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 languages such as Haskell are often datatype-centric and make extensive use of folds on that datatype. Incrementalization of such a program can significantly improve its performance by transforming monolithic atomic folds into incremental computations. Functional incrementalization separates the recursion from the application of the algebra in order to reduce redundant computations and reuse intermediate results. In this paper, we motivate incrementalization with a simple example and present a library for transforming programs using upwards, downwards, and circular incrementalization. Our benchmarks show that incrementalized computations using the library are nearly as fast as handwritten atomic functions.

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 center to a convenient application programming interface (API) or embedded domain-specific language (EDSL) at the circumference, facilitating the development cycle. These *datatype-centric* programs can take the form of games, web applications, GUIs, compilers, databases, etc. We can find examples in many common libraries: finite maps, sets, queues, parser combinators, and zippers. Datatype-generic libraries with a structure representation are also datatype-centric.

Programmers developing datatype-centric programs often define recursive functions that can be defined with primitive recursion. A popular form of recursion, the *fold* (a.k.a. catamorphism or reduce), is (categorically) the unique homomorphism (structure-preserving map) for an initial algebra. That is, an algebraic datatype T provides the starting point (the initial algebra) for a fold to map a value of T to a type S using an algebra $\mathsf{F}\ \mathsf{S} \to \mathsf{S}$, all the while preserving the structure of the type T. Folds can be defined using an endofunctor F (another homomorphism), and an F-algebra. Given a transformation that takes a recursive datatype T to its base functor $\mathsf{F}\ \mathsf{T}$, we can define a fold that works

for any type T. Before we get too far outside the scope of this paper, however, let us put some concrete notation down in Haskell[1].

The base functor for a type t can be represented by a datatype family[1]:

```
data family F t :: * → *
```

A datatype family is a type-indexed datatype[2] that gives us a unique structure for each type t. The types t and F t should have the same structure (i.e. same alternatives and products) with the exception that the recursive points in t are replaced by type parameters in F (this being the reason for the kind $* \to *$). The isomorphism between t and F t is captured by the *InOut* type class.

```
class (Functor (F t)) ⇒ InOut t where
  inF  :: F t t → t
  outF :: t → F t t
```

An algebra for a functor f is defined as a function f s → s for some result type s. In the case of a fold, f is the base functor F t. We use the following type synonyms to identify algebras:

```
type Alg  f s = f s → s
type AlgF t s = Alg (F t) s
```

As mentioned above, the fold function is a structure-preserving map from a datatype to some value determined by an algebra. Using the above definitions and fmap from the *Functor* class, we implement fold as follows.

```
fold :: (InOut t) ⇒ AlgF t s → t → s
fold alg = alg ∘ fmap (fold alg) ∘ outF
```

A fold for a particular datatype T requires three instances: F T, *InOut* T, and *Functor* (F T). Here is the code for a binary tree.

```
data          Tree a   = Tip  | Bin  a (Tree a) (Tree a)
data instance F (Tree a) r = TipF | BinF a r        r
instance Functor (F (Tree a)) where
  fmap _ TipF          = TipF
  fmap f (BinF x rL rR) = BinF x (f rL) (f rR)
instance InOut (Tree a) where
  inF  TipF          = Tip
  inF  (BinF x tL tR) = Bin x tL tR
  outF Tip           = TipF
  outF (Bin x tL tR)  = BinF x tL tR
```

One F-algebra for Tree is calculating the number of binary nodes in a value.

[1] We use Haskell 2010 along with the following necessary extensions: MultiParamTypeClasses, TypeFamilies, FlexibleContexts, KindSignatures, and Rank2Types.

```
sizeAlg :: Alg_F (Tree a) Int
sizeAlg Tip_F           = 0
sizeAlg (Bin_F _ s_L s_R) = 1 + s_L + s_R
```

The simplicity of the above code[2] belies the power it provides: a programmer can define numerous recursive functions for Tree values using fold and an algebra instead of direct recursion. With that understanding, let us return to our story.

Folds are occasionally used in repetition, and this can negatively impact a program's performance. If we analyze the evaluation of such code, we might see the pattern in Fig. 1. The variables x_i are values of some foldable datatype and the results y_i are used elsewhere. If the functions f_i change their values 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 such as Haskell, there is no sharing between the computations of fold in this pattern.

In this article, we propose to solve the problem of repetitive folds by transforming repeated atomic computations into a single *incremental computation*. Incremental computations take advantage of small changes to an input to compute a new output. The key is to subdivide a computation into smaller parts and reuse previous results to compute the final output.

```
x1 <-- f0 x0
y1 <-- fold alg x1
x2 <-- f1 x1
y2 <-- fold alg x2
...
```

Fig. 1. Evaluation of repeated folds. The symbol <-- indicates that the right evaluates to the left.

Our focus is the *incrementalization*[4] of purely functional programs with folds and fold-like functions. To incrementalize a program with folds, we separate the application of the algebra from the recursion. We first merge the components of the F-algebra with the initial algebra (the constructors). We may optionally define smart constructors to simplify use of the transformed constructors. The recursion of the fold is then implicit, blending with the recursion of other functions.

We motivate our work in Section 2 by taking a well-known library, incrementalizing it, and looking at the improvement over the atomic version. In Section 3, we generalize this form of incrementalization—which we call "upwards"—into library form. Sections 4 and 5 develop two alternative forms of incrementalization, "downwards" and "circular." In Section 6, we discuss other aspects of incrementalization. We discuss related work in Section 7 and conclude in Section 8. All of the code presented in this paper is available online[3].

[2] As simple as they are, the F, *Functor*, and *InOut* instances may be time-consuming to write for large datatypes. Fortunately, this code can be generated with tools such as Template Haskell[3].

