Generic Functional Parallel Algorithms: Scan and FFT

CONAL ELLIOTT, Target, USA

Parallel programming, whether imperative or functional, has long focused on arrays as the central data type. Meanwhile, typed functional programming has explored a variety of data types, including lists and various forms of trees. *Generic* functional programming decomposes these data types into a small set of fundamental building blocks: sum, product, composition, and their associated identities. Definitions over these few fundamental type constructions then automatically assemble into algorithms for an infinite variety of data types—some familiar and some new. This paper presents generic functional formulations for two important and well-known classes of parallel algorithms: parallel scan (generalized prefix sum) and fast Fourier transform (FFT). Notably, arrays play no role in these formulations. Consequent benefits include a simpler and more compositional style, much use of common algebraic patterns and freedom from possibility of run-time indexing errors. The functional generic style also clearly reveals deep commonality among what otherwise appear to be quite different algorithms. Instantiating the generic formulations, two well-known algorithms for each of parallel scan and FFT naturally emerge, as well as two possibly new algorithms.

CCS Concepts: • **Theory of computation → Parallel algorithms**;

Additional Key Words and Phrases: generic programming, parallel prefix computation, fast Fourier transform

ACM Reference Format:
Conal Elliott. 2017. Generic Functional Parallel Algorithms: Scan and FFT. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (September 2017), 25 pages.
https://doi.org/10.1145/3110251

1 INTRODUCTION

There is a long, rich history of datatype-generic programming in functional languages [Backhouse et al. 2007; Magalhães and Löh 2012]. The basic idea of most such designs is to relate a broad range of types to a small set of basic ones via isomorphism (or more accurately, embedding-projection pairs), particularly binary sums and products and their corresponding identities ("void" and "unit"). These type primitives serve to connect algorithms with data types in the following sense:

- Each data type of interest is encoded into and decoded from these type primitives.
- Each (generic) algorithm is defined over these same primitives.

In this way, algorithms and data types are defined independently and automatically work together.

One version of this general scheme is found in the Haskell library *GHC.Generics*, in which the type primitives are *functor-level* building blocks [Magalhães et al. 2011]. For this paper, we'll use six: sum, product, composition, and their three corresponding identities, as in Figure 1. There are additional definitions that capture recursion and meta-data such as field names and operator fixity, but the collection in Figure 1 suffices for this paper. To make the encoding of data types easy, *GHC.Generics* comes with a generic deriving mechanism (enabled by the *DeriveGeneric* language extension), so that for regular (not generalized) algebraic data types, one can simply write "**data** ... **deriving** *Generic*"

2475-1421/2017/9-ART1
https://doi.org/10.1145/3110251

```
data      (f + g) a = L1 (f a) | R1 (g a) -- sum
data      (f × g) a = f a × g a           -- product
newtype   (g ∘ f) a = Comp1 (g (f a))     -- composition
data      V1      a                       -- void
newtype   U1      a = U1                  -- unit
newtype   Par1    a = Par1 a              -- singleton
```

Fig. 1. Functor building blocks

```
-- Representable types of kind ∗ → ∗.
class Generic1 f where
  type Rep1 f :: ∗ → ∗
  from1 :: f a → Rep1 f a
  to1   :: Rep1 f a → f a
```

Fig. 2. Functor encoding and decoding

for types of kind $*$ [Magalhães et al. 2010]. For type constructors of kind $* \to *$, as in this paper, one derives $Generic_1$ instead (defined in Figure 2). Instances for non-regular algebraic data types can be defined explicitly, which amounts to giving a representation functor $Rep_1\ f$ along with encoding and decoding operations to_1 and $from_1$. To define a generic algorithm, one provides class instances for these primitives and writes a default definition for each method in terms of $from_1$ and to_1.

The effectiveness of generic programming relies on having at our disposal a variety of data types, each corresponding to a unique composition of the generic type building blocks. In contrast, parallel algorithms are usually designed and implemented in terms of the *single* data type of arrays (or lists in functional formulations; see Section 5). The various array algorithms involve idiosyncratic patterns of traversal and construction of this single data type. For instance, a parallel array reduction with an associative operator involves recursive or iterative generation of numeric indices for extracting elements, taking care that each element is visited exactly once and combined left-to-right. Frequently, an array is split, each half processed recursively and independently, and results combined later. Alternatively, adjacent element pairs are combined, resulting in an array of half size for further processing. (Often, such operations are performed in place, littering the original array with partial reduction results.) The essential idea of these two patterns is the natural fold for perfect, binary leaf trees of two different varieties, but this essence is obscured by implicit *encodings* of trees as arrays. The correctness of the algorithm depends on careful translation of the natural tree algorithm. Mistakes typically hide in the tedious details of index arithmetic, which must be perfectly consistent with the particular encoding chosen. Those mistakes will not be caught by a type-checker (unless programmed with dependent types and full correctness proofs), instead manifesting at run-time in the form of incorrect results and/or index out-of-bound errors. Note also that this array reduction algorithm only works for arrays whose size is a power of two. This restriction is a dynamic condition rather than part of the type signature. If we use the essential data type (a perfect, binary leaf tree) directly rather than via an encoding, it is easy to capture this restriction in the type system and check it statically. The Haskell-based formulations below use GADTs (generalized algebraic data types) and type families.

When we use natural, recursively defined data types *explicitly*, we can use standard programming patterns such as folds and traversals directly. In a language like Haskell, those patterns follow known laws and are well supported by the programming ecosystem. Array encodings make those patterns *implicit*, as a sort of informal guide only, distancing programs from the elegant and well-understood laws and abstractions that motivate those programs, justify their correctness, and point to algorithmic variations that solve related problems or make different implementation trade-offs.

Even the *determinacy* of an imperative, array-based parallel algorithm can be difficult to ensure or verify. When the result is an array, as in scans and FFTs, values are written to indexed locations. In the presence of parallelism, determinacy depends on those write indices being distinct, which again is a subtle, encoding-specific property, unlikely to be verified automatically.

Given these severe drawbacks, why are arrays so widely used in designing, implementing, and explaining parallel algorithms? One benefit is a relatively straightforward mapping from algorithm to efficient implementation primitives. As we will see below, however, we can instead write algorithms in an elegant, modular style using a variety of data types and the standard algebraic abstractions on those data types—such as *Functor*, *Applicative*, *Foldable*, and *Traversable* [McBride and Paterson 2008]—*and* generate very efficient implementations. Better yet, we can define such algorithms generically.

Concretely, this paper makes the following contributions:

- Simple specification of an infinite family of parallel algorithms for each of scan and FFT, indexed by data type and composed out of six generic functor combinators. Two familiar algorithms emerge as the instances of scan and FFT for the common, "top-down" form of perfect binary leaf trees, and likewise two other familiar algorithms for the less common, "bottom-up" form, which is dual to top-down. In addition, two compelling and apparently new algorithms arise from a related third form of perfect "bushes".
- Demonstration of functor composition as the heart of both scan and FFT. Functor composition provides a statically typed alternative to run-time factoring of array sizes often used in FFT algorithms.
- A simple duality between the well-known scan algorithms of Sklansky [1960] and of Ladner and Fischer [1980], revealed by the generic decomposition. This duality is much more difficult to spot in conventional presentations. Exactly the same duality exists between the two known FFT algorithms and is shown clearly and simply in the generic formulation.
- Compositional complexity analysis (work and depth), also based on functor combinators.

The figures in this paper are generated automatically (including optimizations) from the given Haskell code, using a compiler plugin that which also generates synthesizable descriptions in Verilog for massively parallel, hardware-based evaluation [Elliott 2017].

