the middle of traversing her grandmother, and so on in a maternal line back to the root. We'll have visited all the nodes left of this line and thus have computed vs for them; right of the line, each node will contain a $\mu\, p$ waiting for her turn. Correspondingly, our stack is a list of dissections:

$$[\Delta p\ v\ (\mu\, p)]$$

We start, ready to load a tree, with an empty stack.

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

To load a node, we unpack her container of subnodes and step in from the far left.

```
load :: ◭p ↦ p̂ ⇒ (p v → v) → μ p → [p̂ v (μ p)] → v
load φ (In pt) stk = next φ (right{-p-} (L pt)) stk
```

After a step, we might arrive at another subnode, in which case we had better load her, suspending our traversal of her mother by pushing the dissection on the stack.

```
next :: ◭p ↦ p̂ ⇒ (p v → v) →
        (μ p, p̂ v (μ p)) + p v → [p̂ 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
```

Alternatively, our step might have taken us to the far right of a node, in which case we have all her subnodes' values: we are ready to apply the algebra ϕ to get her own value, and start unloading.

Once we have a subnode's value, we may resume the traversal of her mother, pushing the value into her place and moving on.

```
unload :: ◭p ↦ p̂ ⇒ (p v → v) → v → [p̂ v (μ p)] → v
unload φ v (pd : stk) = next φ (right{-p-} (R (pd, v))) stk
unload φ v []         = v
```

On the other hand, if the stack is empty, then we're holding the value for the root node, so we're done! As we might expect:

```
eval :: μ ExprP → Int
eval = tcata φ where
  φ (ValP i)       = i
  φ (AddP v₁ v₂) = v₁ + v₂
```

5. Derivative Derived by Diagonal Dissection

The dissection of a functor is its bifunctor of one-hole contexts distinguishing 'clown' elements left of the hole from 'joker' elements to its right. If we remove this distinction, we recover the usual notion of one-hole context, as given by the *derivative* (McBride 2001; Abbott et al. 2005b). Indeed, we've already seen, the rules for dissection just refine the centuries-old rules of the calculus with a left-right distinction. We can undo this refinement by taking the diagonal of the dissection, identifying clowns with jokers.

$$\partial p\ x = ◭ p\ x\ x$$

Let us now develop the related operations.

5.1 Plugging In

We can add another method to **class** $◭ p \mapsto \hat{p}$,

```
plug :: x → p̂ x x → p x
```

saying, in effect, that if clowns and jokers coincide, we can fill the hole directly and without any need to traverse all the way to the end. The implementation is straightforward.

```
plug{-K₁ a-} x (K₂ z) = magic z

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 ⌊ qx)) = (plug{-p-} x pd ,₁ qx)
plug{-p ×₁ q-} x (R₂ (⌊ px ,₂ qd)) = (px ,₁ plug{-q-} x qd)
```

5.2 Zipping Around

We now have almost all the equipment we need to reconstruct Huet's operations (Huet 1997), navigating a tree of type $\mu\, p$ for some dissectable functor p.

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

I leave zLeft as an exercise, to follow the implementation of the leftward step operation, but the other three are straightforward uses of plug{-p-} and right{-p-}. This implementation corresponds quite closely to the Generic Haskell version from (Hinze et al. 2004), but requires a little less machinery.

```
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
```

Notice that I had to give the typechecker a little help in the definition of zRight. The trouble is that $◭\cdot$ is not known to be *invertible*, so when we say right{-p-} (R (pd, t)), the type of pd does not actually determine p—it's easy to forget that the{-p-} is only a comment. I've forced the issue by collecting p from the type of the input tree and using it to fix the type of the 'far right' failure case. This is perhaps a little devious, but when type inference is compulsory, what can one do?

6. Division: No Clowns!

The derivative is not the only interesting special case of dissection. In fact, my original motivation for inventing dissection was to find an operator $\ell\cdot$ for 'leftmost' on suitable functors p which would induce an isomorphism reminiscent of the 'remainder theorem' in algebra.

$$p\ x\ \cong\ (x, \ell p\ x) + p\ \mathsf{Zero}$$

This $\ell p\ \ x$ is the 'quotient' of $p\ \ x$ on division by x, and it represents whatever can remain after the *leftmost* element in a $p\ \ x$ has been removed. Meanwhile, the 'remainder', $p\ \ \mathsf{Zero}$, represents those ps with no elements at all. Certainly, the finitely-sized containers should give us this isomorphism, but what is $\ell\cdot$? It's the context of the leftmost hole. It should not be possible to move any further left, so there should be *no clowns*! We need

$$\ell p\ x = ◭ p\ \mathsf{Zero}\ x$$

For the polynomials, we shall certainly have

```
divide :: ◭p ↦ p̂ ⇒ p x → (x, p̂ Zero x) + p Zero
divide px = right{-p-} (L px)
```