[3] `http://people.cs.uu.nl/andres/Incrementalization/`

2 Incrementalization in Action

We introduce the Set library 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 API.

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

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

One can think of a value of Set a as a container of a-type elements such that each element is unique. For the implementation, we use an ordered, binary search tree[5] with the datatype introduced in Section 1.

```
type Set = Tree
```

We have several options for constructing sets. Simple construction is performed with empty and singleton, which are trivially defined using Tree constructors. Sets can also be built from lists of elements.

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

The function fromList uses insert which builds a new set given an old set and an additional element. This is where the ordering aspect from the type class *Ord* 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 function to maintain the invariant that a look-up operation has logarithmic access time to any element.

```
balance :: a → Set a → Set a → Set a
balance x tL tR | sL + sR ⩽ 1 = Bin x tL tR
                | sR ⩾ 4 * sL = rotateL x tL tR (size tRL) (size tRR)
                | sL ⩾ 4 * sR = rotateR x tL tR (size tLL) (size tLR)
                | otherwise   = Bin x tL tR
  where (sL, sR)        = (size tL, size tR)
        Bin _ tRL tRR = tR
        Bin _ tLL tLR = tL
```

Here, we use the size of each subtree to determine how to rotate nodes between subtrees. We omit the details[4] of balance but call attention to how often the

[4] It is not important for our purposes to understand how to balance a binary search tree. The details are available in the code of Data.Set and other resources[5].

size function is called. We implement size using fold with the algebra defined in Section 1.

```
size = fold sizeAlg
```

The astute reader will notice that the repeated use of the size in balance leads to the pattern of folds described in Fig. 1. In this example, we are computing a fold over subtrees immediately after computing a fold over the 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 highlight is that we want to store results of computations with Tree values. We start by allocating space for storage.

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

We need to preserve the result of a fold, and the logical location is each recursive point of the datatype. In other words, we annotate each constructor with an additional field to contain the size of that $\mathsf{Tree}^{\mathsf{i}}$ value. This can be visualized as a Tree value with superscript annotations.

```
Bin^4 2 (Bin^1 1 Tip^0 Tip^0) (Bin^2 3 Tip^0 (Bin^1 4 Tip^0 Tip^0))
```

We then define the function $\mathsf{size}^{\mathsf{i}}$ to extract the annotation without any recursion.

```
size^i (Tip^i i)       = i
size^i (Bin^i i _ _ _) = i
```

The next step is to implement the part of the fold that applies the algebra to a value. To avoid obfuscation, we create an API for Tree values by lifting the structural aspects (introduction and elimination) to a type class.

```
class TreeS t where
  type Elem t
  tip      :: t
  bin      :: Elem t → t → t → t
  caseTree :: r → (Elem t → t → t → r) → t → r
```

An instance of Tree_S permits us to use the smart constructors tip and bin for introducing t values and the method caseTree (instead of **case**) for eliminating them. Since the element type depends on the instance type, we use the associated type Elem t to identify the values of the bin nodes. Applying this step to $\mathsf{Tree}^{\mathsf{i}}$, we arrive at the following instance.

```
instance TreeS (Tree^i a) where
  type Elem (Tree^i a) = a
  tip                  = Tip^i 0
  bin x t_L t_R        = Bin^i (1 + size^i t_L + size^i t_R) x t_L t_R
  caseTree t b n = case n of {Tip^i _ → t ; Bin^i _ 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 touches, we adapt the library to use the new datatype and $Tree_S$ instance. The refactoring is not difficult, and the types of all functions should be the same (of course, using $Tree^i$ instead of Tree). Refer to the associated code for the refactored functions. We have one last check to verify that we achieved our objective: speed-up of the Set library.

To benchmark[5] our work, we compare two implementations of the fromList function. The first is given by the definition above. The second is from the aforementioned refactored Set library using the $Tree^i$ type. For each run, we build a set from the words of a wordlist text file. The word counts increase with each input to give an idea how well fromList scales.

Fig. 2 lists the results. To collect these times, we evaluate the values strictly to head normal form. Since Haskell is by default a lazily evaluated language, these times are not necessarily indicative of real-world use; however, they do give the worst case time, in which the entire set is needed.

It is clear (and no surprise) from the results that incrementalization has a significant effect. We have changed the time complexity of the size calculation from linear to constant, thus reducing the time of fromList by nearly 100% for all inputs.

In this section, we developed a library from a naive implementation with atomic folds into an incrementalized implementation. This particular work is by no means novel, and no one would use the naive approach; however, it does identify a design pattern that may improve the efficiency of other programs. In the remainder of this article, we capture that design pattern in a library and explore other variations on the theme.

3 Upwards Incrementalization

We take the design pattern from the previous section and create reusable components and techniques that can be applied to incrementalize another program. We call the approach used in this section *upwards incrementalization*, because we pull the results upward through the tree-like structure of an algebraic datatype.

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 another datatype to extend a functor with a new field.

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

[5] All benchmarks were compiled with GHC 6.10.1 using the `-O2` flag and run on a MacBook with Mac OS X 10.5.8, a 2 GHz Intel Core 2 Duo, and 4 GB of RAM.

	5,911	16,523	26,234	*words*
Atomic	3.876	19.561	61.151	*seconds*
Incrementalized	**0.010**	**0.028**	**0.056**	

Fig. 2. Performance of the atomic and incrementalized fromList.

The type Ann pairs an *annotation* with a functor. Combined, Fix and Ann give us an annotated fixed-point representation.

```
type Fix^a s f = Fix (Ann s f)
```

We supplement this type with its base functor (along with instances of *Functor* and *InOut*) and functions to introduce (in^a), eliminate (out^a), and extract the annotation (ann) from Fix^a values.