2 SOME USEFUL DATA TYPES

2.1 Right-Lists and Left-Lists

Let's start with a very familiar data type of lists:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

This data type is sometimes more specifically called "cons lists". One might also call them "right lists", since they grow rightward:

$$\textbf{data}\ \mathit{RList}\ a = \mathit{RNil} \mid a \triangleleft \mathit{RList}\ a$$

Alternatively, there are "snoc lists" or "left lists", which grow leftward:

$$\textbf{data}\ \mathit{LList}\ a = \mathit{LNil} \mid \mathit{LList}\ a \triangleright a$$

These two types are isomorphic to types assembled from the functor building blocks of Figure 1:

$$\textbf{type}\ \mathit{RList} \simeq U_1 + \mathit{Par}_1 \times \mathit{RList}$$
$$\textbf{type}\ \mathit{LList} \simeq U_1 + \mathit{RList} \times \mathit{Par}_1$$

Spelling out the isomorphisms explicitly,

```
instance Generic1 RList where
  type Rep1 RList = U1 + Par1 × RList
  from RNil      = L1 U1
  from (a ◁ as)  = R1 (Par1 a × as)
  to (L1 U1)             = RNil
  to (R1 (Par1 a × as))  = a ◁ as
```

```
instance Generic1 LList where
  type Rep1 LList = U1 + LList × Par1
  from LNil      = L1 U1
  from (a ◁ as)  = R1 (as × Par1 a)
  to (L1 U1)             = LNil
  to (R1 (as × Par1 a))  = as ▷ a
```

RList and *LList* are isomorphic not only to their underlying representation functors, but also to each other. Why would we want to distinguish between them? One reason is that they may capture different intentions. For instance, a zipper for right lists comprises a left-list for the (reversed) elements leading up to a position and a right-list for the not-yet-visited elements [Huet 1997; McBride 2001]. Another reason for distinguishing left- from right-lists is that they have usefully different instances for standard type classes, leading—as we will see—to different operational characteristics, especially with regard to parallelism.

2.2 Top-down Trees

After lists, trees are perhaps the most commonly used data structure in functional programming. Moreover, in contrast with lists, the symmetry possible with trees naturally leads to parallel-friendly algorithms. Also unlike lists, there are quite a few varieties of trees.

Let's start with a simple binary leaf tree, i.e., one in which data occurs only in leaves:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

One variation is ternary rather than binary leaf trees:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a) (Tree a)
```

Already, this style of definition is starting to show some strain. The repetition present in the data type definition will be mirrored in instance definitions. For instance, for ternary leaf trees,

```
instance Functor Tree where
  fmap h (Leaf a)           = Leaf (h a)
  fmap h (Branch t1 t2 t3)  = Branch (fmap h t1) (fmap h t2) (fmap h t3)
instance Foldable Tree where
  foldMap h (Leaf a)           = h a
  foldMap h (Branch t1 t2 t3) = foldMap h t1 ⊕ foldMap h t2 ⊕ foldMap h t3
instance Traversable Tree where
  traverse h (Leaf a)           = fmap Leaf (h a)
  traverse h (Branch t1 t2 t3) = liftA3 Branch (traverse h t1) (traverse h t2) (traverse h t3)
```

Not only do we have repetition *within* each instance definition (the three occurrences each of *fmap h*, *foldMap h*, and *traverse h* above), we also have repetition *among* instances for n-ary trees for different n. Fortunately, we can simplify and unify with a shift in formulation. Think of a branch node not as having n subtrees, but rather a single uniform n-tuple of subtrees. Assume for now that we have a functor of finite lists statically indexed by length as well as element type:

```
type Vec :: Nat → ∗ → ∗ -- abstract for now
instance Functor (Vec n) where ...
```

instance *Foldable* (*Vec n*) **where** ...
instance *Traversable* (*Vec n*) **where** ...

Define a single type of n-ary leaf trees, polymorphic over n:

data *Tree n a* = *Leaf a* | *Branch* (*Vec n* (*Tree a*))

The more general vector-based instance definitions are simpler than even for the binary-only version *Tree* type given above:

instance *Functor* (*Tree n*) **where**
  fmap h (*Leaf a*) = *Leaf* (*h a*)
  fmap h (*Branch ts*) = *Branch* ((*fmap* ∘ *fmap*) *h ts*)

instance *Foldable* (*Tree n*) **where**
  foldMap h (*Leaf a*) = *h a*
  foldMap h (*Branch ts*) = (*foldMap* ∘ *foldMap*) *h ts*

instance *Traversable* (*Tree n*) **where**
  traverse h (*Leaf a*) = *fmap Leaf* (*h a*)
  traverse h (*Branch ts*) = *fmap Branch* ((*traverse* ∘ *traverse*) *h ts*)

Notice that these instance definitions rely on very little about the *Vec n* functor. Specifically, for each of *Functor*, *Foldable*, and *Traversable*, the instance for *Tree n* needs only the corresponding instance for *Vec n*. For this reason, we can easily generalize from *Vec n* as follows:

data *Tree f a* = *Leaf a* | *Branch* (*f* (*Tree a*))

The instance definitions for "f-ary" trees (also known as the "free monad" for the functor f) are exactly as with n-ary, except for making the requirements on f explicit:

instance *Functor* $f \Rightarrow$ *Functor* (*Tree f*) **where** ...
instance *Foldable* $f \Rightarrow$ *Foldable* (*Tree f*) **where** ...
instance *Traversable f* $\Rightarrow$ *Traversable* (*Tree f*) **where** ...

This generalization covers "list-ary" (rose) trees and even "tree-ary" trees. With this functor-parametrized tree type, we can reconstruct n-ary trees as *Tree* (*Vec n*).

Just as there are both left- and right-growing lists, f-ary trees come in two flavors as well. The forms above are all "top-down", in the sense that successive unwrappings of branch nodes reveal subtrees moving from the top downward. (No unwrapping for the top level, one unwrapping for the collection of next-to-top subtrees, another for the collection of next level down, etc.) There are also "bottom-up" trees, in which successive branch node unwrappings reveal the information in subtrees from the bottom moving upward. In short:

- A top-down leaf tree is either a leaf or an f-structure of trees.
- A bottom-up leaf tree is either a leaf or a tree of f-structures.

In Haskell,

data *TTree f a* = *TLeaf a* | *TBranch* (*f* (*TTree a*))

data *BTree f a* = *BLeaf a* | *BBranch* (*BTree* (*f a*))

Bottom-up trees (*BTree*) are a canonical example of "nested" or "non-regular" data types, requiring polymorphic recursion [Bird and Meertens 1998]. As we'll see below, they give rise to important versions of parallel scan and FFT.

2.3 Statically Shaped Variations

Some algorithms work only on collections of restricted size. For instance, the most common parallel scan and FFT algorithms are limited to arrays of size 2^n, while the more general (not just binary) Cooley-Tukey FFT algorithms require composite size, i.e., $m \cdot n$ for integers $m, n \geq 2$. In array-based algorithms, these restrictions can be realized in one of two ways:

- check array sizes dynamically, incurring a performance penalty; or
- document the restriction, assume the best, and blame the library user for errors.

A third option—much less commonly used—is to statically verify the size restriction at the call site, perhaps by using a dependently typed language and providing proofs as part of the call.

A lightweight compromise is to simulate some of the power of dependent types via type-level encodings of sizes, as with our use of *Nat* for indexing the *Vec* type above. There are many possible definitions for *Nat*. For this paper, assume that *Nat* is a kind-promoted version of the following data type of Peano numbers (constructed via zero and successor):

```
data Nat = Z | S Nat
```

Thanks to promotion (via the *DataKinds* language extension), *Nat* is not only a new data type with value-level constructors *Z* and *S*, but also a new *kind* with *type-level* constructors *Z* and *S* [Yorgey et al. 2012].

2.3.1 GADT Formulation. Now we can define the length-indexed *Vec* type mentioned above. As with lists, there are right- and left-growing versions:

```
data RVec :: Nat → ∗ → ∗ where
  RNil :: RVec Z a
  (◃)  :: a → RVec n a → RVec (S n) a
```

```
data LVec :: Nat → ∗ → ∗ where
  LNil :: LVec Z a
  (▹)  :: LVec n a → a → LVec (S n) a
```

Recall that the generic representations of *RList* and *LList* were built out of sum, unit, identity, and product. With static shaping, the sum disappears from the representation, moving from dynamic to static choice, and each $Generic_1$ instance split into two:

```
instance Generic1 (RVec Z) where
  type Rep1 (RVec Z) = U1
  from RNil = U1
  to U1 = RNil
```

