# Pull-Ups, Push-Downs, and Passing It Around

## Exercises in Functional Incrementalization

Sean Leather[1], Andres Löh[1], and Johan Jeuring[1,2]

[1] Utrecht University, Utrecht, The Netherlands
[2] Open Universiteit Nederland
{leather,andres,johanj}@cs.uu.nl

**Abstract.** Programs in functional programming languages with algebraic datatypes are often datatype-centric and use folds or fold-like functions. Incrementalization of such a program can significantly improve its performance. Functional incrementalization separates the recursion from the calculation and significantly reduces redundant computation. In this paper, we motivate incrementalization with a simple example and present a library for transforming programs using upwards, downwards, and circular incrementalization. We also give a datatype-generic implementation for the library and demonstrate the incremental zipper, a zipper extended with attributes.

## 1 Introduction

In functional programming languages with algebraic datatypes, many programs and libraries "revolve" around a collection of datatypes. Functions in such programs form the spokes connecting the datatypes in the axle to a convenient API or EDSL at the perimeter, facilitating the movement of development. These *datatype-centric* programs can take the form of games, web applications, GUIs, compilers, databases, etc. Many libraries are datatype-centric: see finite maps, sets, queues, parser combinators, and zippers. Datatype-generic libraries with a structure representation are also datatype-centric.

When programmers develop datatype-centric programs, we observe that they write a surprising number of functions that can be defined using a fold (a.k.a. catamorphism) or a fold-like function such as an accumulation [7]. As a primitive form of recursion, a fold traverses an entire value using an algebra to combine the fields of constructors and results of subcomputations. In Haskell, we can define the class of folds using a type class and the related class of algebras as a type family.

```
type family Alg t s :: *
class Fold t where
  fold :: Alg t s → t → s
```

Given some algebra for the type t, the instance of fold for t recursively builds s-type results upward from the leaves of the (finite) value. Here is an example for binary trees.

```
data Tree a = Tip | Bin a (Tree a) (Tree a)
type instance Alg (Tree a) s = (s, a → s → s → s)
instance Fold (Tree a) where
  fold      (t, _) Tip          = t
  fold alg@(_, b) (Bin x tL tR) = b x (fold alg tL) (fold alg tR)
```

Folds are a well-understood class of functions and are occasionally used in repetition, unfortunately, to the detriment of the program's performance. For example, take the pattern of fold use in following function.

```
repFold :: (Fold t) ⇒ Alg t s → (t → t) → t → (s, s)
repFold alg f x = let {s = fold alg x ; x' = f x ; s' = fold alg x'} in (s, s')
```

Given a function and an initial value, repFold applies a fold to x and x'. If x' only differs from x in a "small" way (relative to the size of the value), then the second fold performs a large number of redundant computations. A fold is an *atomic computation*: it computes the results "all in one go." Even in a lazily evaluated language, there is no sharing between the computations of the two folds.

Our solution is to transform an atomic computation such as repFold into an *incremental computation*. Incremental computations take advantage of small changes to an input to compute a new output. The key is to subdivide an computation into smaller parts and reuse the subresults to compute the final output.

Our focus in this article is the *incrementalization* of purely functional programs with folds and fold-like functions. In general, incrementalization is the transformation of a program from having atomic computations to having the same results computed incrementally. To incrementalize a program with a fold, we separate the the application of the algebra from the recursion. We merge the algebra with the constructors and replace the single recursive function with the the recursion already present in other functions of the program.

The presentation of our work begins in Section 2 with a motivating example for incrementalization: we take a well-known library, incrementalize it, and compare the performance. Then, in Section 3, we generalize the work from Section 2—which we call "upwards" incrementalization—and produce a library of tools for incrementalization. Sections 4 and 5 develop two alternative forms of incrementalization, "downwards" and "circular." We generalize even further in Section 6 to show that the tools from the previous three sections can also be used with a fully datatype-generic representation of datatypes. In Section 7, we describe the incremental zipper, a structure than can take an incrementalized datatype to a zipper, supporting navigation and edit functionality while simultaneously preserving incrementality. Lastly, we round up with a general discussion and some related work in Section 8 and conclude in Section 9.

## 2 A Motivating Example

We introduce the library Set as a basis for understanding incrementalization. Starting from a simple, naive implementation, we systematically transform it to a more efficient, incrementalized version.

The Set library has the following programming interface.

```
empty     :: Set a                       insert   :: (Ord a) ⇒ a → Set a → Set a
singleton :: a → Set a                   fromList :: (Ord a) ⇒ [a] → Set a
size      :: Set a → Int                 fold     :: (s, a → s → s → s) → Set a → s
```

The interface is comparable to the Data.Set library provided with the Haskell Platform.