```
data instance F (Fix^a s f) r = In^a_F s (f r)
in^a  :: s → f (Fix^a s f) → Fix^a s f
out^a :: Fix^a s f → f (Fix^a s f)
ann   :: Fix^a s f → s
```

Another function that will be useful later is $\mathsf{foldMap}^a$.

```
foldMap^a :: (Functor f) ⇒ (r → s) → Fix^a r f → Fix^a s f
foldMap^a f = fold (λ(In^a_F s x) → in^a (f s) x)
```

To continue with the example used in the Set library, we now represent the binary search tree using the base functor of Tree introduced in Section 1. We can define an alternative representation for Tree^i as Tree^U.

```
type Type^a s t = Fix^a s (F t)
type Tree^U a   = Type^a Int (Tree a)
```

We can use this type with an instance of Tree_S in the same way as before; however, we must first define a general form of upwards incrementalization.

Recall that our objective is to separate a fold into its elements: the application of the algebra and the recursion. First, let us determine how to get an annotated fixed-point type from a Haskell type; then, we can dissect the fold. Upwards incrementalization is specified by upwards.

```
upwards :: (InOut t) ⇒ Alg_F t s → t → Type^a s t
upwards = fold ∘ pullUp

pullUp :: (Functor f) ⇒ Alg f s → Alg f (Fix^a s f)
pullUp alg fs = in^a (alg (fmap ann fs)) fs
```

The function upwards is naturally defined with fold. The pullUp function transforms an algebra on a functor f with the result s to an algebra that results in the annotated fixed point Fix^a s f. It does this by mapping each annotated fixed point to its annotation in f. The in^a function (also an algebra) pairs the annotation with x. This value is built atomically (since upwards is a fold). To construct the same value incrementally, we define introduction and elimination operations under the Tree_S instance.

```
instance Trees_S (Tree^U a) where
  type Elem (Tree^U a) = a
  tip           = pullUp sizeAlg Tip_F
  bin x t_L t_R = pullUp sizeAlg (Bin_F x t_L t_R)
  caseTree t b n = case out^a n of {Tip_F → t ; Bin_F x t_L t_R → b x t_L t_R}
```

The primary differences from the $\mathsf{Tree}^{\mathsf{i}}$ instance are that we use the base functor constructors and that we wrap them with the algebra pullUp sizeAlg and unwrap them with the coalgebra $\mathsf{out}^{\mathsf{a}}$.

The library defined in this section allows programmers to write programs with upwards incrementalization. Given a datatype, the programmer defines an algebra that they want incrementalized. Using pullUp and the base functor of that datatype, the programmer can easily build incremental results. Smart constructors or a structure type class such as *Trees_S* are not required, but they can simplify the programming by hiding the complexities of incrementalization.

We now compare the performance of our generalized upwards incrementalization against the specialized incrementalization presented in Section 2. Since the incrementalized Set library was refactored to use the *Trees_S* class, we can use the same code for the generalized implementation but with the type $\mathsf{Tree}^{\mathsf{U}}$ instead of $\mathsf{Tree}^{\mathsf{i}}$. We used the same benchmarking methodology as before to collect the results in Fig. 3.

Surprisingly, the generalized fromList performs better, running 15 to 17% faster than the specialized version. It is not clear precisely why this is, though we speculate that it is due to the structure of the explicit fixed point and annotation datatypes.

An alternative benchmark is the time taken to build large values of each datatype. This is independent of any library and reflects clearly the impact of the incrementalization on construction. We compare the evaluation of building three isomorphic tree values: construction of the Tree datatype with "built-in" Haskell syntax, construction of the incrementalized $\mathsf{Tree}^{\mathsf{U}}$ type, and $\mathsf{Tree}^{\mathsf{U}}$ values transformed from constructed Tree values.

We arrive at the results shown in Fig. 4 using QuickCheck[6] to reproducibly generate each arbitrary value with the approximate size shown. Each time is the average over three different random seeds. As with previous comparisons, we evaluate to head normal form. To be consistent with later comparisons, the element type of the trees is Float.

The times of the comparison are virtually indistinguishable. This provides good indication that upwards incrementalization does not impact the performance of constructing values. Again, note that due to lazy evaluation, this only predicts the worst case time, not the expected time for incremental updates.

In the next two sections, we look at other variations of incrementalization. It could be said that upwards is the most "obvious" adaptation of folds; however, it is also limiting in the functions that can be written. Algebraic datatypes are tree-structured and constructed inductively; therefore, it is naturally that information flows upward from the leaves to the root. It is also possible to pass information downward from the root to the leaves as well as both up and down simultaneously. We venture into this territory next.

	5,911	16,523	26,234	*words*
Specialized	10.0	28.4	56.1	*milliseconds*
Generalized	**8.3**	**24.1**	**46.8**	

Fig. 3. Performance of the specialized and generalized fromList.

	1,000	10,000	100,000	*nodes*
Tree	7.4	39.7	137.3	*milliseconds*
Incrementalized with pullUp	**7.4**	**39.5**	**136.5**	
Transformed with upwards	7.4	39.6	137.4	

Fig. 4. Performance of constructing trees with upwards incrementalization using the size algebra.

4 Downwards Incrementalization

There are other directions that incrementalization can take. We have demonstrated upwards incrementalization, and in this section, we discuss its dual: *downwards incrementalization*. In this direction, we accumulate the result of calculations using information from the ancestors of a node. As with its upwards sibling, the result of downwards computations is stored as an annotation on a fixed-point value. To distinguish between the two, we borrow vocabulary from attribute grammars[7]: a downwards annotation is *inherited* by the children while an upwards annotation is *synthesized* for the parent.

Let us establish a specification of a fixed-point value with inherited annotations. To do this, we start with Gibbons' accumulations[8], in particular downwards accumulations. Accumulation is similar to incrementalization in the sense that information flows up or down the structure of the datatype. Accumulations collect this information in the polymorphic elements (e.g. the a in Tree a) while incrementalization collects it at the recursive points. Gibbons modeled downwards accumulation using paths, and we borrow this concept for downwards incrementalization.

A *path* is a route from a constructor in a value to the root of that value (i.e. the sequence of ancestors). The type of a path is characteristic of the datatype whose path we want, so we define Path as a type-indexed datatype. The Path instance for the Tree type helps clarify the structure a path.