```
instance Generic1 (LVec Z) where
  type Rep1 (LVec Z) = U1
  from RNil = U1
  to U1 = RNil
```

```
instance Generic1 (RVec n) ⇒
         Generic1 (RVec (S n)) where
  type Rep1 (RVec (S n)) = Par1 × RVec n
  from (a ◃ as) = Par1 a × as
  to (Par1 a × as) = a ◃ as
```

```
instance Generic1 (LVec n) ⇒
         Generic1 (LVec (S n)) where
  type Rep1 (LVec (S n)) = LVec n × Par1
  from (a ◃ as) = Par1 a × as
  to (Par1 a × as) = a ◃ as
```

For leaf trees, we have a choice between imperfect and perfect trees. A "perfect" leaf tree is one in which all leaves are at the same depth. Both imperfect and perfect can be "statically shaped", but we'll use just perfect trees in this paper, for which we need only a single type-level number signifying the depth of all leaves. For succinctness, rename *Leaf* and *Branch* to "*L*" and "*B*". For reasons soon to be explained, let's also rename the types *TTree* and *BTree* to "*RPow*" and "*LPow*":

```
data RPow :: (* → *) → Nat → * → * where
  L :: a → RPow f Z a
  B :: f (RPow f n a) → RPow f (S n) a
```

```
data LPow :: (* → *) → Nat → * → * where
  L :: a → LPow f Z a
  B :: LPow f n (f a) → LPow f (S n) a
```

As with vectors, statically shaped f-ary trees are generically represented like their dynamically shaped counterparts but with dynamic choice (sum) replaced by static choice:

```
instance Generic1 (RPow f Z) where
  type Rep1 (RPow f Z) = Par1
  from1 (L a) = Par1 a
  to1 (Par1 a) = L a
```

```
instance Generic1 (LPow f Z) where
  type Rep1 (LPow f Z) = Par1
  from1 (L a) = Par1 a
  to1 (Par1 a) = L a
```

```
instance Generic1 (RPow f n) ⇒
         Generic1 (RPow f (S n)) where
  type Rep1 (RPow f (S n)) = f ∘ RPow f n
  from1 (B ts) = Comp1 ts
  to1 (Comp1 ts) = B ts
```

```
instance Generic1 (LPow f n) ⇒
         Generic1 (LPow f (S n)) where
  type Rep1 (LPow f (S n)) = LPow f n ∘ f
  from1 (B ts) = Comp1 ts
  to1 (Comp1 ts) = B ts
```

We can then give these statically shaped data types *Functor*, *Foldable*, and *Traversable* instances matching the dynamically shaped versions given above. In addition, they have *Applicative* and *Monad* instances. Since all of these types are memo tries [Hinze 2000], their class instances instance follow homomorphically from the corresponding instances for functions [Elliott 2009].

2.3.2 Type Family Formulation. Note that *RVec n* and *LVec n* are essentially n-ary functor *products* of Par_1. Similarly, *RPow f n* and *LPow f n* are n-ary functor *compositions* of f. Functor product and functor composition are both associative only up to isomorphism. While *RVec* and *RPow* are right associations, *LVec* and *LPow* are left associations. As we will see below, different associations, though isomorphic, lead to different algorithms.

Instead of the GADT-based definitions given above for *RVec*, *LVec*, *RPow*, and *LPow*, we can make the repeated product and repeated composition more apparent by using closed type families [Eisenberg et al. 2014], with instances defined inductively over type-level natural numbers:

```
type family RVec n where
  RVec Z     = U1
  RVec (S n) = Par1 × RVec n
```

```
type family LVec n where
  LVec Z     = U1
  LVec (S n) = LVec n × Par1
```

```
type family RPow h n where
  RPow h Z     = Par1
  RPow h (S n) = h ∘ RPow h n
```

