Clowns to the Left of me, Jokers to the Right *(Pearl)*

Dissecting Data Structures

Conor McBride
University of Nottingham
ctm@cs.nott.ac.uk

Abstract

This paper 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 undergoing *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 here as a *generic* program, albeit for polynomial functors only. The notion is certainly applicable more widely, but here I prefer to concentrate 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. Further, the derivative is readily recovered from the dissection. Indeed, it is the dissection structure which delivers Huet's operations for *navigating* zippers.

The original motivation for dissection was to define 'division', capturing the notion of *leftmost* hole, canonically distinguishing values with no elements from those with at least one. Division gives rise to an isomorphism corresponding to the *remainder theorem* in algebra. By way of a larger 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; I.1.1 [*Symbolic and Algebraic Manipulation*]: Expressions and Their Representation

General Terms Algorithms, Design, Languages, Theory

Keywords Datatype, Differentiation, Dissection, Division, Generic Programming, Iteration, Polynomial, Stack, Tail Recursion, Traversal, Zipper

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

POPL'08, January 7–12, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-689-9/08/0001…$5.00

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 datatypes, 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' and a *hole* in the expression where it belongs, with unprocessed data ahead of us and processed data behind. We should expect something a bit like Huet's 'zipper' representation of a one-hole context (Huet 1997), a stack-like structure carrying position information and cacheing all the out-of-focus data for each node between the hole and the root. However, now we need different sorts of stuff on either side of the hole.

In the case of our evaluator, suppose we proceed left-to-right. Whenever we face an Add, we start by going 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 them. Correspondingly, a Stack of these direction-with-cache choices completely determines 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-recursive program, at each stage stuck in the middle with either an expression to decompose, in which case we load the stack and go left, or a value to return, in which case we unload the stack and try to move right.

[2] For brevity, I write $\cdot + \cdot$ for Either, L for Left and R for 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 Expr waiting to the right of us, or R v_1, on the right with an Int cached to the left of us. Danvy and colleagues have shown us how to calculate abstract machines from *programs* by transforming them into a sequential, continuation-passing style, then defunctionalising the result (Ager et al. 2003). Here, however, the structure of the program comes from the structure of its data. Correspondingly, we can mechanize the stack construction more directly, by working from the *types*.

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, q x)1            -- 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 an old idea (Aitken and Reppy 1992), but it is not standard Haskell: 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, using Haskell's rather powerful facilities for *ad hoc* polymorphism (Wadler and Blott 1989). 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, q)1 = (fmap f p, fmap f q)1
```

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, Id e2)1
```

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)
```

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 value in v from recursive data in $\mu\, p$? First, we can expand a $\mu\, p$ tree as a $p\ (\mu\, p)$ container of subtrees; next, we can use p's fmap operator to deliver a $p\ v$, recursively computing the value for each subtree; finally, we can post-process the $p\ v$ value container to produce a final result in v. 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
```

[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.

A 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 elements. 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, q x y)2
type 12 = K2 ()
```

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

```
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, q)2  = (bimap f g p, bimap f g q)2
```

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. If you can reduce an element of Zero to a well-defined value, you can exchange it for anything you want!

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

I have used Haskell's seq operator to insist that refute evaluate its argument. This is necessarily undefined, hence the error clause can never be executed. In effect, refute rejects 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:[4]

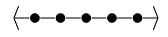
```
inflate :: Functor p ⇒ p Zero → p x
inflate = fmap refute
```

Now that we have Zero, allow me to abbreviate

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

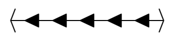
3. Clowns, Jokers and Dissection

Let us now develop this idea of 'dissecting' functors with the help of some visual stimuli. If we consider functors parametrised by elements, depicted ●, we can draw a typical value in some $p\ x$ as a container of ●s:



We shall need to relate these functors to bifunctors which refine the notion of element into two kinds: clowns (◀) to the left and jokers (▶) to the right of some hole (◯). More particularly, we shall need three operators which take polynomial functors to bifunctors, classifying their elements as clowns, jokers or the hole.

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

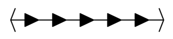


We can define this uniformly:

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

Note that $\angle$Id $\cong$ Fst.

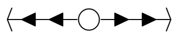
Secondly, 'all jokers' $\lrcorner p$ is the analogue for the right parameter.



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

Note that $\lrcorner$Id $\cong$ Snd.

Thirdly, 'dissection' Δp takes p to the bifunctor which chooses a position in a p, storing clowns to the left of it and jokers to the right.

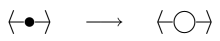


We must clearly define dissection case by case. Let us work informally and think through what to do for each polynomial type constructor. Constants have no positions for elements,

so there is no way to dissect them:

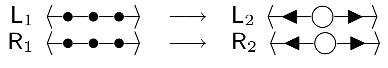
```
Δ(K1 a) = 02
```

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.