```
data family Path t
data instance Path (Tree a)
  = PRoot | PBinL a (Path (Tree a)) | PBinR a (Path (Tree a))
data instance F (Path (Tree a)) r = PRootF | PBinLF a r | PBinRF a r
```

In downward accumulations, every element is replaced with the path from that constructor. Then, a fold is applied to each path to determine the result that is stored in the element. In downwards incrementalization, we annotate every constructor with its path. The primitives for this operation are defined by the *Paths* class and exemplified by the Tree instance.

```
class (InOut t, InOut (Path t), ZipWith (F t)) ⇒ Paths t where
  proot  :: Path t
  pnode  :: F t r → F t (Path t → Path t)
instance Paths (Tree t) where
  proot                = PRoot
  pnode Tip_F          = Tip_F
  pnode (Bin_F x _ _) = Bin_F x (PBin_L x) (PBin_R x)
```

The methods proot and pnode are used to link the constructors of Path t to the constructors of the type t. The mapping is quite straightforward: there is always one root constructor, and the remaining constructors match recursive nodes. The function paths uses the methods of *Paths* to annotate every recursive point with its path.

```
paths :: (Paths t) ⇒ t → Type^a (Path t) t
paths = app^a proot ∘ fold (in^a id ∘ zipApp comp^a pnode)
app^a x       = foldMap^a ($x)
comp^a f      = foldMap^a (∘f)
zipApp f g x = zipWith f (g x) x
```

In paths, we are folding over the type t with an algebra that again folds over the annotated value to push the latest known constructor to the bottom of each child path using function composition. We follow up with a second fold to apply the composed functions to proot. We use an instance of the class *ZipWith* to merge the recursive path nodes (that contain functions applying the constructor) with the original base function.

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

The instances of *ZipWith* are trivial. Alternatively, we might have used a datatype-generic programming library to zip functors together. To get an intuition of how paths works, refer to the following example.

```
Bin^PRoot 2 Tip^PBin_L 2 PRoot
  (Bin^PBin_R 2 PRoot 1 Tip^PBin_L 1 (PBin_R 2 PRoot) Tip^PBin_R 1 (PBin_R 2 PRoot))
```

Each constructor is initially annotated with the with id. As the outer fold works upwards, the inner fold compose the current path constructor (e.g. $\mathsf{PBin_L}$ or $\mathsf{PBin_R}$) with the results. Finally, the function annotation for every node is applied to the root PRoot.

We can now give a specification for inherited annotations.

```
downwards :: (Paths t) ⇒ Alg_F (Path t) s → t → Type^a s t
downwards alg = foldMap^a (fold alg) ∘ paths
```

We use paths to annotate all recursive points with their paths, and we fold over the annotated result with an algebra that contains a fold over a path. The path algebra is provided by the programmer. For example, suppose we want to calculate the depth of a constructor:

```
depthAlg^D p = case p of {PRoot_F → 1 ; PBin_LF _ i → succ i ; PBin_RF _ i → succ i}
```

Applying the depth algebra with a fold draws the information up from the bottom, but in the case of a path, the "bottom" is the root of the tree. In this way, inheritance is flipped head-over-heels.

Given the number of folds and redundant traversals of the fixed-point value and paths, the definition of downwards is clearly inefficient. We may, of course, improve its performance with manual optimizations, but in the end, it will still be a fold. Instead, we deviate from this in our approach for downwards incrementalization. The primary difference lies with the algebraic structure.

```
type Alg^D f i = forall s.i → f s → f i
type Alg^D_F t i = Alg^D (F t) i
```

This algebra gives us the point of view of a constructor in a recursive datatype. We no longer fold over a path, but rather inherit an i-type annotation, which would have been the result of a fold on the path, from the parent. The type f s → f i indicates that this algebra changes the elements of a functor f using the inherited value, and the explicit quantification over s (producing rank-2 polymorphism below) preserves the downward direction of data flow. We used a similar device in the type of pnode. Also similar to pnode, an Alg^D algebra must preserve the structure of the input.

We perform downwards incrementalization with an algebra transformation similar to pullUp. The function pushDown demonstrates some similarities with paths.

```
pushDown :: (ZipWith f) ⇒ i → Alg^D f i → Alg f (Fix^a i f)
pushDown init alg = in^a init ∘ zipApp push (alg init)
  where push i = pushDown i alg ∘ out^a
```

We use zipApp to merge altered and unaltered functor values. The altered values come by applying the initialized algebra. (In paths, we alter with pnode.) We have removed most folds in pushDown, but we cannot remove recursion completely. Finite values are constructed inductively (i.e. upward), yet we are pushing inherited annotations down to the children. If we construct a new Bin 1 x y value from two Bin values x and y, we must (in a sense) ensure that x and y receive their inheritance from their new parent.

The function pushDown takes a different algebra from downwards, but the difference between the algebra on paths and Alg^D requires manageable changes. The PRoot_F case is replaced by an initial inherited value, and the left and right Bin paths are replaced by a single Bin_F case. We rewrite depthAlg^D as the following:

```
depthInit     = 1
depthAlg i t = case t of {Tip_F → Tip_F ; Bin_F x _ _ → Bin_F x (succ i) (succ i)}
```

We can then use pushDown depthInit depthAlg to define the smart constructors—in an instance of $Tree_S$, perhaps—for downwards incrementalization.

Evaluating downwards incrementalization would ideally be done with a program that made use of it. We have observed, however, that it is not clear how useful downwards incrementalization is. We reuse the depth algebra from Gibbons[8] in our example, but we have found very few interesting algebras. As a matter of opinion, the incrementalization found in the next section appears much more useful. Indeed, perhaps downwards incrementalization serves better to introduce some concepts, in a simpler setting, that reappear in the next section. At the very least, this section serves to make the discussion of incrementalization more complete.

To evaluate the performance of downwards incrementalization, we benchmark the construction of tree values annotated with depth. We look at the same comparison for downwards as we did for upwards. The details are given in Fig. 5.

Downwards incrementalization is clearly less efficient than upwards incrementalization and constructing Tree values: 12 to 21% slower. The recursive downward push accounts for the extra time. On the other hand, it is encouraging to see that, at the worst case, downwards depth incrementalization is not too much slower. In general, the evaluation time of downwards incrementalization can vary greatly depending on the depth of the values and especially the algebra used. As an algebra, depth is perhaps detrimental to efficiency since every new node on top of the tree results in updates to every node down to the leaves. Lastly, note that the downwards transformation is 8 to 11% slower in evaluation. This difference indicates the time spent folding over the paths and the repetitive folds over the tree.

The downwards direction puts an interesting twist on purely functional incrementalization. The development of a Path and its use in downwards allows us to understand inherited annotations while the incrementalizing function pushDown provides a more efficient approach. In the next section, we look at the merger of upwards and downwards incrementalization.

5 Circular Incrementalization

Combining upwards and downwards incrementalization leads us to another form: *circular incrementalization*. Circular incrementalization merges the functionality of both to allow for much more interesting algebras. Information flows both from the descendants to the ancestors and vice versa. Circularity is achieved by introducing feedback at the leaves and the root such that the result of one direction of flow may influence the other. Fittingly, we annotate recursive points with both synthesized and inherited annotations. The following functions serve to access the annotations:

```
inh :: Fix^a (i, s) f → i
syn :: Fix^a (i, s) f → s
```

We illustrate circular annotations with a specification similar to upwards and downwards. Every annotation combines both synthesized data from the *subtree* (whose root is the annotated node) and inherited data from the *context* (the

	1,000	10,000	100,000	*nodes*
Tree	7.4	39.7	137.3	*milliseconds*
Incrementalized with pushDown	**8.3**	**46.6**	**167.1**	
Transformed with downwards	9.0	51.7	185.4	

Fig. 5. Performance of constructing trees with downwards incrementalization using the depth algebra.

entire tree-like value inverted, with a path from the current node to the root: the same concept used in zippers[9]). At each node, we can use the algebra to pass results either up or down or both. In our model, we build both subtrees and contexts for each node and compute each annotation with a fold. In order to create circularity, we use algebras whose results are functions that take the other annotation as an argument. We first describe contexts, and then we extend an annotated value with subtrees.

A context is an expansion of a path, and it can be defined in much the same way. We use the type-indexed datatype Context and the type class *Contexts* to give the structure of a datatype's context and the primitives for building it, respectively.

```
data family Context t
data instance Context (Tree a)
  = CRoot | CBin_L a (Context (Tree a)) (Tree a) | CBin_R a (Tree a) (Context (Tree a))
data instance F (Context (Tree a)) r
  = CRoot_F | CBin_LF a r (Tree a) | CBin_RF a (Tree a) r
class (InOut t, InOut (Context t), ZipWith (F t)) ⇒ Contexts t where
  croot  :: Context t
  cnode  :: F t t → F t (Context t → Context t)
instance Contexts (Tree a) where
  croot                = CRoot
  cnode Tip_F          = Tip_F
  cnode (Bin_F x t_L t_R) = Bin_F x (λc → CBin_L x c t_R) (CBin_R x t_L)
```

Note the differences from paths. A context traces a path to the root, but a Context value, unlike a Path, contains a node's sibling recursive values. To annotate all nodes with contexts, we use the following function:

```
contexts :: (Contexts t) ⇒ t → Type^a (Context t) t
contexts = app^a croot ∘ fold (in^a id ∘ zipApp comp^a (cnode ∘ fmap rm^a))
rm^a :: (InOut t) ⇒ Type^a s t → t
rm^a = fold (in_F ∘ out^a_F)
```

The only difference from paths is the need to fold the fixed point into its built-in representation for context nodes. This is in accordance with the fact that the only difference Context has from Path is the inclusion of sibling values represented with the Haskell type. Here is an example of a context-annotated value.

```
Bin^CRoot 2 Tip^CBin_L 2 CRoot (Bin 1 Tip Tip)
  (Bin^CBin_R 2 Tip CRoot 1 Tip^CBin_L 1 (CBin_R 2 Tip CRoot) Tip Tip^CBin_R 1 Tip (CBin_R 2 Tip CRoot))
```

Given a value with contexts, we pair each annotation with its subtree via a fold over a fixed-point value.