Semantically, a value of Set a is a "container" of a-type elements such that each element is unique. The Set type is abstract to the programmer, and we (the library developers) may change the implementation as we see fit. We implement the underlying data structure as an ordered, binary search tree [2].

```
type Set a = Tree a
```

We have several options for constructing sets. Simple construction is performed with empty and singleton (trivially defined with the constructors of Tree), and larger sets can be built from arbitrary lists of elements.

```
fromList = foldl (flip insert) empty
```

The function fromList uses another function from our interface, insert, which builds a new set given an old set and an additional element. This is where the ordering aspect is used.

```
insert x Tip           = singleton x
insert x (Bin y tL tR) = case compare x y of
                           LT → balance y (insert x tL) tR
                           GT → balance y tL            (insert x tR)
                           EQ → Bin       x tL          tR
```

We use the balance[1] function to maintain an invariant that the the structure of the tree always has logarithmic access time to any element.

```
balance x tL tR | sL + sR ⩽ 1 = Bin x tL tR
                | sR ⩾ 4 ∗ sL  = rotateL x tL tR
                | sL ⩾ 4 ∗ sR  = rotateR x tL tR
                | otherwise    = Bin x tL tR
  where sL = size tL
        sR = size tR
```

This function uses the size of each subtree to determine how to rotate nodes between subtrees. The functions rotateL and rotateR are not important for this discussion, but note that they use the sizes of even deeper subtrees to determine the number of nodes to rotate. We implement size using the instance of the *Fold* class defined earlier for the Tree datatype.

[1] It is not important for our purposes to understand the details of balancing a binary search tree. We refer the reader to [2] for details on the design.

```
sizeAlg = (0, λ_ s_L s_R → 1 + s_L + s_R)
size    = fold sizeAlg
```

The Set library presented here seems reasonable; however, once the programmer starts using it, she quickly realizes that it is quite slow. The primary issue is the repetitive use of size. As we have seen, size is used in balance, rotateL, and rotateR. Realizing that size is defined as a fold leads us to deduce that we have a situation like that of repFold. It is especially depressing that size is computing a result for subtrees immediately after computing the result for the enclosing parent. These are redundant computations, and the subresults should be reused. In fact, size is an atomic function that is ideal for incrementalization.

The key point to first realize is that we want to store intermediate results of computations on Tree values. We start by allocating space for storage.

```
data Tree a = Tip Int | Bin Int a (Tree a) (Tree a)
```

We need to keep around the result of a fold, and the logical locations are the recursive points of the datatype. In other words, we annotate each constructor with an additional field to contain the size of that Tree value. As a result of this transformation, the function size can be redefined to extract the annotation.

```
size (Tip s      ) = s
size (Bin s _ _ _) = s
```

The next step is to implement the part of the fold that applies the algebra to a value. To avoid obfuscating our code, we create an API for Tree values by lifting the structural aspects to a type class.

```
class Tree_S t a | t → a where
  tip      :: t
  bin      :: a → t → t → t
  caseTree :: t → r → (a → t → t → r) → r
```

The following instance of $Tree_S$ permits us to use the smart constructors tip and bin for introduction and the method caseTree instead of the **case** expression for elimination.

```
instance Tree_S (Tree a) a where
  tip            = Tip (fst sizeAlg)
  bin x t_L t_R  = Bin (snd sizeAlg x (size t_L) (size t_R)) x t_L t_R
  caseTree n t b = case n of {Tip _ → t ; Bin _ x t_L t_R → b x t_L t_R }
```

We have separated the components of sizeAlg and merged them with the constructors, in effect creating an initial algebra that computes the size.

For the finishing touch, we adapt the implementation to use our new datatype and functions.

```
insert x t = caseTree t (singleton x)
                 $ λy t_L t_R → case compare x y of
                                    LT → balance y (insert x t_L) t_R
```

```
GT → balance y tL       (insert x tR)
EQ → bin     x tL       tR
```

The refactoring is not difficult, but we need to verify that we achieved our objective: speed-up of the Set library.

To benchmark our work, we compare two implementations of fromList, one with an atomic size and the other with the incrementalized library. We constructed a Set from the list of words for each of three text inputs of increasing word count.

| | 5911 | 16523 | 26234 words |
|---|---|---|---|
| Atomic | 0.56 | 2.79 | 8.63 seconds |
| Incremental | 0.02 | 0.06 | 0.10 |

It is evident from these results that incrementalization had a significant effect on the performance of the insert function.

In the next section, we generalize the steps taken to incrementalize Set, so that we can concretely define the components of incrementalization. With the generalized approach, we no longer directly transform a program, but instead develop a library of tools for incrementalization.

## 3 Generalizing to Upwards Incrementalization

In this section, we take the process used in Section 2 to incrementalize the Set library and create some reusable components and techniques that can be applied to incrementalize another program. We call the approach used in this section *upwards incrementalization*, because we draw the results upward through the tree-like structure of an algebraic datatype value.

The first step we took in Section 2 was to allocate space for storing intermediate results. As mentioned, the logical locations for storage are the recursive points of a datatype. That leads us to identify the fixed-point view as a natural representation.

```
newtype Fix f = In {out :: f (Fix f)}
```

The type Fix encapsulates the recursion of some functor type f and allows us access to each recursive point. We use a second datatype to extend the fixed-point with storage for the results.

```
data Ext1 s f r = Ext1 s (f r)
```

The type $Ext_1$ pairs a single intermediate result called an *annotation* with a functor. Combined, Fix and $Ext_1$ give us an extended fixed-point representation.

```
type Fix1 s f = Fix (Ext1 s f)
```

We supplement this with some helper functions.

```
in₁  :: s → f (Fix₁ s f) → Fix₁ s f
out₁ :: Fix₁ s f → f (Fix₁ s f)
ann  :: Fix₁ s f → s
```

The representation of Tree is now split into the type-specific functor (with the appropriate *Functor* instance) and the extended fixed-point.

```
data TreeF a r = TipF | BinF a r r
type TreeU a   = Fix₁ Int (TreeF a)
```

With $\mathsf{Tree_U}$ in this form, the size function is now synonymous with ann.