```
ΔId = 12
```

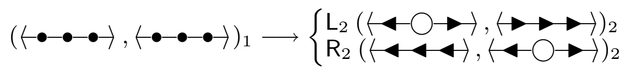
Dissecting a $p +_1 q$, we get either a dissected p or a dissected q.



[4] If the compiler adopts a uniform representation for polymorphic data, it may be profitable to replace inflate by an 'unsafe' cast.

$\Delta(p +_1 q) = \Delta p \; +_2 \; \Delta q$

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).



$\Delta(p \times_1 q) = \Delta p \times_2 \diagdown q \; +_2 \; \diagup p \times_2 \Delta q$

How can we implement this? Fortunately, Haskell supports the overloading of functions involving classes of *related* types (Peyton Jones et al. 1997). Instance declarations resemble logic programs and constraint-solving proceeds by type-directed backchaining, rather like Prolog but without backtracking. It pays to think of type classes as a programming language (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 $|p \to \hat{p}$ 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.[5]

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 ΔId ↦ 1₂

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

instance (Δp ↦ p̂, Δq ↦ q̂) ⇒
  Δ(p ×₁ q) ↦ p̂ ×₂ ╲q +₂ ╱p ×₂ q̂
```

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

$\Delta(\mathsf{K}_1\ \mathsf{Int} +_1 \mathsf{Id} \times_1 \mathsf{Id}) \mapsto 0_2 \; +_2 \; 1_2 \times_2 \diagdown \mathsf{Id} \; +_2 \; \diagup \mathsf{Id} \times_2 1_2$

A bit of simplification tells us:

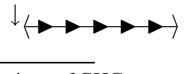
$$\Delta \mathsf{ExprP}\ \mathsf{Int}\ \mathsf{Expr} \cong \mathsf{Expr} + \mathsf{Int}$$

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

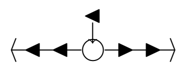
If we're serious about representing the state of a traversal by a dissection, we had better make sure that we have some means to move from one position to the next. In this section, we'll develop a method for the $\Delta p \mapsto \hat{p}$ class which lets us move rightward one position at a time. I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

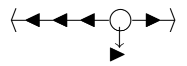


[5] Recent unsupported extensions of GHC support 'associated type families', a more functional approach to type-level programming. I am grateful to Brandon Moore for the news that the code in this paper ports seamlessly.

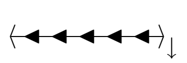
Once we've started our traversal, we'll be in a dissection. To be *ready* to move, we must have a clown to put into the hole.



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 of right. I shall adopt the style of *polytypic* programming (Jansson and Jeuring 1997), pretending to match on the polynomial parameter as if it were a special kind of argument.

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

Sadly, these 'arguments' are just Haskell comments, but they serve a useful documentary purpose. In particular, they show in which instance each clause belongs. If you paste each clause of right{-p-} into the corresponding $\Delta p \mapsto \hat{p}$ instance, Haskell's type class mechanism can interpret each appeal to right{-p-} by *compile-time* recursion on p.