```
type family LPow h n where
  LPow h Z     = Par1
  LPow h (S n) = LPow h n ∘ h
```

Note the similarity between the *RVec* and *RPow* type family instances and the following definitions of multiplication and exponentiation on Peano numbers (with RHS parentheses for emphasis):

$$0 * a = 0$$
$$(1 + n) * a = a + (n * a)$$

$$0 * a = 0$$
$$(n + 1) * a = (n * a) + a$$

$$h \uparrow 0 = 1$$
$$h \uparrow (1 + n) = h * (h \uparrow n)$$

$$h \uparrow 0 = 1$$
$$h \uparrow (n + 1) = (h \uparrow n) * h$$

Because the type-family-based definitions are expressed in terms of existing generic building blocks, we inherit many existing class instances rather than having to define them. For the same

reason, we *cannot* provide them (since instances already exist), which will pose a challenge (though easily surmounted) with FFT on vectors, as well as custom *Show* instances for displaying structures.

Although *RPow* and *LPow* work with any functor argument, we will use uniform pairs in the examples below. The uniform *Pair* functor can be defined in a variety of ways, including $Par_1 \times Par_1$, *RVec* 2, *LVec* 2, or its own algebraic data type:

data *Pair a* = *a* :# *a* **deriving** (*Functor*, *Foldable*, *Traversable*)

For convenience, define top-down and bottom-up *binary* trees:

type *RBin* = *RPow Pair*
type *LBin* = *LPow Pair*

2.4 Bushes

In contrast to vectors, the tree types above are perfectly balanced, as is helpful in obtaining naturally parallel algorithms. From another perspective, however, they are quite unbalanced. The functor composition operator is used fully left-associated for *LPow* and fully right-associated for *RPow* (hence the names). It's easy to define a composition-balanced type as well:

type family *Bush n* **where**
Bush Z = *Pair*
Bush (*S n*) = *Bush n* ∘ *Bush n*

While each *RBin n* and *LBin n* holds 2^n elements, each statically shaped *Bush n* holds 2^{2^n} elements. Moreover, there's nothing special about *Pair* or *binary* composition here. Either could be replaced or generalized.

Our "bush" type is inspired by an example of nested data types that has a less regular shape [Bird and Meertens 1998]:

data *Bush a* = *NilB* | *ConsB a* (*Bush* (*Bush a*))

Bushes are to trees as trees are to vectors, in the following sense. Functor product is associative up to isomorphism. Where *RVec* and *LVec* choose fully right- or left-associated products, *RBin* and *LBin* form perfectly and recursively balanced products (being repeated compositions of *Pair*). Likewise, functor composition is associative up to isomorphism. Where *RBin* and *LBin* are fully right- and left-associated compositions, *Bush n* forms balanced compositions. Many other variations are possible, but the *Bush* definition above will suffice for this paper.

3 PARALLEL SCAN

Given a sequence $a_0, \ldots, a_{n-1}$, the "prefix sum" is a sequence $b_0, \ldots, b_n$ such that $b_k = \sum_{0 \le i < k} a_i$. More generally, for any associative operation $\oplus$, the "prefix scan" is defined by $b_k = \bigoplus_{0 \le i < k} a_i$, with b_0 being the identity for $\oplus$. (One can define a similar operation if we assume semigroup—lacking identity element—rather than monoid, but the development is more straightforward with identity.)

Scan has broad applications, including the following, taken from a longer list [Blelloch 1990]:

- Adding multi-precision numbers
- Polynomial evaluation
- Solving recurrences
- Sorting
- Solving tridiagonal linear systems
- Lexical analysis

- Regular expression search
- Labeling components in two dimensional images

An efficient, *parallel* scan algorithm thus enables each of these applications to be performed in parallel. Scans may be "prefix" (from the left, as above) or or "suffix" (from the right). We will just develop prefix scan, but generic suffix scan works out in the same way.

Note that a_k does *not* influence b_k. Often scans are classified as "exclusive", as above, or "inclusive", where a_k does contribute to b_k. Note also that there is one more output element than input, which is atypical in the literature on parallel prefix algorithms, perhaps because scans are often performed in place. As we will see below, the additional output makes for an elegant generic decomposition.

The standard list prefix scans in Haskell, *scanl* and *scanr*, also yield one more output element than input, which is possible for lists. For other data types, such as trees and especially perfect ones, there may not be a natural place to store the extra value. For a generic scan applying to many different data types, we can simply form a product, so that scanning maps $f\ a$ to $f\ a \times a$. The extra summary value is the fold over the whole input structure. We thus have the following class for left-scannable functors:

$$\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{LScan}\ f\ \textbf{where}\ \mathit{lscan} :: \mathit{Monoid}\ a \Rightarrow f\ a \to f\ a \times a$$

The *Functor* superclass is just for convenience and can be dropped in favor of more verbose signatures elsewhere.

When f is in *Traversable*, there is a simple and general specification using operations from the standard Haskell libraries:

$$\mathit{lscan} \equiv \mathit{swap} \circ \mathit{mapAccumL}\ (\lambda \mathit{acc}\ a \to (\mathit{acc} \oplus a, \mathit{acc}))\ \emptyset$$

where $(\oplus)$ and $\emptyset$ are the combining operation and its identity from *Monoid*, and