The $\mathsf{Fix_1}$ representation allows us to generalize more aspects of incrementalization, namely the approach to implementing folds. We exchange the previous algebra type, Alg, for $\mathsf{Alg_U}$ and the *Fold* class for $\mathit{Incr}_U$.

```
type family AlgU (f :: * → *) s :: *
class IncrU f where
  pullUp :: AlgU f s → f s → s
```

Interestingly, this is the same approach often seen in the datatype-generic literature for the fixed-point fold. We are capturing only the part of the fold that applies the algebra. We leave the recursion to the remainder of the functions in the library that perform it (e.g. insert). The instances for $\mathsf{Tree_F}$ are straightforward.

```
type instance AlgU (TreeF a) s = (s, a → s → s → s)
instance IncrU (TreeF a) where
  pullUp (t, _) TipF           = t
  pullUp (_, b) (BinF x sL sR) = b x sL sR
```

Note that the type instance $\mathsf{Alg_U}$ ($\mathsf{Tree_F}$ a) s) has the same type as the instance Alg (Tree a) s. Also, the pullUp instance for $\mathsf{Tree_F}$ is similar to the fold instance for Tree, excluding the recursive calls.

The $\mathsf{Fix_1}$ type now comes back into the picture to construct incrementalized values. The function $\mathsf{in_U}$ takes an algebra and an unannotated functor, extracts the annotations from the children, applies the algebra to get an annotation, and pairs the annotation with the functor.

```
inU :: (IncrU f, Functor f) ⇒ AlgU f s → f (Fix₁ s f) → Fix₁ s f
inU alg fx = in₁ (pullUp alg (fmap ann fx)) fx
```

The construction of $\mathsf{Tree_U}$ values can be hidden, as in Section 2, using the $\mathit{Tree}_S$ class.

```
instance TreeS (TreeU a) a where
  tip            = inU sizeAlg TipF
  bin x tL tR    = inU sizeAlg (BinF x tL tR)
  caseTree n t b = case out₁ n of {TipF → t ; BinF x tL tR → b x tL tR}
```

Conveniently, the previous definition of insert does not need to change.

The work of this section gives a successful generalization of upwards incrementalization. We can package the components shown into a library and use that library to incrementalize datatype-centric programs. However, unlike many libraries that provide an API (e.g. Set), for incrementalization to be effective, the user should use the functor representation instead of the standard algebraic datatype. That is primarily why incrementalization is a transformation and not a collection of functions. Fortunately, functors such as $\mathsf{Tree}_\mathsf{F}$ (and their *Functor* instances) can be automatically generated with tools like Template Haskell [16], and we have shown how to define a type class to mask the usage of the library.

In the next two sections, we diverge from redefining Set to discuss other forms of incrementalization. The algebras will be different, but we continue to use the $\mathsf{Tree}_\mathsf{F}$ type for concrete examples.

## 4 Downwards Incrementalization

There are other directions that incrementalization can take. We have already demonstrated upwards incrementalization which involves passing values from the children to the parent. The obvious dual is *downwards incrementalization*, passing values from parent to children. In this direction, we accumulate the result of calculations using information from the ancestors of a node.

As with the upwards direction, the result of incremental computations is stored as an annotation. To distinguish between the two, we borrow some language from attribute grammars [11]: a downwards annotation is *inherited* by the children while an upwards annotation is *synthesized* for the parent.

We introduce the algebra type $\mathsf{Alg}_\mathsf{D}$ and type class $\mathit{Incr}_D$ for defining downwards incrementalization.

```
type family Alg_D (f :: * → *) i :: *
class Incr_D f where
  pushDown :: Alg_D f i → f s → i → f i
```

The intention of pushDown is that we take an inherited value from the parent, apply the algebra, and pass on the results to the children. The function only needs the structure of the container argument, not the values, so we give the elements an "unknowable" type s.

Following the example in Section 3 with the tree functor, we define its downwards instances as follows. The algebra is defined as a tuple of functions in which each function takes as parameters all of the non-recursive fields plus an incremental value from the parent.

```
type instance Alg_D (Tree_F a) i = (i → Tree_F a i, a → i → Tree_F a i)
```

The pushDown function performs case analysis on a value and applies the appropriate component of the algebra to the fields of that constructor and the inherited value.

```
instance IncrD (TreeF a) where
  pushDown (t, _) TipF         = t
  pushDown (_, b) (BinF x _ _) = b x
```

The evaluation of pushDown alg fs i should be a value fi that is structurally equivalent to fs but with its recursive points filled with inherited values for the children.

Similar to the incremental construction $\mathsf{in}_U$, we define a function $\mathsf{in}_D$ for downwards incrementalization, but its requirements are somewhat different. We no longer pull synthesized values up, but rather push inherited values down. The function pushDown provides a functor of inherited values, and we need to merge this with a functor of $\mathsf{Fix}_1$ values (the argument to $\mathsf{in}_D$). This calls for a generic zipWith function. We might use any one of the datatype-generic programming libraries, but to make this article complete, we will use a type class.

```
class ZipWith f where
  zipWith :: (a → b → c) → f a → f b → f c
```

We can now define $\mathsf{in}_D$.

```
inD :: (IncrD f, ZipWith f) ⇒ AlgD f i → i → f (Fix1 i f) → Fix1 i f
inD alg ini fx = in1 ini (zipWith push (pushDown alg fx ini) fx)
  where push i = inD alg i ∘ out1
```

Besides the obvious uses of pushDown over pullUp and zipWith over fmap, there are several other differences from $\mathsf{in}_U$: the annotation comes directly from an initial inherited value, and the constructor is modified via recursive applications of pushDown. We will return to these points momentarily, but let us first see an example of downwards incrementalization.