To compute the inverse, I could try waiting for you to implement the leftward step: I know we are sure to reach the *far* left, for your only alternative is to produce a clown! However, an alternative is at the ready. I can turn a leftmost hole into any old hole if I have

```
inflateFst :: Bifunctor p ⇒ p Zero y → p x y
inflateFst = unsafeCoerce♯   -- faster than bimap magic id
```

Now, we may just take

```
divide˘ :: ◭p ↦ p̂ ⇒ (x, p̂ Zero x) + p Zero → p x
divide˘ (L (x, pl)) = plug{-p-} x (inflateFst pl)
divide˘ (R pz)      = inflate pz
```

It is straightforward to show that these are mutually inverse by induction on polynomials.

7. A Generic Abstractor

So far this has all been rather jolly, but is it just a mathematical amusement? Why should I go to the trouble of constructing an explicit context structure, just to write a fold you can give directly by higher-order recursion? By way of a finale, let me present a more realistic use-case for dissection, where we exploit the first-order representation of the context by inspecting it: the task is to abstract all occurrences of one term from another, in a generic first-order syntax.

7.1 Free Monads and Substitution

What is a 'generic first-order syntax'? A standard way to get hold of such a thing is to define the *free monad* p^* of a (container-like) functor p (Barr and Wells 1984).

```
data p* x = V x | C (p (p* x))
```

The idea is that p represents the signature of constructors in our syntax, just as it represented the constructors of a datatype in the $\mu\, p$ representation. The difference here is that $p^*\ x$ also contains *free variables* chosen from the set x. The monadic structure of p^* is that of substitution.

```
instance Functor p ⇒ Monad (p*) where
  return x = V x
  V x ≫= σ = σ x
  C pt ≫= σ = C (fmap (≫=σ) pt)
```

Here ≫= is the *simultaneous* substitution from variables in one set to terms over another. However, it's easy to build substitution for a single variable on top of this. If we a term t over Maybe x, we can substitute some s for the distinguished variable, Nothing. Let us rename Maybe to S, 'successor', for the occasion:

```
type S = Maybe
(⇂) :: Functor p ⇒ p* (S x) → p* x → p* x
t ⇂ s = t ≫= σ where
  σ Nothing = s
  σ (Just x) = V x
```

Our mission is to compute the 'most abstract' inverse to (⇂s), for suitable p and x, some

```
(↾) :: ... ⇒ p* x → p* x → p* (S x)
```

such that $(t \restriction s) \downharpoonleft s = t$, and moreover that fmap Just s occurs *nowhere* in $t \restriction s$. In order to achieve this, we've got to abstract *every* occurrence of s in t as V Nothing and apply Just to all the other variables. Taking $t \restriction s$ = fmap Just t is definitely wrong!

7.2 Indiscriminate Stop-and-Search

The obvious approach to computing $t \restriction s$ is to traverse t checking everywhere if we've found s. We shall need to be able to test equality of terms, so we first must confirm that our signature functor p *preserves* equality, i.e., that we can lift equality eq on x to equality $\cdot \lceil eq \rceil \cdot$ on $p\ x$.

```
class PresEq p where
  · ⌈·⌉ · ::(x → x → Bool) → p x → p x → Bool
instance Eq a ⇒ PresEq (K₁ a) where
  K₁ a1 ⌈eq⌉ K₁ a2 = a1 ≡ a2
instance PresEq Id where
  Id x1 ⌈eq⌉ Id x2 = eq x1 x2
instance (PresEq p, PresEq q) ⇒ PresEq (p +₁ q) where
  L₁ p1 ⌈eq⌉ L₁ p2 = p1 ⌈eq⌉ p2
  R₁ q1 ⌈eq⌉ R₁ q2 = q1 ⌈eq⌉ q2
  _ ⌈eq⌉     _     = False
instance (PresEq p, PresEq q) ⇒ PresEq (p ×₁ q) where
  (p1 ,₁ q1) ⌈eq⌉ (p2 ,₁ q2) = p1 ⌈eq⌉ p2 ∧ q1 ⌈eq⌉ q2
instance (PresEq p, Eq x) ⇒ Eq (p* x) where
  V x  ≡ V y  = x ≡ y
  C ps ≡ C pt = ps ⌈≡⌉ pt
  _    ≡ _    = False
```

We can now make our first attempt:

```
(↾) :: (Functor p, PresEq p, Eq x) ⇒ p* x → p* x → p* (S x)
t    ↾ s | t ≡ s = V Nothing
V x  ↾ s         = V (Just x)
C pt ↾ s         = C (fmap (↾s) pt)
```