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)  → refute 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{-$p \times_1 q$-} $x$ = case $x$ of
  L $(pj, qj)_1$              → mindp (right{-$p$-} (L $pj$))       $qj$
  R (L$_2$ $(pd, \lrcorner qj)_2$, $c$) → mindp (right{-$p$-} (R $(pd, c)$)) $qj$
  R (R$_2$ $(\angle pc, qd)_2$, $c$) → mindq $pc$ (right{-$q$-} (R $(qd, c)$))
  where
    mindp (L $(j, pd)$) $qj$ = L ($j$, L$_2$ $(pd, \lrcorner qj)_2$)
    mindp (R $pc$)      $qj$ = mindq $pc$ (right{-$q$-} (L $qj$))
    mindq $pc$ (L $(j, qd)$) = L ($j$, R$_2$ $(\angle pc, qd)_2$)
    mindq $pc$ (R $qc$)      = R $(pc, qc)_1$
```

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.

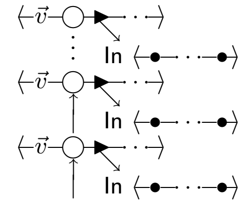
```
tmap :: $\Delta p \mapsto \hat{p} \Rightarrow (s \to t) \to p\ s \to p\ t$
tmap $f$ $ps$ = continue (right (L $ps$)) where
  continue (L $(s, pd)$) = continue (right (R $(pd, f\ s)$))
  continue (R $pt$)      = $pt$
```

These programs may seem fiddly, but in fact they're remarkably easy to write because they're precisely typed and abstract. By making clowns and jokers separate parameters, we distinguish them from other data and from each other. The types attract us towards the programs which make sense, a virtuous tendency which could only be strengthened by improving access to 'live' type information during the editing process.

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, $\phi :: p\ v \to v$, 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 :: $\Delta p \mapsto \hat{p} \Rightarrow (p\ v \to v) \to \mu\, p \to v$
tcata $\phi$ $t$ = load $\phi$ $t$ [ ]
```

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

```
load :: $\Delta p \mapsto \hat{p} \Rightarrow (p\ v \to v) \to \mu\, p \to [\hat{p}\ v\ (\mu\, p)] \to v$
load $\phi$ (In $pt$) $stk$ = next $\phi$ (right (L $pt$)) $stk$
```

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

```
next :: $\Delta p \mapsto \hat{p} \Rightarrow (p\ v \to v) \to$
          $(\mu\, p, \hat{p}\ v\ (\mu\, p)) + p\ v \to [\hat{p}\ v\ (\mu\, p)] \to v$
next $\phi$ (L $(t, pd)$) $stk$ = load $\phi$ $t$ $(pd : stk)$
next $\phi$ (R $pv$)      $stk$ = unload $\phi$ $(\phi\ pv)$ $stk$
```

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

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

```
unload :: $\Delta p \mapsto \hat{p} \Rightarrow (p\ v \to v) \to v \to [\hat{p}\ v\ (\mu\, p)] \to v$
unload $\phi$ $v$ $(pd : stk)$ = next $\phi$ (right (R $(pd, v)$)) $stk$
unload $\phi$ $v$ [ ]          = $v$
```

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 :: $\mu$ ExprP → Int
eval = tcata $\phi$ where
  $\phi$ (ValP $i$)       = $i$
  $\phi$ (AddP $v_1$ $v_2$) = $v_1 + v_2$
```

By design, dissection captures the notion of state for the left-to-right transformation of a (necessarily finite) container-like structure. As a consequence, a stack of dissections captures the state of the natural recursion over finite trees built from such containers, turning its control structure into data. I can imagine a number of motivations for doing this, besides mathematical curiosity.

Firstly, you might be programming with recursive data structures in a resource-aware setting, such as that of Hofmann and Jost (2003). By turning the control structure into data, you eliminate stack in favour of heap, bringing the necessary resources under control of the type system. If you build the stack by consuming the input and the output by consuming the stack, you might arrive at a rationalised reconstruction of the *pointer reversal* technique for traversing trees without fear of stack overflow.

Secondly, you might want your traversal process to support suspension and resumption. Dissection makes the state of a traversal into first-class data, bringing the schedule of the computation under much finer control.

Thirdly, and perhaps most interestingly, your program might benefit from manipulating its control structures more directly. For example, you might handle exceptional values by discarding a chunk of stack, rather than propagating them layer by layer. In effect, more possibilities open when your control structure is not only first-class, but also *first-order*. Filliâtre's work on 'backtracking iterators' (2006), also closely connected with zippers, shows interesting possibilities in this direction.

5. Derivative Derived by Diagonal Dissection

If we're interested in the possibility to manipulate first-order representations of contexts, it seems appropriate to revisit the zipper. In his seminal functional pearl, Huet (1997) not only shows how to represent one-hole contexts as stack-like structures, but also how to navigate efficiently around a tree decomposed as the pair of a zipper and a subtree in focus. In this section, I'll examine the way the *derivative* of a functor, now a special case of dissection, gives rise to the zipper datatype (McBride 2001; Abbott et al. 2005b). Moreover, I'll show how the dissection's explicit left-to-right analysis delivers Huet-style navigation.

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. As we've already seen, the rules for computing dissections just refine the centuries-old rules of the differential calculus with this left-right distinction. We can undo this refinement by taking the diagonal of the dissection, identifying clowns with jokers.

$$\partial p\ x = \Delta p\ x\ x$$

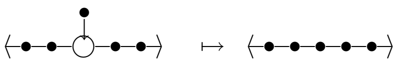
Let us now develop the related operations.

5.1 Plugging In

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