```
subtrees :: (InOut t) ⇒ Type^a c t → Type^a (c, t) t
subtrees = fold (λ(In_F^a c x) → in^a (c, in_F (fmap rm^a x)) x)
```

With subtree and context annotations, we can define circular annotations.

Circular annotations can be constructed with the following function:

```
circular :: (Contexts t)
         ⇒ Alg_F (Context t) (s → i) → Alg_F t (i → s) → t → Type^a (i, s) t
circular alg^D alg^U = foldMap^a cycle ∘ subtrees ∘ contexts
  where cycle (ct, st) = let (i, s) = (fold alg^D ct s, fold alg^U st i) in (i, s)
```

Two algebras are necessary: one for each context and one for each subtree. The result of each algebra is a function that is the inverse of the other. Subsequently, each fold is applied to the result of the other, using a technique called *circular programming*[10]. A circular program uses lazy evaluation to avoid multiple explicit traversals, and this is key to circular incrementalization. It allows us to define algebras that rely on as-yet-unknown inputs. Feedback occurs when the upwards algebra $\mathsf{alg}^{\mathsf{U}}$ takes input from the downwards algebra $\mathsf{alg}^{\mathsf{D}}$, and vice versa. It is possible to create multiple passes by defining multiple such dependencies. Of course, it is also possible to create non-terminating cycles, but we accept that chance in order to support expressive algebras.

Examples of problems that can be solved by circular incrementalization include the "repmin" problem from Bird[10] and the "diff" problem from Swierstra[11]. The latter is used to show why attribute grammars matter. Circular incrementalization shares some similarities with attribute grammar systems, so it is worth exploring this in more detail.

The naive implementation of Swierstra's problem is a function that, given a list of numbers, calculates the difference of each number from the average of the whole list and returns the results in a list.

```
diff :: [Float] → [Float]
diff xs = let avg = sum xs / genericLength xs in map (subtract avg) xs
```

Swierstra demonstrates two more efficient implementations: one is manually developed and significantly more complex, and the other is generated from a simpler attribute grammar specification using the UUAG system. We add to this an implementation using circular incrementalization, though our definition works on Tree values instead of lists.

The following types serve as specification for the annotations.

```
newtype Diff^i = DI {avg^i :: Float}
data    Diff^s = DS{sum^s :: Float, size^s :: Float, diff^s :: Float}
```

In the inherited annotation $\mathsf{Diff}^{\mathsf{i}}$, we have the only inherited value, the average. In the synthesized annotation $\mathsf{Diff}^{\mathsf{s}}$, we have the sum of all element values, the size or count of elements (genericLength for lists), and the resulting difference.

The algebra for subtrees establishes the synthesized annotation.

```
diffAlg^U :: Alg_F (Tree Float) (Diff^i → Diff^s)
diffAlg^U Tip_F          _ = DS{sum^s = 0, size^s = 0, diff^s = 0}
diffAlg^U (Bin_F x s_L s_R) i = dbin^s x (s_L i) (s_R i) i

dbin^s :: Float → Diff^s → Diff^s → Diff^i → Diff^s
dbin^s x s_L s_R i = 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}
```

In the Tip component, the values are initialized to zero. In the Bin component, we perform the operations: summing the elements, counting the number of elements, and computing the difference from the average. Since the elements of the algebra are functions, we apply s_L and s_R to the inherited value i, ultimately used in $diff^s$.

The algebra for contexts establishes the inherited annotation.

```
diffAlg^D :: Alg_F (Context (Tree Float)) (Diff^s → Diff^i)
diffAlg^D CRoot_F          s   = DI{avg^i = sum^s s / size^s s}
diffAlg^D (CBin_LF x i t_R) s_L = let j = i $ dbin^s x s_L (fold diffAlg^U t_R j) j in j
diffAlg^D (CBin_RF x t_L i) s_R = let j = i $ dbin^s x (fold diffAlg^U t_L j) s_R j in j
```

The $CRoot_F$ case holds the calculation of the average because the sum and size have been determined for the entire tree. The $CBin_{LF}$ and $CBin_{RF}$ cases determine the synthesized annotations with folds of the trees not included in the subtree. In these cases, we must also apply the upwards algebra effectively in reverse: the result is used by the algebra's function element i and passed onwards. Note that we must be sure to always use the final inherited annotation (j in these cases) in, for instance, fold $diffAlg^U$ t_R j. Otherwise, the average value does not arrive correctly at every node. Using j instead of i leads to more circular programming, but it does not lead to cycles since there is no other feedback route from $diffAlg^D$ into $diffAlg^U$.

Here is an example of a value annotated with the diff algebra.

```
Bin^DI 1.5,DS 3 2 0.5 2 Tip^DI 1.5,DS 0 0 0
  (Bin^DI 1.5,DS 1 1 (−0.5) 1 Tip^DI 1.5,DS 0 0 0 Tip^DI 1.5,DS 0 0 0)
```

The same average value has been inherited by every node. In the synthesized annotations, the Bin constructors have the appropriate sum, size, and difference annotations, and the Tip constructors have zeroes.

The specification of circular incrementalization above is clear but (of course) not efficient. To define an efficient version, as with the downwards approach, we use a different algebraic structure.

```
type Alg^C f i s = i → f s → (s, f i)
type Alg^C_F t i s = Alg^C (F t) i s
```

The Alg^C type expands upon Alg^D such that the synthesized annotations are available for use as well as passed on to the parent. Put another way, we marry

the Alg and $\mathsf{Alg}^{\mathsf{D}}$ types and bear $\mathsf{Alg}^{\mathsf{C}}$. The circular algebra is used in the following algebra transformation.