A simple example is calculating the depth of each node from the root. We can define an atomic depths function on Tree to do this.

```
depths :: Tree a → Tree Int
depths Tip             = Tip
depths (Bin _ tL tR) = Bin 1 (fmap (+1) (depths tL)) (fmap (+1) (depths tR))
```

The incrementalization of depths needs an algebra and an initial value.

```
depthAlg = (const TipF, λx i → let i' = 1 + i in BinF x i' i')
depthIni  = 1
```

We use $\mathsf{in}_D$ in the smart constructors of the instance of $\mathit{Tree}_S$.

```
instance TreeS (Fix1 Int (TreeF a)) a where
  tip            = inD depthAlg depthIni TipF
  bin x tL tR    = inD depthAlg depthIni (BinF x tL tR)
  caseTree n t b = case out1 n of {TipF → t ; BinF x tL tR → b x tL tR}
```

The instance of *ZipWith* for $\mathsf{Tree}_F$ necessary for $\mathsf{in}_D$ is trivial to define. To demonstrate how to access the downwards incremental values, we define a look-up function for the depth of a particular element in a binary search tree such as we used for the Set library.

```
depthOf k t = caseTree t Nothing
                $ λx t_L t_R → case compare k x of
                                  EQ → Just (ann t)
                                  LT → depthOf k t_L
                                  GT → depthOf k t_R
```

Returning to points raised before the example, we should highlight the use of recursion in $\mathsf{in}_D$. As we claimed earlier, the goal of incrementalization is to improve efficiency for some kinds of computation, and that remains true for downwards incrementalization. Algebraic datatypes are naturally constructed in a bottom-up manner, but $\mathsf{in}_D$ requires pushing the computation down the tree, thus resulting in rebuilding the entire functor argument. We can avoid this redundancy by memoizing $\mathsf{in}_D$ on the inherited value.

There are several options for memoization from which we might choose. GHC supports a rough form of global memoization using stable name primitives [13]. Generic tries may be used for purely functional memo tables [8] in lazy languages. In general, the best choice for memoization is strongly determined by the algebra used, but the options above present potential problems when used with incrementalization. They create a memo table for every node in a tree, and this can lead to an undesirable space explosion. For example, if the memo table at every node in the depth example contains two entries, then the size of the incrementalized value is triple the size of an unincrementalized one. To avoid potential space issues, we implement memoization with a table size of one and an equality check. The following re-definition of $\mathsf{in}_D$ introduces the memoization in the local function push.

```
in_D :: (Incr_D f, ZipWith f, Eq i) ⇒ Alg_D f i → i → f (Fix_1 i f) → Fix_1 i f
in_D alg ini fx = in_1 ini (zipWith push (pushDown alg fx ini) fx)
  where push i x | i == ann x = x
                 | otherwise = in_D alg i (out_1 x)
```

Note that top-level calls to $\mathsf{in}_D$ are not memoized, because they always construct new values.

The downwards direction puts an interesting about-face on purely functional incrementalization. Combining downwards with upwards incrementalization leads us to another interesting twist: circular incrementalization.

## 5 Circular Incrementalization

Circular incrementalization merges the functionality of upwards and downwards incrementalization to allow for much more interesting algebras. Incremental values may not only pass from the children to the parent but also in the reverse direction. We also add the possibility for the upwards result of the root to be used to produce the initial downwards value. Even the downwards result of each leaf is fed into the upwards incremental function. The path of values thus creates a circle of dependencies.

We are merging the functionality of the two previous sections, so we first merge their representation. We now annotate the fixed-point representation with both inherited and synthesized values.

```
data Ext₂ i s f r = Ext₂ i s (f r)
type Fix₂ i s f   = Fix (Ext₂ i s f)
```

Here are some functions (defined in the obvious way) that will be useful as we explore circular incrementalization.

```
in₂   :: i → s → f (Fix₂ i s f) → Fix₂ i s f
out₂  :: Fix₂ i s f → f (Fix₂ i s f)
syn   :: Fix₂ i s f → s
inh   :: Fix₂ i s f → i
```

The new algebra type and the type class specifying its application also merge the declarations of Sections 3 and 4.

```
type family Alg_C (f :: * → *) i s :: *
class Incr_C f where
  passAround :: Alg_C f i s → f s → i → (s, f i)
```

The function passAround joins the parameters of pullUp :: $\mathsf{Alg_U}$ f s → f s → s and pushDown :: $\mathsf{Alg_D}$ f i → f s → i → f i and tuples the outputs. Unlike in pushDown, here we use the synthesized values of the children as we did with pullUp. The instance for $\mathsf{Tree_F}$ will help illuminate the purpose of $\mathit{Incr_C}$.

```
type instance Alg_C (Tree_F a) i s = (i → (s, Tree_F a i), a → s → s → i → (s, Tree_F a i))
instance Incr_C (Tree_F a) where
  passAround (t, _) Tip_F              = t
  passAround (_, b) (Bin_F x s_L s_R) = b x s_L s_R
```

The $\mathsf{Alg_C}$ instance for $\mathsf{Tree_F}$ is defined by tupling both the parameters and the results of the previous instances of $\mathsf{Alg_U}$ and $\mathsf{Alg_D}$ (followed by currying the parameters). It is important for circularity that the leaf node of a value (here the $\mathsf{Tip_F}$) always have a function of the form i → s in its algebraic component. Defining the passAround function is straightforward given the convenient arrangement of the algebraic components.