Here, I'm exploiting Haskell's *Boolean guards* to test for a match first: only if the fails do we fall through and try to search more deeply inside the term. This is short and obviously correct, but it's rather inefficient. If s is small and t is large, we shall repeatedly compare s with terms which are far too large to stand a chance of matching. It's rather like testing if xs has suffix ys like this.

```
hasSuffix :: Eq x ⇒ [x] → [x] → Bool
hasSuffix xs     ys | xs ≡ ys = True
hasSuffix []     ys           = False
hasSuffix (x:xs) ys           = hasSuffix xs ys
```

If we ask hasSuffix `"xxxxxxxxxxxx"` `"xxx"`, we shall test if `'x'` ≡ `'x'` thirty times, not three. It's more efficient to reverse both lists and check *once* for a *prefix*. With fast reverse, this takes linear time.

```
hasSuffix :: Eq x ⇒ [x] → [x] → Bool
hasSuffix xs ys = hasPrefix (reverse xs) (reverse ys)
hasPrefix :: Eq x ⇒ [x] → [x] → Bool
hasPrefix xs     []               = True
hasPrefix (x:xs) (y:ys) | x ≡ y   = hasPrefix xs ys
hasPrefix _      _                = False
```

7.3 Hunting for a Needle in a Stack

We can adapt the 'reversal' idea to our purposes. The divide function tells us how to find the leftmost position in a polynomial container, if it has one. If we iterate divide, we can navigate our way down the left spine of a term to its leftmost leaf, stacking the contexts as we go. That's a way to reverse a tree!

A leaf is either a variable or a constant. A term either is a leaf or has a leftmost subterm. To see this, we just need to adapt divide for the possibility of variables.

```
data Leaf p x = VL x | CL (p Zero)
leftOrLeaf :: Δp ↦ p̂ ⇒
                 p* x → (p* x, p̂ Zero (p* x)) + Leaf p x
leftOrLeaf (V x)  = R (VL x)
leftOrLeaf (C pt) = fmap CL (divide pt)
```

Now we can reverse the term we seek into the form of a 'needle'—a leaf with a straight spine of leftmost holes running all the way back to the root

```
needle :: Δp ↦ p̂ ⇒ p* x → (Leaf p x, [p̂ Zero (p* x)])
needle t = grow t [] where
  grow t pls = case leftOrLeaf t of
    L (t′, pl) → grow t′ (pl : pls)
    R l        → (l, pls)
```

Given this needle representation of the search term, we can implement the abstraction as a stack-driven traversal, hunt which tries for a match only when it reaches a suitable leaf. We need only check for our needle when we're standing at the end of a left spine at least as long. Let us therefore split our 'state' into an inner left spine and an outer stack of dissections.

```
(↾) :: (Δp ↦ p̂, PresEq p, PresEq2 p̂, Eq x) ⇒
        p* x → p* x → p* (S x)
t ↾ s = hunt t [] [] where
  (neel, nees) = needle s
  hunt t spi stk = case leftOrLeaf t of
    L (t′, pl) → hunt t (pl : spi) stk
    R l        → check spi nees (l ≡ neel)
     where
       check = ···
```

Current technology for type annotations makes it hard for me to write hunt's type in the code. Informally, it's this:

```
hunt :: p* x → [ℓp (p* x)] → [▲p (p* (S x)) (p* x)] →
    p* (S x)
```

Now, check is rather like hasPrefix, except that I've used a little accumulation to ensure the expensive equality tests happen after the cheap length test.

```
check spi' [] True = next (V Nothing) (spi' ⇈++ stk)
check (spl : spi') (npl : nees') b =
  check spi' nees' (b ∧ spl ⌈magic |≡⌉ npl)
check _ _ _ = next (leafS l) (spi ⇈++ stk) where
  leafS (VL x)  = V (Just x)
  leafS (CL pz) = C (inflate pz)
```

For the equality tests we need $\cdot\lceil\cdot\mid\cdot\rceil\cdot$, the bifunctorial analogue of $\cdot\lceil\cdot\rceil\cdot$, although as we're working with ℓp, we can just use magic to test equality of clowns. The same trick works for Leaf equality:

```
instance (PresEq p, Eq x) ⇒ Eq (Leaf p x) where
  VL x ≡ VL y = x ≡ y
  CL a ≡ CL b = a ⌈magic⌉ b
  _    ≡ _    = False
```

Now, instead of returning a Bool, check must explain how to *move on*. If our test succeeds, we must move on from our matching subterm's position, abstracting it: we throw away the matching prefix of the spine and stitch its suffix onto the stack. However, if the test fails, we must move right from the current *leaf*'s position, injecting it into p^* (S x) and stitching the original spine to the stack. Stitching (⇈++) is just a version of 'append' which inflates a leftmost hole to a dissection.

```
(⇈++) :: Bifunctor p ⇒ [p Zero y] → [p x y] → [p x y]
[] ⇈++ pxys = pxys
(pzy : pzys) ⇈++ pxys = inflateFst pzy : pzys ⇈++ pxys
```

Correspondingly, next tries to move rightwards given a 'new' term and a stack. If we can go right, we get the next 'old' term along, so we start hunting again with an empty spine.

```
next t' (pd : stk) = case right{-p-} (R (pd, t')) of
  L (t, pd') → hunt t [] (pd' : stk)
  R pt'      → next (C pt') stk
next t' [] = t'
```

If we reach the far right of a p, we pack it up and pop on out. If we run out of stack, we're done!