```
passAround :: (Functor f, ZipWith f) ⇒ (s → i) → Alg^C f i s → Alg f (Fix^a (i, s) f)
passAround fun alg fis = in^a (i, s) (zipWith pass fi fis)
  where i       = fun s
        (s, fi) = alg i (fmap syn fis)
        pass j  = passAround (const j) alg ∘ out^a
```

The function passAround borrows some aspects from pullUp and pushDown. From the former, we take the upwards algebra applied to the mapped synthesized results. As with the latter, we push the inherited annotations downward by zipping the structures together and recursing. But unlike either previous form, passAround also has circular dependencies on the annotations. The circularity of i and s works in the same way as circular: by enabling the algebra to implicitly traverse the structure and pass around annotations.

The attentive reader will notice that passAround supports restricted capabilities compared to circular. In passAround, we only have feedback from synthesized to inherited annotations at the top level, using the fun parameter. In the internal nodes, the fun argument is const j, meaning we simply pass the inherited value downwards. Since the definition of circular uses algebras with function results, feedback can happen at any node. This means that circular is more expressive; however, it also means that algebras for passAround are simpler to define. At this point, we do not see the need for the increased expressiveness of circular, but we do appreciate the simplified algebra of passAround.

We can solve the diff problem using the parameters of passAround. First, we define the top-level feedback function.

```
diffFun^C :: Diff^s → Diff^i
diffFun^C s = DI{avg^i = sum^s s / size^s s}
```

This is nothing more than the calculation of the inherited annotation. The circular algebra is also quite simple.

```
diffAlg^C :: Alg^C_F (Tree Float) Diff^i Diff^s
diffAlg^C _ Tip_F             = (DS{sum^s = 0, size^s = 0, diff^s = 0}, Tip_F)
diffAlg^C i (Bin_F x s_L s_R) = (dbin^s x s_L s_R i, Bin_F x i i)
```

Unlike with $\mathsf{diffAlg}^{\mathsf{D}}$, we are not concerned with deciding which inherited annotation to pass on, and we do not need any (additional) circularity. We may complete the circular incrementalization for Tree by defining a straightforward Tree_S instance using passAround $\mathsf{diffFun}^{\mathsf{C}}$ $\mathsf{diffAlg}^{\mathsf{C}}$ for the smart constructors.

We benchmark circular incrementalization in the same way as downwards with one addition. We manually define a fast, accumulating diff function (type Tree Float → Tree Float) that replaces each element with its difference. This function provides a basis for comparison with more typical atomic implementation. See Fig. 6 for results.

Circular incrementalization is 12 to 26% slower than downwards incrementalization. It also is 20 to 47% slower than the accumulation. The accumulation

	1,000	10,000	100,000	*nodes*
Tree	7.4	39.7	137.3	*milliseconds*
Accumulating diff	7.7	41.3	142.9	
Incrementalized with passAround	**9.2**	**53.4**	**210.5**	
Transformed with circular	30.6	737.1	17,233.4	

Fig. 6. Performance of constructing trees with circular incrementalization using the diff algebra.

is, of course, an atomic function and would incur costs when used again while the time for incrementalization would be amortized over repeated computations. The circular transformation is radically less efficient and would not be useful in practice.

This concludes our look at the various forms of incrementalization. In the following sections, we discuss aspects of incrementalization and related work.

6 Discussion

Several aspects of incrementalization deserve further discussion.

6.1 Combining Algebras

To simplify the presentation of incrementalization, we have ignored a potential issue. Suppose we have the function union for incrementalized trees. If we define the function for the Set library, the solution is the same as without incrementalization. On the other hand, we may define it more generally with the type $\mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t}$. But with the approach described in this paper, there is no guarantee that the annotation types in each parameter match the same algebra. For example, we might have a type Int for size and a type Int for sum. How do we merge these when we combine values from each parameter?

One option is to allow for different algebras and to use an additional algebra for pairing the annotations together. At every node, we compute the annotations for the algebras of each input. For union, this would result in a type $\mathsf{Type}^{\mathsf{a}}\ \mathsf{r}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ (\mathsf{r}, \mathsf{s})\ \mathsf{t}$. However, for some applications (such as the Set library) this may not be desirable, since it changes the annotation types.

An alternative option is to ensure that an algebra is unique for the entire program. We can do this by creating a multiparameter type class for annotations that attach a type s to a type t: $\mathit{Annot}\ \mathsf{s}\ \mathsf{t}$. We then define union to have the type $(\mathit{Annot}\ \mathsf{s}\ \mathsf{t}) \Rightarrow \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t} \to \mathsf{Type}^{\mathsf{a}}\ \mathsf{s}\ \mathsf{t}$. There can be only one instance of *Annot* for this pair of types, so the type system prevents an inconsistency between the parameters.

6.2 Optimization

One possibility for speeding up incrementalization is the use of *memoization* or storing the results of a function application in order to reuse them if the function is applied to the same argument. Typically, this technique involves a table mapping arguments to results. In both downwards and circular incrementalization,

pushing the inherited annotation down leads to recursion through the subtree structure, and this is a large time factor in the these techniques. To memoize the downwards function (push in pushDown and pass in passAround), we must create a memo table for the inherited annotation.

There are several options for memoization from which we might choose. GHC supports a rough form of global memoization using stable name primitives[12]. Generic tries may be used for purely functional memo tables[13] 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. Creating a memo table for every node in a tree can lead to an undesirable space explosion. For example, suppose the memo table at every node in the downwards depth example contains two entries. The size of the incrementalized value is already triple the size of an unincrementalized one.

To avoid space issues, we use a memo table size of one with an equality check. The function push is easily modified (as is pass, using inh instead of ann)