To construct values, we adapt the definitions of $\mathsf{in_U}$ and $\mathsf{in_D}$ into a new function, $\mathsf{in_C}$, that accumulates both synthesized and inherited values.

```
in_C :: (Incr_C f, Functor f, ZipWith f, Eq i)
       ⇒ Alg_C f i s → (s → i) → f (Fix₂ i s f) → Fix₂ i s f
in_C alg top fx = in₂ ini s (zipWith push fi fx)
  where ini     = top s
        (s, fi) = passAround alg (fmap syn fx) ini
        push i x | i == inh x  = x
                 | otherwise = in_C alg (const i) (out₂ x)
```

The expression ini = top s handles the wrapping of the upwards result to the downward path at the root level, but we do not pass this function down. Instead, we use const i to pass only the inherited result downwards. By following the uses of synthesized and inherited values in $\mathsf{in}_\mathsf{C}$, we see the inherent *circular programming* [4]. A circular program uses lazy evaluation to avoid multiple traversals, and this is key to allowing us to define circular incrementalization.

Circular incrementalization lets us solve more interesting problems than allowed by either previous form of incrementalization. Among these problems is the "repmin" problem [4] used to introduce circular programming, but we shall solve a problem that has been used to show why attribute grammars matter. Wouter Swierstra [17] describes the issue of writing an efficient and readable program to calculate the difference of each value in a list from the average of all values in the list. The naive and inefficient implementation gives a precise meaning to the idea.

```
diff :: [Float] → [Float]
diff xs = let avg ys = sum ys / genericLength ys in map (λx → x − avg xs) xs
```

Swierstra implements this function both with an efficient though complex definition in Haskell and with a simpler attribute grammar specification for the UUAG system. We can translate the specification directly to a circular incremental algebra for $\mathsf{Tree}_\mathsf{F}$ (or any other functor).

We translate the attributes into two parts: annotations and an algebra. In the declarations of the synthesized and inherited annotations, we give the types along with useful names.

```
data    Diff_S = DS {sum_s :: Float, size_s :: Float, diff_s :: Float}
newtype Diff_I = DI {avg_i :: Float}
```

The synthesized annotations include $\mathsf{sum}_\mathsf{s}$ for the total value of all the Floats, $\mathsf{size}_\mathsf{s}$ for the total count, and the final result of the computation, $\mathsf{diff}_\mathsf{s}$. The one inherited value, $\mathsf{avg}_\mathsf{i}$, is the same value computed by avg above. We translate the semantic functions of the attributes to the following algebra.

```
diffAlg = (t, b)
  where t _          = (DS {sum_s = 0, size_s = 0, diff_s = 0}, Tip_F     )
        b x s_L s_R i = (s                                    , Bin_F x i i)
          where s = DS {sum_s  = x + sum_s s_L + sum_s s_R
                       , size_s = 1 + size_s s_L + size_s s_R
                       , diff_s = x − avg_i i}
```

For the synthesized annotations, the initial values are given in the $\mathsf{Tip}_\mathsf{F}$ component, and the computations in the $\mathsf{Bin}_\mathsf{F}$ component. The definitions for $\mathsf{sum}_\mathsf{s}$, $\mathsf{size}_\mathsf{s}$, and $\mathsf{diff}_\mathsf{s}$ are all as expected, and the inherited values are passed onto the children without modification. The top-level function computes the average using the synthesized sum and size.

```
diffFun s = DI {avg_i = sum_s s / size_s s}
```

As usual, we define a $\mathit{Tree}_S$ instance for the smart constructors.

```
instance Tree_S (Fix2 Diff_I Diff_S (Tree_F Float)) Float where
  tip            = inc_C diffAlg diffFun Tip_F
  bin x t_L t_R  = inc_C diffAlg diffFun (Bin_F x t_L t_R)
  caseTree n t b = case out2 n of { Tip_F → t ; Bin_F x t_L t_R → b x t_L t_R }
```

Now, we can define the incrementalized version of diff.

```
diff_C :: (Tree_S (Fix2 i Diff_S f) a, Tree_S t Float) ⇒ Fix2 i Diff_S f → t
diff_C n = caseTree n tip (λ_ t_L t_R → bin (diffOf n) (diff_C t_L) (diff_C t_R))
```

Of course, if we only need to get the difference for any one node, we only need to use diffOf.

```
diffOf :: Fix2 i Diff_S f → Float
diffOf = diff_s ∘ syn
```

This function would be a more efficient use of incrementalization, since it involves less reconstruction of values.

This section concludes our look at the various forms of incrementalization. In the next section, we explore a more generic representation. In Section 7, we delve into an interesting use of incrementalized values, the zipper.