```
plug :: x → p̂ x x → p x
```

saying, in effect, that if clowns and jokers coincide, we can fill the hole directly, with no need to traverse and replace, all the way to the end.



The implementation is straightforward. As with right, the comments show you which instance declaration should receive each clause.

```
plug{-K1 a-} x (K2 z) = refute 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,\qx)2)  = (plug{-p-} x pd, qx)1
plug{-p ×1 q-} x (R2 (∠px, qd)2) = (px, plug{-q-} x qd)1
```

5.2 Zipping Around

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

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

I leave zLeft as an exercise, to follow your implementation of the leftward step operation, but the other three are straightforward uses of plug and right. 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 t pd), pds)

zDown (In pt, pds) = case right (L pt) of
  L (t, pd) → Just (t, pd : pds)
  R _       → Nothing

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

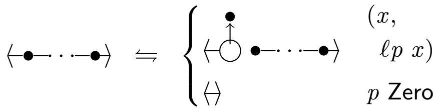
Notice that I had to give the typechecker a little help in the definition of zRight. The trouble is that $\Delta\cdot$ is not *invertible*. When we say right (R (pd, t)), the type of pd is given by some $\hat{p}$ which does not actually determine the corresponding p, and thence the appropriate instance of $\Delta\cdot \mapsto \hat{p}$. I've forced the issue by collecting p from the type of the input tree and using it to fix the type of the 'all clowns' failure case emerging from the appeal to right, thus forcing the selection of the $\Delta p \mapsto \hat{p}$ instance in a less than perspicuous manner. I wish I didn't have to be this devious, but there is currently no direct notation for me to be explicit about which instance I want—it must be *inferred*!

6. Division: No Clowns!

I originally stumbled into dissection whilst trying to 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 Zero, represents those ps with no elements at all. We can see this choice as follows:



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

$$\ell p\ x = \Delta p\ \mathsf{Zero}\ x$$

For the polynomials, we shall certainly have

```
divide :: Δp ↦ p̂ ⇒ p x → (x, p̂ Zero x) + p Zero
divide px = right (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[6]

```
inflateFst :: Bifunctor p ⇒ p Zero y → p x y
inflateFst = bimap refute id
```

Now, we may invert divide as follows:

```
unite :: Δp ↦ p̂ ⇒ (x, p̂ Zero x) + p Zero → p x
unite (L (x, pl)) = plug x (inflateFst pl)
unite (R pz)      = inflate pz
```

It is straightforward to show that divide and unite are mutually inverse by induction on polynomials.