$$\mathit{mapAccumL} :: \mathit{Traversable}\ t \Rightarrow (b \to a \to b \times c) \to b \to t\ a \to b \times t\ c$$

Although all of the example types in this paper are indeed in *Traversable*, using this *lscan* specification as an implementation would result in an entirely sequential implementation, since data dependencies are *linearly* threaded through the whole computation.

Rather than defining *LScan* instances for all of our data types, the idea of generic programming is to define instances only for the small set of fundamental functor combinators and then automatically compose instances for other types via the generic encodings (derived automatically when possible). To do so, we can simply provide a default signature and definition for functors with such encodings:

$$\begin{aligned}&\textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{LScan}\ f\ \textbf{where}\\ &\quad \mathit{lscan} :: \mathit{Monoid}\ a \Rightarrow f\ a \to f\ a \times a\\ &\quad \textbf{default}\ \mathit{lscan} :: (\mathit{Generic}_1\ f, \mathit{LScan}\ (\mathit{Rep}_1\ f), \mathit{Monoid}\ a) \Rightarrow f\ a \to f\ a \times a\\ &\quad \mathit{lscan} = \mathit{first}\ \mathit{to}_1 \circ \mathit{lscan} \circ \mathit{from}_1\end{aligned}$$

Once we define *LScan* instances for our six fundamental combinators, one can simply write "**instance** *LScan F*" for any functor F having a $\mathit{Generic}_1$ instance (derived automatically or defined manually). For our statically shaped vector, tree, and bush functors, we have a choice: use the GADT definitions with their manually defined $\mathit{Generic}_1$ instances (exploiting the *lscan* default), or use the type family versions without the need for the encoding (from_1) and decoding (to_1) steps.

3.1 Easy Instances

Four of the six needed generic *LScan* instances are easily handled:

$$\begin{aligned}&\textbf{instance}\ \mathit{LScan}\ V_1 \quad \textbf{where}\ \mathit{lscan} = \lambda\ \textbf{case}\\ &\textbf{instance}\ \mathit{LScan}\ U_1 \quad \textbf{where}\ \mathit{lscan}\ U_1 = (U_1, \emptyset)\end{aligned}$$

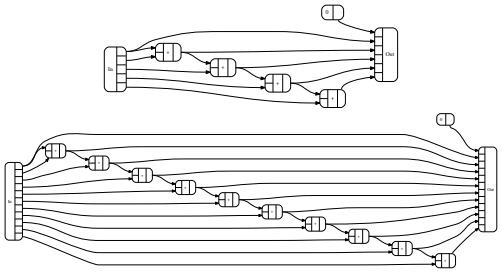

Fig. 3. *lscan* @(*RVec* 5) and *lscan* @(*RVec* 11)

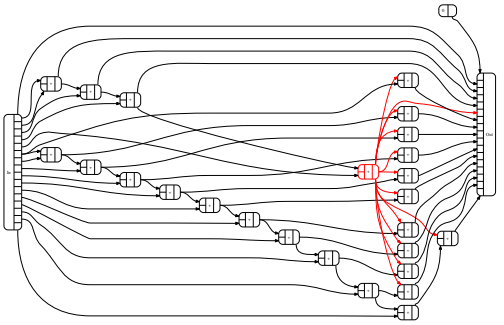

Fig. 4. *lscan* @(*RVec* 5 × *RVec* 11) *[W=26, D=11]*

instance $LScan\ Par_1$ **where** $lscan\ (Par_1\ a) = (Par_1\ \emptyset, a)$

instance $(LScan\ f, LScan\ g) \Rightarrow LScan\ (f + g)$ **where**
$lscan\ (L_1\ fa) = first\ L_1\ (lscan\ fa)$
$lscan\ (R_1\ ga) = first\ R_1\ (lscan\ ga)$

Comments:

- Since there are no values of type $V_1\ a$, a complete case expression needs no clauses. (The definition relies on the *LambdaCase* and *EmptyCase* language extensions.)
- An empty structure can only generate another empty structure with a summary value of $\emptyset$.
- For a singleton value $Par_1\ a$, the combination of values before the first and only one is $\emptyset$, and the summary is the value a.
- For a sum, scan and re-tag. (The higher-order function *first* applies a function to the first element of a pair, carrying the second element along unchanged [Hughes 1998].)

Just as the six functor combinators guide the composition of parallel algorithms, they also determine the performance characteristics of those parallel algorithms in a compositional manner. Following Blelloch [1996], consider two aspects of performance:

- *work*, the total number of primitive operations performed, and
- *depth*, the longest dependency chain, and hence a measure of ideal parallel computation time.

For parallel scan, work and depth of U_1, V_1, and Par_1 are all zero. For sums,

$$W\ (f + g) = W\ f\ `max`\ W\ g$$
$$D\ \ (f + g) = D\ \ f\ `max`\ D\ \ g$$

3.2 Product

Suppose we have linear scans, as in Figure 3. We will see later how these individual scans arise from particular functors f and g (of sizes five and eleven), but for now take them as given. To understand *lscan* on functor products, consider how to combine the scans of f and g into scan for $f \times g$.

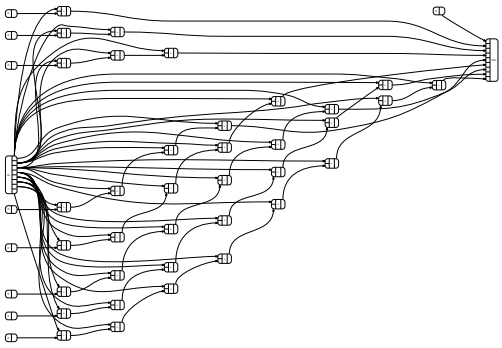

Fig. 5. *lscan* @(*RVec* 8), unoptimized *[W=36, D=8]*

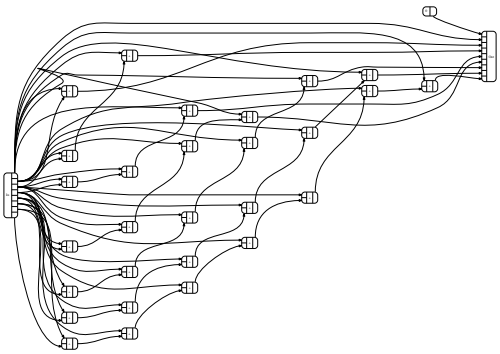

Fig. 6. *lscan* @(*RVec* 8), optimized *[W=28, D=7]*

Because we are left-scanning, every prefix of f is also a prefix of $f \times g$, so the *lscan* results for f are also correct results for $f \times g$. The prefixes of g are not prefixes of $f \times g$, however, since each g-prefix misses all of f. The prefix *sums*, therefore, are lacking the summary (fold) of all of f, which corresponds to the last output of the *lscan* result for f. All we need to do, therefore, is adjust each g result by the final f result, as shown in Figure 4. The general product instance:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{LScan}\ f, \mathit{LScan}\ g) \Rightarrow \mathit{LScan}\ (f \times g)\ \textbf{where} \\
\quad \mathit{lscan}\ (\mathit{fa} \times \mathit{ga}) = (\mathit{fa}' \times \mathit{fmap}\ (\mathit{fx} \oplus)\ \mathit{ga}', \mathit{fx} \oplus \mathit{gx}) \\
\qquad \textbf{where} \\
\qquad\quad (\mathit{fa}', \mathit{fx}) = \mathit{lscan}\ \mathit{fa} \\
\qquad\quad (\mathit{ga}', \mathit{gx}) = \mathit{lscan}\ \mathit{ga}
\end{array}
$$

The work for $f \times g$ is the combined work for each, plus the cost of adjusting the result for g. The depth is the maximum depth for f and g, plus one more step to adjust the final g result.

$$
\begin{aligned}
W\ (f \times g) &= W\ f + W\ g + |g| + 1 \\
D\ \ (f \times g) &= (D\ f\ \text{`}\mathit{max}\text{`}\ D\ g) + 1