## 6 Datatype-Generic Incrementalization

We have shown that incrementalization can be generalized from the approach used in Set to components provided by a library and usable for any datatype represented as a fixed-point. With circular incrementalization for example, the library user must provide instances of *Incr*$_C$, *Functor*, and *ZipWith*. In fact, we can generalize even further to the point where the user must only provide a representation of the datatype. To do this, we use *pattern functors*.

Pattern functors are functor types that represent the structure of a datatype with recursive points. They are ideal for defining generic functions such as folds, rewriting [19], and the zipper. As we have seen, incrementalization generalizes well with a fixed-point view, so pattern functors are a natural extension.

We use the following pattern functor datatypes.

```
newtype K a     r = K a
newtype I       r = I r
data    U       r = U
data    (f :*: g) r = f r :*: g r
data    (f :+: g) r = L (f r) | R (g r)
```

These represent the constant types (K), recursive locations (I), nullary constructors (U), constructor fields (:*:), and alternatives (:+:). To model the Tree structure with pattern functors, we can define an instance of *Tree*$_S$.

```
type Tree_PF a = U :+: K a :*: I :*: I
instance Tree_S (Fix (Tree_PF a)) a where
```

```
tip            = In (L U)
bin x t_L t_R  = In (R (K x :*: I t_L :*: I t_R))
caseTree n t b = case out n of {L U → t ; R (K x :*: I t_L :*: I t_R) → b x t_L t_R}
```

To get incrementalization, we must be able to use $\mathsf{in}_\mathsf{U}$, $\mathsf{in}_\mathsf{D}$, or $\mathsf{in}_\mathsf{C}$ here instead of In.

As it turns out, incrementalization of pattern functors is straightforward. We simply define instances of any of the incremental classes that we want. There are too many to list, but here are a few interesting instances for circular incrementalization[2].

```
type instance Alg_C I i s = s → i → (s, i)
instance Incr_C I where
  passAround f (I x) = second I ∘ f x
type instance Alg_C (I :*: g) i s = s → i → (Alg_C g i s, i)
instance (Incr_C g) ⇒ Incr_C (I :*: g) where
  passAround f (I x :*: y) i = let (g, i') = f x i
                                   (s, gi) = passAround g y i
                               in  (s, I i' :*: gi)
```

Note that the $\mathsf{Alg}_\mathsf{C}$ I instance for recursive points does not return the structure. Unlike the instance for $\mathsf{Alg}_\mathsf{C}$ ($\mathsf{Tree}_\mathsf{F}$ a), this is a guarantee we have when working with pattern functors: the types define (and abstract over) the structure, so we only compute the inherited and synthesized values.

Suppose that we wanted to incrementalize the type $\mathsf{Tree}_\mathsf{PF}$ with the diff algebra as we did before. The new $\mathsf{diffAlg}_\mathsf{PF}$ can use the same types, but the structure of the components must account for the structure of the representation. Notably, the single function b in diffAlg is now a function $\mathsf{b}_\mathsf{L}$ that generates a function $\mathsf{b}_\mathsf{R}$.

```
diffAlg_PF = (t, b_L)
  where b_L x s_L i_L = (b_R, i_L)
          where b_R s_R i_R = (s, i_R)
        ...
```

The functions $\mathsf{b}_\mathsf{L}$ and $\mathsf{b}_\mathsf{R}$ are associated with recursive points in the R alternative of $\mathsf{Tree}_\mathsf{PF}$.

To complete the incrementalization of the pattern functor representation, we need only define the expected instance of $\mathit{Tree}_S$; however, we must still fulfill some obligations in order to do that. Recall that $\mathsf{in}_\mathsf{C}$ requires the datatype to have an instance of *Functor* and *ZipWith*. The pattern functor instances for these classes and the instance of $\mathit{Tree}_S$ for $\mathsf{Fix}_2$ $\mathsf{Diff}_\mathsf{I}$ $\mathsf{Diff}_\mathsf{S}$ ($\mathsf{Tree}_\mathsf{PF}$ Float) are not difficult to define. We leave them as an exercise for the reader.

With all of the aforementioned type class instances in our library, we can now more easily incrementalize a program with datatypes represented as pattern functors. The library user only needs to provide or generate the representation.

---

[2] We use the function second :: (*Arrow* a) ⇒ a b c → a (d, b) (d, c) for convenience.

## 7 The Incremental Zipper

One interesting extension to the story on incrementalization is the design of an incremental zipper. The zipper [9] is a technique for navigating and editing a value of an algebraic datatype. We can make the zipper even more useful by incrementalizing it, allowing values to be computed incrementally as we navigate and edit the zipper. For this section, we rely significantly on the definition of the zipper as defined in [14], though we define ours for only one type, not a family of mutually recursive types. To save space, we attempt to present only what is new here.

To introduce the zipper, we first present the type class *Zipper* for functors.

```
class (Functor f) ⇒ Zipper f where
  fill  :: Ctx f r → r → f r
  first :: (r → Ctx f r → a) → f r → Maybe a
  next  :: (r → Ctx f r → a) → Ctx f r → r → Maybe a
```

The instances of *Zipper* and uses of its methods follow much the same pattern described in [14]. For example, fill is used when navigating up the zipper to plug the hole created by the context, Ctx f r, with the current value of the focus, r. Likewise, first is used for going down, and next for going right.