To see why dissection is a necessary precursor to division, think about dividing a *composition*. 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. For example,

$$(\mathsf{Id\ Nothing}, \mathsf{Id}\ (\mathsf{Just}\ x))_1 :: (\mathsf{Id} \times_1 \mathsf{Id})\ (\mathsf{Maybe}\ x)$$

has its leftmost element in the second component. We need to be able to express the idea that we have only *q-leaves* left of the p-hole, which could be anywhere, but with different stuff to the left and to the right. By generalising to dissection, we get the correct behaviour for composition—the *chain rule*.

```
Δ(p ∘1 q) = Δq ×2 (Δp) ∘2 (∠q;\q)
```

where

```
data (p ∘2 (q; r)) c j = (p (q c j) (r c j)) ∘2 (·;·)
```

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

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

which exactly captures the 'leaves left of the hole' intuition. Let us now put division and dissection to work!

7. Generic Generalisation

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 in the course of a recursive computation. The task is to implement a generalisation mechanism, transforming an expression by replacing all occurrences of a given subexpression by a variable. This is a common technique in proof by induction: generalisation strengthens inductive hypotheses. The Coq proof assistant, for example, has a tactic for generalsation. Let us now

[6] Again, in some systems inflateFst can effectively be replaced by a cast.

develop an efficient generalisation algorithm for 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. Following Bird and Paterson (1999), we can use Maybe x to represent a variable set which distinguishes a new, bound variable Nothing from old, free variables Just x. Let us rename Maybe to S, 'successor', for this purpose. We may readily eliminate the bound variable by instantiating it with a term constructed over the free variables, as follows:

```
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
```

Generalisation can now be seen as the task of computing the 'most abstract' inverse to $(\lfloor s)$. That is, for suitable p and x, we need some

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

such that $(t \upharpoonright s) \lfloor s = t$, and moreover that fmap Just s occurs *nowhere* in $t \upharpoonright s$. 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 $t \upharpoonright s =$ fmap Just t is definitely wrong!

7.2 Indiscriminate Stop-and-Search

The obvious approach to computing $t \upharpoonright s$ is to traverse t checking everywhere if we've found s.

```
(⌈) :: (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 at the root. I write ≡ for Haskell's Boolean equality test (== in ASCII), which is available for types in the Eq class. Only if the match fails do we fall through and try to search more deeply inside the term.

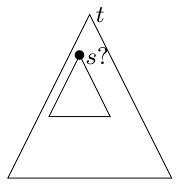
How do we know we can test equality of terms? We first must confirm that our signature functor p *preserves* equality, i.e., that we can lift equality eq on x to equality $\cdot \lceil eq \rceil \cdot$ on $p\ x$.

```
instance (PresEq p, Eq x) ⇒ Eq (p* x) where
  V x  ≡ V y  = x ≡ y
  C ps ≡ C pt = ps ⌈≡⌉ pt
  _    ≡ _    = False
```

Lifting equality is quite mechanical. The only interesting case is for sums, where structural difference is actually possible.

```
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, q₁)₁ ⌈eq⌉ (p₂, q₂)₁ = p₁ ⌈eq⌉ p₂ ∧ q₁ ⌈eq⌉ q₂
```

Our first attempt at generalisation 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. We search for s's root everywhere, whether or not its leaves reach the edge, as shown below.



It's rather like testing if a list 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.

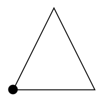
```
hasSuffix :: Eq x ⇒ [x] → [x] → Bool
hasSuffix xs ys = hasPrefix (reverse xs) (reverse ys)
hasPrefix :: Eq x ⇒ [x] → [x] → Bool
hasPrefix xs     []                 = True
hasPrefix (x:xs) (y:ys) | x ≡ y     = hasPrefix xs ys
hasPrefix _      _                  = False
```

7.3 Hunting for a Needle in a Stack

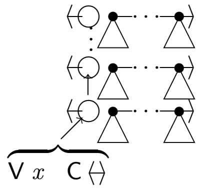
We can adapt the 'reversal' idea to our more arboreal problem. The divide function tells us how to find the leftmost position in a polynomial container, if it has one. By iterating divide, we can navigate our way down the left spine of a term to its leftmost leaf, stacking the contexts as we go. That's 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 :: ⩓p ↦ 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'—the leftmost leaf with a straight spine of leftmost holes running all the way back to the root, as shown schematically, and in detail:

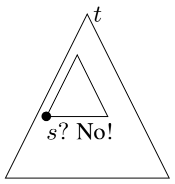


given by

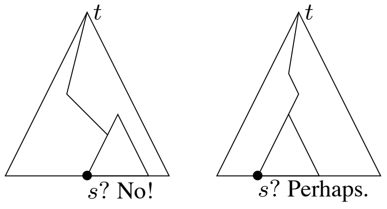