```
push i x | i == ann x = x
         | otherwise = pushDown i alg (out^a x)
```

In our experiments, unfortunately, memoization did not have a significant effect. The memoized pushDown was up to 1% slower with the depth algebra, and the memoized passAround was up to 1% faster with the diff algebra.

We leave it to future work to explore other forms of optimization.

6.3 Going Datatype-Generic

We have described incrementalization as a library with type classes for a programmer to instantiate. It can also serve as part of a datatype-generic programming library.

For example, for datatypes that are represented using *pattern functors* or structural functor types with explicit recursive points, we can define generic functions such as folds and zips to use with all such types. Such a library is described for rewriting[14] and mutually recursive datatypes[15]. Both *Functor* and *ZipWith* can be instantiated with such a library, so by extension, we can use the above definitions for pullUp, pushDown, and passAround. More details are available in a technical report[16].

6.4 Applications of Incrementalization

We continue to search for particular uses of incrementalization outside of the Set library, but one particular application appears very attractive: a generic incremental *zipper*. The zipper[9] is a technique for navigating and editing values of algebraic datatypes. By incrementalizing a zipper, annotations may be computed incrementally as we navigate and change a value. Examples where this would be useful include (partial) evaluation of an abstract syntax tree and online formatting of structured documents or code. We have outlined an implementation of a generic incremental zipper elsewhere[16].

7 Related Work

As we have mentioned, incrementalization is quite similar to attribute grammars[7]. In addition, Fokkinga, et al[17] prove that attribute grammars can be translated to folds. An annotation on a node in incrementalization is the result of computing an attribute on a production.

Saraiva, et al[18] 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 (i.e. Haskell) instead of using a grammar or code generation. On the other hand, we cannot easily boost performance by rewriting.

Viera, et al[19] describe first-class attribute grammars in Haskell. Their approach ensures the well-formedness of the grammar and allows for combining attributes in a type-safe fashion. Our approach to combining annotations is more ad-hoc and we do not ensure well-formedness; however, we believe our approach is simpler to understand and implement. We also show that our technique can improve the performance of a library.

Our initial interest in incremental computing was inspired by Jeuring's work on incremental algorithms for lists[20]. This work shows that incremental algorithms can also be defined not just on lists but on algebraic datatypes in general.

Carlsson[21] translates an imperative ML library supporting high-level incremental computations[22] 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 is purely functional and uses the structure of the datatype to determine where annotations are placed. We can also hide the incrementalization using type classes such as $Tree_S$. Incrementalization, however, is limited to computations that can be defined as folds, while Carlsson's work is more free-form.

Bernardy[23] 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 incrementalization.

8 Conclusion

We have presented a number of exercises in purely functional incrementalization. Incrementalizing programs decouples recursion from computation and storing intermediate 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.

Acknowledgments We thank Edward Kmett for an insightful blog entry and Stefan Holdermans and the anonymous reviewers for suggestions that contributed

to significant improvements. This work has been partially funded by the Netherlands Organization for Scientific Research through the project on "Real-Life Datatype-Generic Programming" (612.063.613).

References

1. Chakravarty, M.M.T., Keller, G., Peyton Jones, S.L., Marlow, S.: Associated Types with Class. In: POPL. (2005) 1–13
2. Hinze, R., Jeuring, J., Löh, A.: Type-indexed data types. Science of Computer Programming **51**(1-2) (2004) 117–151
3. Sheard, T., Peyton Jones, S.L.: Template Meta-programming for Haskell. In: Haskell. (2002) 1–16
4. Liu, Y.A.: Efficiency by Incrementalization: An Introduction. Higher-Order and Symbolic Computation **13**(4) (2000) 289–313
5. Adams, S.: Functional Pearls: Efficient sets – a balancing act. J. of Functional Programming **3**(04) (1993) 553–561
6. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: ICFP. (2000) 268–279
7. Knuth, D.E.: Semantics of context-free languages. Theory of Computing Systems **2**(2) (1968) 127–145
8. Gibbons, J.: Upwards and downwards accumulations on trees. In: MPC. (1993) 122–138
9. Huet, G.: The Zipper. J. of Functional Programming **7**(05) (1997) 549–554
10. Bird, R.S.: Using Circular Programs to Eliminate Multiple Traversals of Data. Acta Informatica **21**(3) (October 1984) 239–250
11. Swierstra, W.: Why Attribute Grammars Matter. The Monad.Reader **4** (2005)
12. Peyton Jones, S.L., Marlow, S., Elliott, C.: Stretching the storage manager: weak pointers and stable names in Haskell. In: IFL. (2000) 37–58
13. Hinze, R.: Memo functions, polytypically! In Jeuring, J., ed.: WGP. (2000)
14. van Noort, T., Rodriguez Yakushev, A., Holdermans, S., Jeuring, J., Heeren, B.: A lightweight approach to datatype-generic rewriting. In: WGP. (2008) 13–24
15. Rodriguez Yakushev, A., Holdermans, S., Löh, A., Jeuring, J.: Generic Programming with Fixed Points for Mutually Recursive Datatypes. In: ICFP. (2009) 233–244
16. Leather, S., Löh, A., Jeuring, J.: Pull-Ups, Push-Downs, and Passing It Around: Exercises in Functional Incrementalization. Technical Report UU-CS-2009-024, Dept. of Information and Computing Sciences, Utrecht University (November 2009)
17. Fokkinga, M.M., Jeuring, J., Meertens, L., Meijer, E.: A Translation from Attribute Grammars to Catamorphisms. The Squiggolist **2**(1) (1991) 20–26
18. Saraiva, J.a., Swierstra, S.D., Kuiper, M.: Functional Incremental Attribute Evaluation. In: CC. (2000) 279–294
19. Viera, M., Swierstra, S.D., Swierstra, W.: Attribute Grammars Fly First-Class: How to do Aspect Oriented Programming in Haskell. In: ICFP. (2009) 245–256
20. Jeuring, J.: Incremental algorithms on lists. In van Leeuwen, J., ed.: SION Conference on Computer Science in the Netherlands. (1991) 315–335
21. Carlsson, M.: Monads for incremental computing. In: ICFP. (2002) 26–35
22. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: POPL. (2002) 247–259
23. Bernardy, J.P.: Lazy Functional Incremental Parsing. In: Haskell. (2009) 49–60