The context is defined as a type-indexed datatype whose constructors combine to create a derivative of the index type [12].

```
data family Ctx (f :: ∗ → ∗) :: ∗ → ∗
```

The instances of the Ctx can be defined directly from [14].

A zipper is usually referenced by its current location. The location is traditionally defined as an expression (the *focus*) and a collection of one-hole contexts.

```
data Loc :: ∗ → ∗ → (∗ → ∗) → ∗where
  Loc :: (Zipper f, Incr_C f, ZipWith f, Eq i)
        ⇒ Alg_C f i s → (s → i) → Fix_2 i s f → [Ctx f (Fix_2 i s f)] → Loc i s f
```

For the incremental zipper, we use the generalized algebraic datatype Loc with constraints that ensure that we have the proper instances defined (while not revealing them to casual observers for convenience). The fields of the Loc constructor include the focus as an incrementalized fixed-point and the list of contexts. In order to support incrementalization, Loc also stores the same algebra and top-level function used by $\mathsf{in}_C$.

In our incremental zipper library, we define the following functions for constructing, navigating, and editing zippers.

```
enter :: (Zipper f, Incr_C f, ZipWith f, Eq i)
          ⇒ Alg_C f i s → (s → i) → Fix_2 i s f → Loc i s f
leave :: Loc i s f → Fix_2 i s f
up    :: Loc i s f → Maybe (Loc i s f)
down  :: Loc i s f → Maybe (Loc i s f)
```

```
right    :: Loc i s f → Maybe (Loc i s f)
update :: (Functor (Ctx f)) ⇒ (Fix₂ i s f → Fix₂ i s f) → Loc i s f → Loc i s f
```

Most of these functions can be defined by translation from [14]. One, however, requires a more thorough look. The function update requires not only that we update the focus (as is evident from the signature), but that we must also update the contexts. In circular incrementalization, values are passed via functions from the bottom to the top of the value and vice versa. Consequently, a local change in a subexpression may result in a larger change to the entire value.

In order to modify the value in the definition of update, we need to update the annotations of the contexts as we do with the focus. Recall the type of passAround.

```
passAround :: (Incr_C f) ⇒ Alg_C f i s → f s → i → (s, f i)
```

Unfortunately, we cannot simply use the same algebra $\mathsf{Alg_C}$ on the context that we have on the focus. Instead, we treat each context as a functor with a hole. The top-down view of the context produces an s and takes a single i-value. The view from the bottom appears takes an s-value and produces an i. The result is a function $s \to i \to (s, i)$ for a context. We extend the *Zipper* class with methods for the above two operations.

```
class (Functor f) ⇒ Zipper f where
  ...
  fill_s  :: (Zipper g) ⇒ Ctx f (Fix₂ i s g) → s → f s
  seek_i :: Ctx f s → f i → i
```

The first function, $\mathsf{fill_s}$, performs a similar task to fill, except that it fills the recursive points with the synthesized annotation instead of the focus. The second, $\mathsf{seek_i}$, performs a zip-like search of a context and i-filled focus to find the hole in the context. With the hole, we know which inherited value to give to the focus. Combining these functions produces the following.

```
digest :: (Zipper f, Zipper g, Incr_C f) ⇒ Alg_C f i s → Ctx f (Fix₂ i s g) → s → i → (s, i)
digest alg c s = second (seek_i c) ∘ passAround alg (fill_s c s)
```

The update function uses digest when traversing the list of contexts, pushing synthesized values to the parent (the next context) and inherited values to the children (both the previous context and enclosed subexpressions).

# 8 Discussion and Related Work

In this section, we discuss various aspects of incrementalization as well as compare to related work.

## 8.1 Form of Incrementalizable Functions

The form of incrementalization that we have presented allows us to transform a program (esp. a datatype-centric one) with functions defined in certain ways into a program with the same functions defined incrementally. For upwards incrementalization, that "certain way" is the fold. We gave the example of size defined as a fold and transformed it. For downwards and circular incrementalization, defining the form of the non-incrementalized function requires looking at Gibbons' accumulations [7].

Downward accumulations, for example, pass information from the root to the leaves. They can be written using the paths function, declared as a type class.

```
data family Thread (f :: * → *) a :: *
class (Functor f) ⇒ Paths f where
  paths :: f a → f (Thread f a)
```

The paths function replaces every element in a container with a thread containing that element and the path back to the root. A value of the type-indexed datatype Thread mirrors a path from the root to a node of the type f. We can see this in the Thread instance for Tree (as defined in Section 1).

```
data instance Thread Tree a = TT a | TL (Thread Tree a) a | TR (Thread Tree a) a
```

The downwards accumulation function is written using paths and a *Fold* instance for the Thread instance.

```
accum_D :: (Fold (Thread f a), Paths f) ⇒ Alg (Thread f a) s → f a → f s
accum_D alg = fmap (fold alg) ∘ paths
```

In order to use $\mathsf{accum}_D$, we need the algebra for the thread. Given an algebra for the Tree thread, e.g. $(a \to s, s \to a \to s, s \to a \to s)$, we can easily rewrite depths from Section 4 as a downwards accumulation. Similarly, many functions that can be written as accumulations can also be incrementalized and vice versa.