```
needle :: ◭p ↦ p̂ ⇒ p* x → (Leaf p x, [p̂ Zero (p* x)])
needle t = grow t [] where
  grow t pls = case leftOrLeaf t of
    L (t′, pl) → grow t′ (pl : pls)
    R l        → (l, pls)
```

Given this 'needle' representation of the search term, we can implement the abstraction as a stack-driven traversal, hunt (below), which tries for a match only when it reaches a suitable leaf. As our point of focus corresponds to a leaf in s, it's now easy to rule out internal positions in t:



Moreover, 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.

```
(↾) :: (◭p ↦ 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 = · · ·
```

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

```
hunt :: p* x →                        -- term in focus
        [ℓp (p* x)] →                 -- local spine
        [◭p (p* (S x)) (p* x)] →      -- stack to root
        p* (S x)
```

Now, check is rather like hasPrefix, except that I've used a lazy accumulator to ensure that the expensive equality tests for the rest of the term are evaluated only as soon as we know that the spine is at least as long as the needle.

```
check spi′ [] True =
  next (V Nothing) (fmap inflateFst spi′ ++ stk)
check (spl : spi′) (npl : nees′) b =
  check spi′ nees′ (b ∧ spl ⌈refute |≡⌉ npl)
check _ _ _ = next (leafS l) (fmap inflateFst 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 refute 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 ⌈refute⌉ 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—to stitch, just inflate the spine to a stack, then append. However, if the test fails, we must move right from the current *leaf*'s position, injecting it into $p^*\,(\mathsf{S}\ x)$ and stitching the original spine to the stack.

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. Dissection is the structure which supports *navigation* of first-order, first-class contexts, for more flexible management of both data and control in a purely functional setting. In my other work—implementing typecheckers for interactive programming environments—dissection-based control structures are invaluable in managing what is effectively a process of term traversal interruptable non-locally by fresh information at any time.

On a more theoretical note, what's clearly missing here is a *semantic* characterisation of dissection, with respect to which the operational rules for ◭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 induced notion of division can be used to calculate power series representations for data structures, establishing a significant connection with the notion of *combinatorial species* as studied by Joyal (1986) and others. The details, however, deserve a paper of their own.

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

◭[] = ◣[] ×₂ ◥[]

Moreover, if p has a dissection, it is an interesting exercise to construct the dissection of its free monad p^*. However, if we want to address more complex phenomena, such as mutually recursive datatypes (requiring multiple parameters) or iterated dissection (representing multiple holes), we shall rapidly reach the limits of the Haskell techniques I've shown here. But it's a delight that Haskell allows us to come even this close to implementing the general pattern, rather than its individual instances.

In principle, and in dependently typed practice, the whole development extends readily to the *multivariate* case. The general Δ_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 generalises Brzozowski's (1964) notion of the 'partial derivative' of a regular expression, with the set of sorts corresponding to the alphabet.

But if there is a broader message for programmers and programming language designers here, 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!

Acknowledgments

I wish to thank EPSRC for funding this research through grants EP/C512022/1 and EP/C511964/1. This work has benefited considerably from the criticism and advice of many people. I should like to express my warmest gratitude to Thorsten Altenkirch, Lucas Dixon, Peter Hancock, Dan Piponi, Tarmo Uustalu, Philip Wadler, and the anonymous referees.

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.

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, pages 8–19. ACM, 2003.

William Aitken and John Reppy. Abstract value constructors. Technical Report TR 92-1290, Cornell University, 1992.

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.

Richard Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–92, 1999.

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

Jean-Christophe Filliâtre. Backtracking iterators. Technical Report 1428, CNRS-LRI, January 2006.

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.

Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 185–197. ACM, 2003.

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.

Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, Amsterdam, The Netherlands, June 1997.

Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, January 1989.