8. Discussion

The story of dissection has barely started, but I hope I have communicated the intuition behind it and sketched some of its potential applications. Of course, what's missing here is a more *semantic* characterisation of dissection, with respect to which the operational rules for $\blacktriangle p$ may be justified.

It is certainly straightforward to give a shapes-and-positions analysis of dissection in the categorical setting of *containers* (Abbott et al. 2005a), much as we did with the derivative (Abbott et al. 2005b). The basic point is that where the derivative requires element positions to have decidable *equality* ('am I in the hole?'), dissection requires a total order on positions with decidable *trichotomy* ('am I in the hole, to the left, or to the right?'). The details, however, deserve a paper of their own.

I have shown dissection for polynomials here, but it is clear that we can go much further. For example, the dissection of *list* gives a list of clowns and a list of jokers:

$$\blacktriangle[\,] = \angle[\,] \times_2 \backslash[\,]$$

Meanwhile, the *chain rule*, for functor composition, becomes

$$\blacktriangle(p \circ_1 q) = \blacktriangle q \times_2 (\blacktriangle p) \circ_2 (\angle q; \backslash q)$$

where

$$\textbf{data}\ (p \circ_2 (q; r))\ c\ j = (p\ (q\ c\ j)\ (r\ c\ j)) \circ_2 (\cdot;\cdot)$$

That is, we have a dissected p, with clown-filled qs left of the hole, joker-filled qs right of the hole, and a dissected q in the hole. If you specialise this to *division*, you get

$$\ell(p \circ_1 q)\ x \cong \ell q\ x \times \blacktriangle p\ (q\ \mathsf{Zero})\ (q\ x)$$

The leftmost x in a $p\ (q\ x)$ might not be in a leftmost p position: there might be q-leaves to the left of the q-node containing the first element. That is why it was necessary to invent $\blacktriangle\cdot$ to define $\ell\cdot$, an operator which deserves further study in its own right. For finite structures, its iteration gives rise to a power series formulation of datatypes directly, finding all the elements left-to-right, where iterating $\partial\cdot$ finds them in any order. There is thus a significant connection with the notion of *combinatorial species* as studied by Joyal (Joyal 1986) and others.

The whole development extends readily to the *multivariate* case, although this a little more than Haskell can take at present. The general $\blacktriangle_i$ dissects a mutli-sorted container at a hole of sort i, and splits all the sorts into clown- and joker-variants, doubling the arity of its parameter. The corresponding ℓ_i finds the contexts in which an element of sort i can stand leftmost in a container. This corresponds exactly to Brzozowski's notion of the 'partial derivative' of a regular expression (Brzozowski 1964).

But if there is a message for programmers and programming language designers, it is this: the miserablist position that types exist only to police errors is thankfully no longer sustainable, once we start writing programs like this. By permitting calculations of types and from types, we discover what programs we can have, just for the price of structuring our data. What joy!

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005a. Applied Semantics: Selected Topics.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. ∂ for data: derivatives of data structures. *Fundamenta Informaticae*, 65(1&2):1–28, 2005b.

Michael Barr and Charles Wells. *Toposes, Triples and Theories*, chapter 9. Number 278 in Grundlehren der Mathematischen Wissenschaften. Springer, New York, 1984.

Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.

Janusz Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964.

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. To appear.

Thomas Hallgren. Fun with functional dependencies. In Joint Winter Meeting of the Departments of Science and Computer Engineering, Chalmers University of Technology and Goteborg University, Varberg, Sweden., January 2001.

Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. *Science of Computer Programmming*, 51:117–151, 2004.

Gérard Huet. The Zipper. *Journal of Functional Programming*, 7 (5):549–554, 1997.

Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In *Proceedings of POPL '97*, pages 470–482. ACM, 1997.

André Joyal. Foncteurs analytiques et espéces de structures. In *Combinatoire énumérative*, number 1234 in LNM, pages 126 – 159. 1986.

Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.cs.nott.ac.uk/~ctm/diff.pdf`, 2001.

Conor McBride. Faking It (Simulating Dependent Types in Haskell). *Journal of Functional Programming*, 12(4& 5):375–392, 2002. Special Issue on Haskell.

Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A Functional Notation for Functional Dependencies. In *The 2001 ACM SIGPLAN Haskell Workshop*, Firenze, Italy, September 2001.