### 8.2 Attribute Grammars

In some ways, incrementalization appears similar to attribute grammars [11]. The attributes for nodes in a value are defined by the algebra for that value's type, and Fokkinga, et al [6] prove that attribute grammars can be translated to folds. This similarity is the reason we used the terms "inherited" and "synthesized" for the annotations.

Saraiva, et al [15] demonstrate incremental attribute evaluation for a purely functional implementation of attribute grammars. They transform syntax trees to improve the performance of incremental computation. Our approach is considerably more "lightweight" since we write our programs directly in the target language (e.g. Haskell) instead of using a grammar or code generation. On the other hand, we lack the significant boost to performance available to them by rewriting the syntax tree.

Viera, et al [20] describe first-class attribute grammars in Haskell. Their approach ensures the well-formedness of the grammar and allows for combining attributes using type-level programming. Our approach to combining attributes is more ad-hoc and we do not ensure well-formedness; however, we believe our approach is much simpler to understand and implement. We also show that our techniques can improve the performance of a library.

### 8.3 Incremental Computing

Our initial interest in incremental computing was inspired by Jeuring's work on incremental algorithms for lists [10]. We show that incremental algorithms can also be defined not just on lists but on many algebraic datatypes.

Carlsson [5] translates an imperative ML library supporting high-level incremental computations [1] into a monadic library for Haskell. His approach relies on references to store intermediate results and requires explicit specification of the incremental components. In contrast, our approach uses the structure of the datatype to determine where annotations are placed, and we can hide the incrementalization using techniques such as smart constructors and type classes such as $\mathit{Tree}_S$.

### 8.4 Incremental Zipper

In Section 7, we define a library for a generic incremental zipper. Uustalu and Vene [18] implement a zipper using comonadic functional attribute evaluation. Coming from the angle of dataflow, they arrive at a similar conclusion to ours; however, they neither identify the algebra of attributes nor describe a completely generic zipper.

Bernardy [3] defines a lazy, incremental, zipper-based parser for the text editor Yi. His implementation is rather specific to its purpose and lacks an apparent generalization to other datatypes. Further study is required to determine whether Yi can take advantage of an incremental zipper as we have shown.

## 9 Conclusion

We have presented a number of exercises in purely functional incrementalization using Haskell and datatype-generic programming. Incrementalizing programs decouples recursion from computation and storing intermdediate results. Thus, we remove redundant computation and improve the performance of some programs. By utilizing the fixed-point structure of algebraic datatypes, we demonstrate a library that captures all the elements of incrementalization for folds and accumulations. We have also introduced the incremental zipper, a library that can be used with incrementalized datatypes to support incremental computation while editing a value.

**Acknowledgements** Edward Kmett wrote a blog entry in response to one of our own with some insightful observations. This research has been partially funded by the Netherlands Organization for Scientific Research (NWO), through the project on "Real-life Datatype-Generic Programming" (612.063.613).

## References

1. Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *POPL 2002*, pages 247–259. ACM, 2002.

2. Stephen Adams. Functional Pearls: Efficient sets – a balancing act. *J. of Functional Programming*, 3(04):553–561, 1993.
3. Jean-Philippe Bernardy. Lazy Functional Incremental Parsing. In *Haskell 2009*, pages 49–60. ACM, 2009.
4. R. S. Bird. Using Circular Programs to Eliminate Multiple Traversals of Data. *Acta Informatica*, 21(3):239–250, October 1984.
5. Magnus Carlsson. Monads for incremental computing. In *ICFP 2002*, pages 26–35. ACM, 2002.
6. Maarten M. Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. A Translation from Attribute Grammars to Catamorphisms. 2(1):20–26, 1991.
7. Jeremy Gibbons. Upwards and downwards accumulations on trees. In *MPC 1993*, pages 122–138, 1993.
8. Ralf Hinze. Memo functions, polytypically! In Johan Jeuring, editor, *WGP 2000: Proceedings of the Second Workshop on Generic Programming*, July 2000.
9. Gérard Huet. The Zipper. *J. of Functional Programming*, 7(05):549–554, 1997.
10. Johan Jeuring. Incremental algorithms on lists. In Jan van Leeuwen, editor, *Proc. of the SION Conference on Computer Science in the Netherlands*, pages 315–335, 1991.
11. Donald E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2):127–145, June 1968.
12. Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. 2001.
13. Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in Haskell. pages 37–58. 2000.
14. Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic Programming with Fixed Points for Mutually Recursive Datatypes. In *ICFP 2009*, pages 233–244. ACM, 2009.
15. João Saraiva, S. Doaitse Swierstra, and Matthijs Kuiper. Functional Incremental Attribute Evaluation. pages 279–294. 2000.
16. Tim Sheard and Simon L. Peyton Jones. Template Meta-programming for Haskell. In *Haskell 2002*, pages 1–16. ACM, 2002.
17. Wouter Swierstra. Why Attribute Grammars Matter. *The Monad.Reader*, 4, July 2005.
18. Tarmo Uustalu and Varmo Vene. Comonadic functional attribute evaluation. *Trends in Functional Programming 6*, pages 145–162, 2007.
19. Thomas van Noort, Alexey Rodriguez Yakushev, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A lightweight approach to datatype-generic rewriting. In *WGP 2008*, pages 13–24. ACM, 2008.
20. Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute Grammars Fly First-Class: How to do Aspect Oriented Programming in Haskell. In *ICFP 2009*, pages 245–256. ACM, 2009.