\end{aligned}
$$

We now have enough functionality for scanning vectors using the GADT or type family definitions from Section 2.3. Figure 5 shows *lscan* for *RVec* 8 (*right* vector of length 8). The zero-additions are easily optimized away, resulting in Figure 6. In this picture (and many more like it below), the data types are shown in flattened form in the input and output (labeled *In* and *Out*), and work and depth are shown in the caption (as W and D). As promised, there is always one more output than input, and the last output is the fold that summarizes the entire structure being scanned.

The combination of left scan and right vector is particularly unfortunate, as it involves quadratic work and linear depth. The source of quadratic work is the product instance's *right* adjustment combined with the right-associated shape of *RVec*. Each single element is used to adjust the entire suffix, requiring linear work at each step, summing to quadratic. We can verify the complexity by using the definition of *RVec* and the complexities for the generic building blocks involved.

$$
\begin{aligned}
&W\ (\mathit{RVec}\ 0) = W\ U_1 = 0 \\
&W\ (\mathit{RVec}\ (S\ n)) = W\ (\mathit{Par}_1 \times \mathit{RVec}\ n) = W\ \mathit{Par}_1 + W\ (\mathit{RVec}\ n) + |\mathit{RVec}\ n| + 1 \\
&\qquad\qquad\qquad = W\ (\mathit{RVec}\ n) + O(n) \\
&D\ (\mathit{RVec}\ 0) = D\ U_1 = 0 \\
&D\ (\mathit{RVec}\ (S\ n)) = D\ (\mathit{Par}_1 \times \mathit{RVec}\ n) = (D\ \mathit{Par}_1\ \text{`}\mathit{max}\text{`}\ D\ (\mathit{RVec}\ n)) + 1 = D\ (\mathit{RVec}\ n) + O(1)
\end{aligned}
$$

Thus, $W\ (\mathit{RVec}\ n) = O(n^2)$, and $D\ (\mathit{RVec}\ n) = O(n)$.

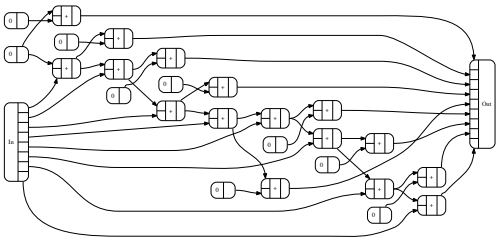

Fig. 7. *lscan* @(*LVec* 8), unoptimized *[W=16, D=8]*

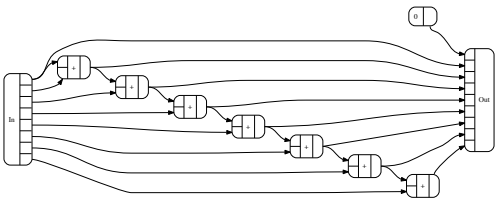

Fig. 8. *lscan* @(*LVec* 8), optimized *[W=7, D=7]*

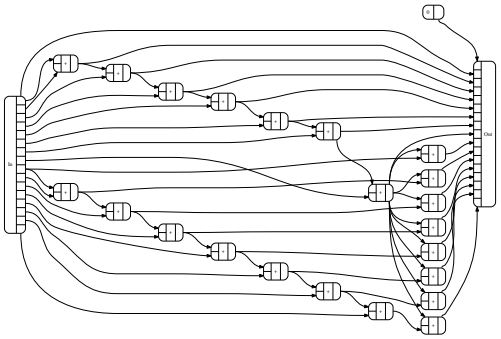

Fig. 9. *lscan* @(*LVec* 8 × *LVec* 8) *[W=22, D=8]*

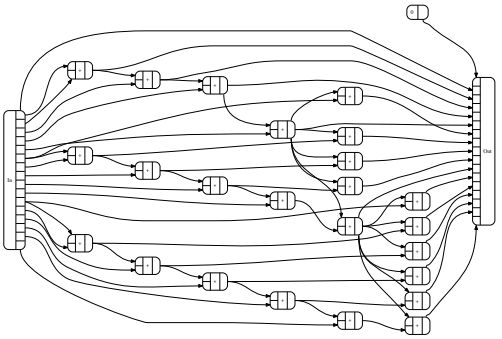

Fig. 10. *lscan* @((*LVec* 5 × *LVec* 5) × *LVec* 6) *[W=24, D=6]*

In contrast, with left-associated vectors, each prefix summary (left) is used to update a single element (right), leading to linear work, as shown in Figure 7 and (optimized) Figure 8.

$$W\ (LVec\ 0) = W\ U_1 = 0$$
$$W\ (LVec\ (S\ n)) = W\ (LVec\ n \times Par_1) = W\ (LVec\ n) + W\ Par_1 + |Par_1| + 1 = W\ (LVec\ n) + 2$$
$$D\ (RVec\ 0) = D\ U_1 = 0$$
$$D\ (RVec\ (S\ n)) = D\ (Par_1 \times RVec\ n) = (D\ Par_1\ `max`\ D\ (RVec\ n)) + 1 = D\ (RVec\ n) + O(1)$$

Thus, $W\ (RVec\ n) = O(n)$, and $D\ (RVec\ n) = O(n)$.

Although work is greatly reduced (from quadratic to linear), depth remains at linear, because unbalanced data types lead to unbalanced parallelism. Both *RVec* and *LVec* are "parallel" in a sense, but we only get to perform small computations in parallel with large one (especially apparent in the unoptimized Figures 5 and 7), so that the result is essentially sequential.

To get more parallelism, we could replace a type like *LVec* 16 with a isomorphic product such as *LVec* 5 × *LVec* 11, resulting in Figure 4, reducing depth from 15 to 11. More generally, scan on *LVec* m × *LVec* n has depth $((m-1)\ `max`\ (n-1)) + 1 = m\ `max`\ n$. For an ideal partition adding up to p, we'll want $m = n = p\ /\ 2$. For instance, replace *LVec* 16 with the isomorphic product *LVec* 8 × *LVec* 8, resulting in Figure 9 with depth eight. Can we decrease the depth any further? Not as a single product, but we can as more than one product, as shown in Figure 10 with depth six.

3.3 Composition

We now come to the last of our six functor combinators, namely composition, i.e., structures of structures. Suppose we have a triple of quadruples: *LVec* 3 ∘ *LVec* 4. We know how to scan each quadruple, as in Figure 11. How can we combine the results of each scan into a scan for *LVec* 3 ∘ *LVec* 4? We already know the answer, since this composite type is essentially (*LVec* 4 ×

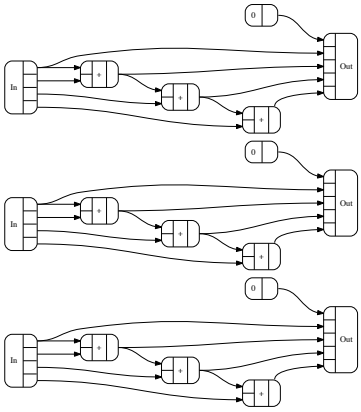

Fig. 11. triple *lscan* @(*LVec* 4)

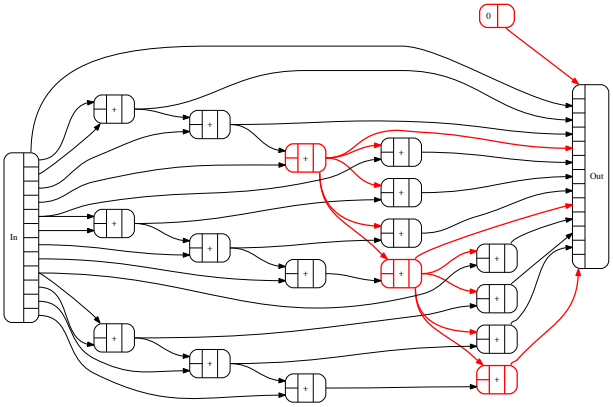

Fig. 12. *lscan* @(*LVec* 3 ∘ *LVec* 4) *[W=18, D=5]*

LVec 4) × *LVec* 4, the scan for which is fully determined by the Par_1 and product instances and is shown in Figure 12.

Let's reflect on this example as we did with binary products above. Since the prefixes of the first quadruple are all prefixes of the composite structure, their prefix sums are prefix sums of the composite and so are used as they are. For every following quadruple, the prefix sums are lacking the sum of all elements from the earlier quadruples and so must be adjusted accordingly, as emphasized in Figure 12.

Now we get to the surprising heart of generic parallel scan. Observe that the sums of elements from all earlier quadruples are computed entirely from the final summary results from each quadruple. We end up needing the sum of every *prefix* of the triple of summaries, and so we are computing not just three prefix scans over *LVec* 4 but also *one additional scan* over *LVec* 3 (highlighted in Figure 12). Moreover, the apparent inconsistency of adjusting all quadruples *except* for the first one is an illusion brought on by premature optimization. We can instead adjust *every* quadruple by the corresponding result of this final scan of summaries, the first summary being zero. These zero-additions can be optimized away later.

The general case is captured in an *LScan* instance for functor composition:

```
instance (LScan g, LScan f, Zip g) ⇒ LScan (g ∘ f) where
  lscan (Comp1 gfa) = (Comp1 (zipWith adjustl tots' gfa'), tot)
    where
      (gfa', tots) = unzip (fmap lscan gfa)
      (tots', tot) = lscan tots
      adjustl t    = fmap (t ⊕)
```

The work for scanning $g \circ f$ includes work for each f, work for the g of summaries, and updates to all results (before optimizing away the zero adjust, which doesn't change order). The depth is the depth of f (since each f is handled in parallel with the others), followed by the depth of a single g scan.[1]

$$W\ (g \circ f) = |g| \cdot W\ f + W\ g + |g| \cdot |f|$$
$$D\ \ (g \circ f) = D\ f + D\ g$$

[1]This simple depth analysis is pessimistic in that it does not account for the fact that some g work can begin before all f work is complete.

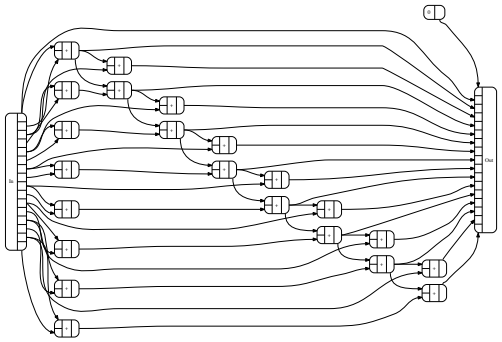

Fig. 13. *lscan* @(*LVec* 8 ∘ *Pair*) *[W=22, D=8]*

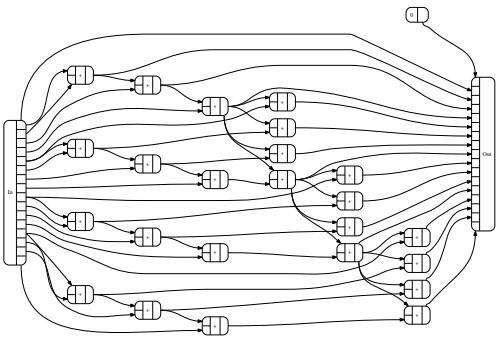

Fig. 14. *lscan* @(*LVec* 4 ∘ *LVec* 4) *[W=24, D=6]*

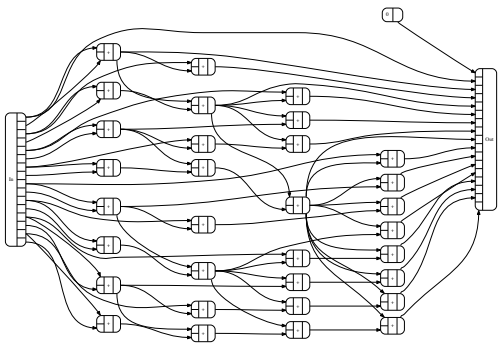

Fig. 15. *lscan* @(*RBin* 4) *[W=32, D=4]*

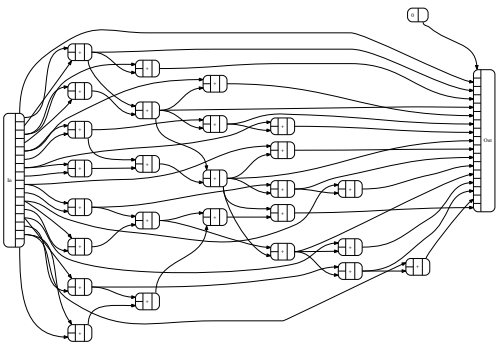

Fig. 16. *lscan* @(*LBin* 4) *[W=26, D=6]*

3.4 Other Data Types

We now know how to scan the full vocabulary of generic functor combinators, and we've seen the consequences for several data types. Let's now examine how well generic scan works for some other example structures. We have already seen *Pair* ∘ *LVec* 8 as *LVec* 8 × *LVec* 8 in Figure 9. The reverse composition leads to quite a different computation shape, as Figure 13 shows. Yet another factoring appears in Figure 14.

Next let's try functor exponentiation in its left- and right-associated forms. We just saw the equivalent of *RPow* (*LVec* 4) 2 (and *LPow* (*LVec* 4) 2) as Figure 14. Figures 15 and 16 show *RBin* 4 and *LBin* 4 (top-down and bottom-up perfect binary leaf trees of depth four). Complexities for *RPow h*:

$$W\ (RPow\ h\ 0) = W\ Par_1 = 0$$
$$W\ (RPow\ h\ (S\ n)) = W\ (h \circ RPow\ h\ n) = |h| \cdot W\ (RPow\ h\ n) + W\ h + |h|^{S\ n}$$
$$D\ (RPow\ h\ 0) = D\ Par_1 = 0$$
$$D\ (RPow\ h\ (S\ n)) = D\ (h \circ RPow\ h\ n) = D\ h + D\ (RPow\ h\ n)$$

For any fixed h, $W\ h + |h|^{S\ n} = O(n)$, so the Master Theorem [Cormen et al. 2009, Chapter 4] gives a solution for W. Since $D\ h = O(1)$ (again, for fixed h), D has a simple solution.

$$W\ (RPow\ h\ n) = O(|RPow\ h\ n| \cdot log\ |RPow\ h\ n|)$$
$$D\ \ (RPow\ h\ n) = O(n) = O(log\ |RPow\ h\ n|)$$

Complexity for *LPow h* works out somewhat differently:

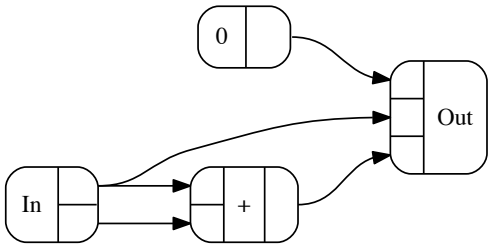

Fig. 17. *lscan* @(*Bush* 0) *[W=1, D=1]*

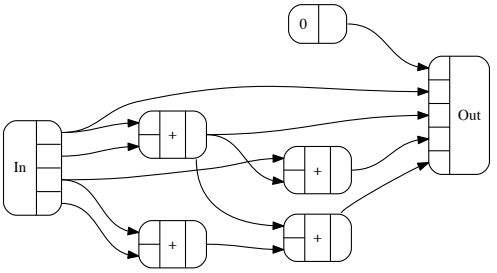

Fig. 18. *lscan* @(*Bush* 1) *[W=4, D=2]*

$$W\ (LPow\ h\ 0) = W\ Par_1 = 0$$
$$W\ (LPow\ h\ (S\ n)) = W\ (LPow\ h\ n \circ h) = |LPow\ h\ n| \cdot W\ h + W\ (LPow\ h\ n) + |h|^{S\ n}$$
$$D\ (LPow\ h\ 0) = D\ Par_1 = 0$$
$$D\ (LPow\ h\ (S\ n)) = D\ (LPow\ h\ n \circ h) = D\ (LPow\ h\ n) + D\ h$$

With a fixed h, $|LPow\ h\ n| \cdot W\ h + |h|^{S\ n} = O(|LPow\ h\ n|)$, so the Master Theorem gives a solution *linear* in $|LPow\ h\ n|$, while the depth is again logarithmic:

$$W\ (LPow\ h\ n) = O(|LPow\ h\ n|)$$
$$D\ \ (LPow\ h\ n) = O(n) = O(log\ |LPow\ h\ n|)$$

For this reason, parallel scan on bottom-up trees can do much less work than on top-down trees. They also have fan-out bounded by $|h|$, as contrasted with the linear fan-out for top-down trees—an important consideration for hardware implementations. On the other hand, the depth for bottom-up trees is about twice the depth for top-down trees.

Specializing these *RPow h* and *LPow h* scan algorithms to $h = Pair$ and then optimizing away zero-additions (as in Figures 15 and 16) yields two well-known algorithms: *lscan* on *RBin n* is from [Sklansky 1960], while *lscan* on *LBin n* is from [Ladner and Fischer 1980].

Finally, consider the *Bush* type from Section 2.4. Figures 17 through 19 show *lscan* for bushes of depth zero through two. Depth complexity:

$$D\ (Bush\ 0) = D\ Pair = 1$$
$$D\ (Bush\ (S\ n)) = D\ (Bush\ n \circ Bush\ n) = D\ (Bush\ n) + D\ (Bush\ n) = 2 \cdot D\ (Bush\ n)$$

Hence

$$D\ (Bush\ n) = 2^n = 2^{log_2\ (log_2\ |Bush\ n|)} = log_2\ |Bush\ n|$$

Work complexity is trickier:

$$\begin{aligned} W\ (Bush\ (S\ n)) &= W\ (Bush\ n \circ Bush\ n) = |Bush\ n| \cdot W\ (Bush\ n) + W\ (Bush\ n) + |Bush\ (S\ n)| \\ &= 2^{2^n} \cdot W\ (Bush\ n) + W\ (Bush\ n) + 2^{2^{n+1}} \\ &= (2^{2^n} + 1) \cdot W\ (Bush\ n) + 2^{2^{n+1}} \end{aligned}$$

A closed form solution is left for later work. Figures 20 and 21 offer an empirical comparison, including some optimizations not taken into account in the complexity analyses above. Note that top-down trees have the least depth, bottom-up trees have the least work, and bushes provide a compromise, with less work than top-down trees and less depth than bottom-up trees.

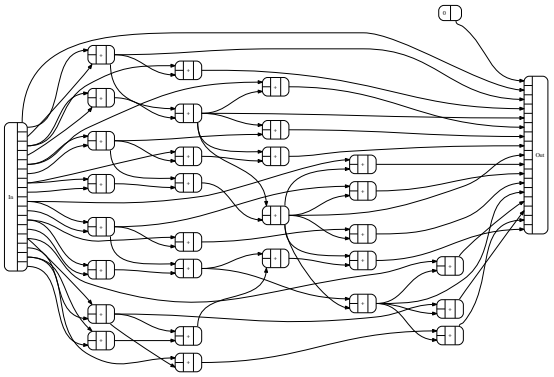

Fig. 19. *lscan* @(*Bush* 2) *[W=29, D=5]*

	operations	depth
RBin 4	32	4
LBin 4	26	6
Bush 2	29	5

Fig. 20. *lscan* for 16 values

	operations	depth
RBin 8	1024	8
LBin 8	502	14
Bush 3	718	10

Fig. 21. *lscan* for 256 values

3.5 Some Convenient Packaging

For generality, *lscan* works on arbitrary monoids. For convenience, let's define some specializations. One way to do so is to provide functions that map between non-monoids and monoids. Start with a class similar to *Generic* for providing alternative representations:

```
class Newtype n where
  type O n :: *
  pack   :: O n → n
  unpack :: n → O n
```

This class also defines many instances for commonly used types [Jahandarie et al. 2014]. Given this vocabulary, we can scan structures over a non-monoid by packing values into a chosen monoid, scanning, and then unpacking:[2]

```
lscanNew :: ∀n o f.(Newtype n, o~O n, LScan f, Monoid n) ⇒ f o → f o × o
lscanNew = (fmap unpack *** unpack) ∘ lscan ∘ fmap (pack @n)

lsums, lproducts :: (LScan f, Num a) ⇒ f a → f a × a
lalls, lanys :: LScan f ⇒ f Bool → f Bool × Bool
lsums    = lscanNew @(Sum a)
lproducts = lscanNew @(Product a)
lalls    = lscanNew @All
lanys    = lscanNew @Any
...
```

3.6 Applications

As a first simple example application of parallel scan, let's construct powers of a given number x to fill a structure f, so that successive elements are x^0, x^1, x^2 etc. A simple implementation builds a structure with identical values using *pure* (from *Applicative*) and then calculates all prefix products:

```
powers :: (LScan f, Applicative f, Num a) ⇒ a → f a × a
powers = lproducts ∘ pure
```

[2]The (***) operation applies two given functions to the respective components of a pair, and the "@" notation is visible type application [Eisenberg et al. 2016].

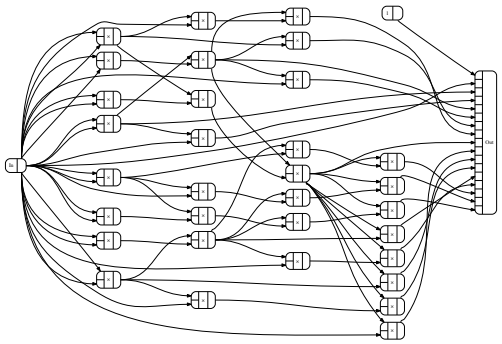

Fig. 22. *powers* @(*RBin* 4), no CSE *[W=32, D=4]*

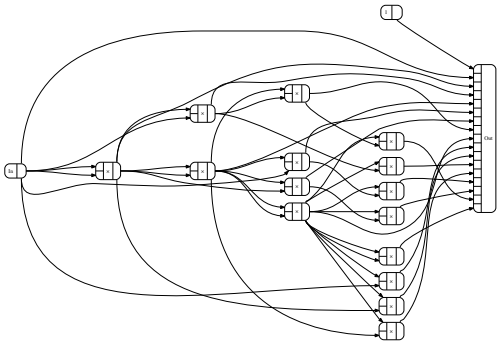

Fig. 23. *powers* @(*RBin* 4), CSE *[W=15, D=4]*

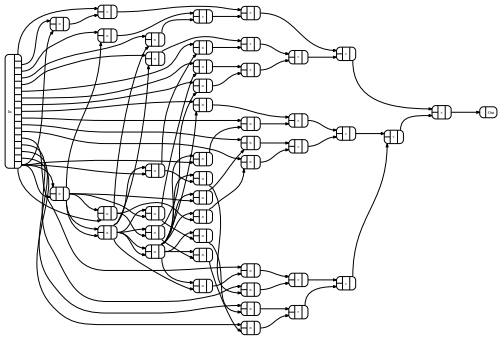

Fig. 24. *evalPoly* @(*RBin* 4) *[W=29+15, D=9]*

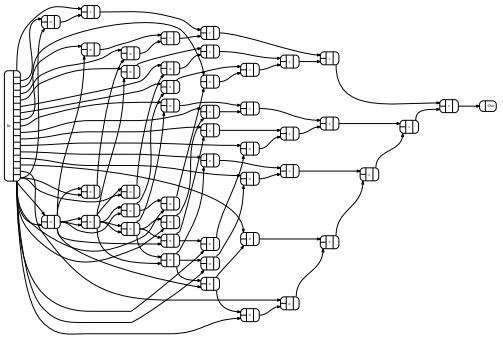

Fig. 25. *evalPoly* @(*LBin* 4) *[W=29+15, D=11]*

Figure 22 shows one instance of *powers*. A quick examination shows that there is a lot of redundant computation due to the special context of scanning over identical values. For instance, for an input x, we compute x^2 eight times and x^4 four times. Fortunately, automatic common subexpression elimination (CSE) can remove such redundancies easily, resulting in Figure 23.

Building on this example, let's define polynomial evaluation, mapping a structure of coefficients $a_0, \ldots, a_{n-1}$ and a parameter x to $\sum_{0 \le i < n} a_i x^i$. A very simple formulation is to construct all of the powers of x and then form a dot product with the coefficients:

$$
\begin{aligned}
&\mathit{evalPoly} :: (\mathit{LScan}\ f, \mathit{Foldable}\ f, \mathit{Applicative}\ f, \mathit{Num}\ a) \Rightarrow f\ a \to a \to a \\
&\mathit{evalPoly}\ \mathit{coeffs}\ x = \mathit{coeffs} \cdot \mathit{fst}\ (\mathit{powers}\ x) \\
&(\cdot) :: (\mathit{Foldable}\ f, \mathit{Applicative}\ f, \mathit{Num}\ a) \Rightarrow f\ a \to f\ a \to a \\
&u \cdot v = \mathit{sum}\ (\mathit{liftA}_2\ (*)\ u\ v)
\end{aligned}
$$

Figures 24 and 25 show the results for top-down and bottom-up trees.

4 FFT

4.1 Background

The fast Fourier transform (FFT) algorithm computes the Discrete Fourier Transform (DFT), reducing work from $O(n^2)$ to $O(n \log n)$. First discovered by Gauss [Heideman et al. 1984], the algorithm was rediscovered by Danielson and Lanczos [1942], and later by Cooley and Tukey [1965], whose work popularized the algorithm.

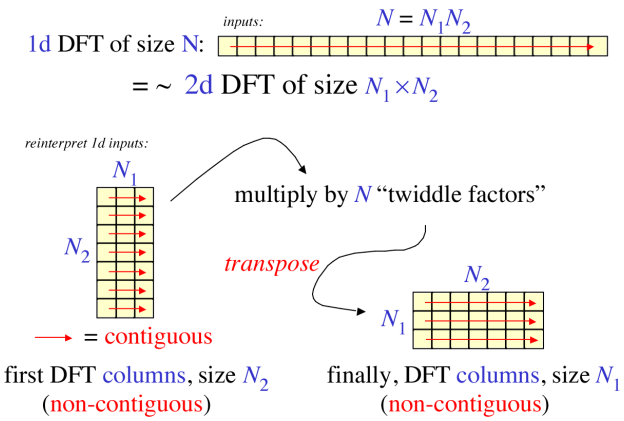

Fig. 26. Factored DFT [Johnson 2010]

Given a sequence of complex numbers, $x_0, \ldots, x_{N-1}$, the DFT is defined as

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi kn}{N}}, \quad \text{for } 0 \le k < N$$

Naively implemented, this DFT definition leads to quadratic work. The main trick to FFT is to factor N and then optimize the DFT definition, removing some exponentials that turn out to be equal to one. For $N = N_1 N_2$,

$$X_k = \sum_{n_1=0}^{N_1-1} \left[e^{-\frac{2\pi i}{N} n_1 k_2}\right] \left(\sum_{n_2=0}^{N_2-1} x_{N_1 n_2 + n_1} e^{-\frac{2\pi i}{N_2} n_2 k_2} \right) e^{-\frac{2\pi i}{N_1} n_1 k_1}$$

In this form, we can see two smaller sets of DFTs: N_1 of size N_2 each, and N_2 of size N_1 each. If we use the same method for solving these $N_1 + N_2$ smaller DFTs, we get a recursive FFT algorithm, visually outlined in Figure 26.

Rather than implementing FFT via sequences or arrays as usual, let's take a step back and consider a more structured approach.

4.2 Factor Types, not Numbers!

The summation formula above exhibits a trait typical of array-based algorithms, namely index arithmetic, which is tedious to write and to read. This arithmetic has a purpose, however, which is to interpret an array as an array of arrays. In a higher-level formulation, we might replace arrays and index arithmetic by an *explicit* nesting of structures. We have already seen the fundamental building block of structure nesting, namely functor composition. Instead of factoring numbers that represent type sizes, factor the types themselves.

As with scan, we can define a class of FFT-able structures and a generic default. One new wrinkle is that the result shape differs from the original shape, so we'll use an associated functor "*FFO*":

```
class FFT f where
  type FFO f :: * → *
  fft :: f ℂ → FFO f ℂ
  default fft :: (Generic1 f, Generic1 (FFO f), FFT (Rep1 f), FFO (Rep1 f) ~ Rep1 (FFO f))
```

$$\Rightarrow f\ \mathbb{C} \to FFO\ f\ \mathbb{C}$$
$$\mathit{fft}\ xs = to_1 \circ \mathit{fft}\ xs \circ from_1$$

Again, instances for U_1 and Par_1 are easy to define (exercise). We will *not* be able to define an instance for $f \times g$. Instead, for small functors, such as short vectors, we can simply use the DFT definition. The uniform pair case simplifies particularly nicely:

instance *FFT Pair* **where**
  type *FFO Pair* = *Pair*
  $\mathit{fft}\ (a :\# b) = (a + b) :\# (a - b)$

The final case is $g \circ f$, which is the heart of FFT. Figure 26 tells us almost all we need to know, leading to the following definition:

instance ... $\Rightarrow$ *FFT* $(g \circ f)$ **where**
  type *FFO* $(g \circ f)$ = *FFO* $f \circ$ *FFO* g
  $\mathit{fft} = Comp_1 \circ \mathit{ffts}' \circ \mathit{transpose} \circ \mathit{twiddle} \circ \mathit{ffts}' \circ unComp_1$

where *ffts′* performs several non-contiguous FFTs in parallel:

$\mathit{ffts}' :: \ldots \Rightarrow g\ (f\ \mathbb{C}) \to FFO\ g\ (f\ \mathbb{C})$
$\mathit{ffts}' = \mathit{transpose} \circ \mathit{fmap}\ \mathit{fft} \circ \mathit{transpose}$

Finally, the "twiddle factors" are all powers of a primitive N^{th} root of unity:

$\mathit{twiddle} :: \ldots \Rightarrow g\ (f\ \mathbb{C}) \to g\ (f\ \mathbb{C})$
$\mathit{twiddle} = (liftA_2 \circ liftA_2)\ (*)\ \mathit{omegas}$

$\mathit{omegas} :: \ldots \Rightarrow g\ (f\ (\mathit{Complex}\ a))$
$\mathit{omegas} = \mathit{fmap}\ \mathit{powers}\ (\mathit{powers}\ (\mathit{exp}\ (-i * 2 * \pi\ /\ \mathit{fromIntegral}\ (\mathit{size}\ @(g \circ f)))))$

The *size* method calculates the size of a structure. Unsurprisingly, the size of a composition is the product of the sizes.

The complexity of *fft* depends on the complexities of *twiddle* and *omegas*. Since *powers* (defined in Section 3.6) is a prefix scan, we can compute *omegas* efficiently in parallel, with one *powers* for g and then one more for each element of the resulting $g\ \mathbb{C}$, the latter collection being constructed in parallel. Thanks to scanning on constant structures, *powers* requires only linear work even on top-down trees (normally $O(n \log n)$). Depth of *powers* is logarithmic.

$$\begin{aligned} W_{omegas}\ (g\ (f\ \mathbb{C})) &= O(|g| + |g| \cdot |f|) = O(|g \circ f|) \\ D_{omegas}\ (g\ (f\ \mathbb{C})) &= log_2\ |g| + log_2\ |f| \\ &= log_2\ (|g| \cdot |f|) \\ &= log_2\ |g \circ f| \end{aligned}$$

After constructing *omegas*, *twiddle* multiplies two $g\ (f\ \mathbb{C})$ structures element-wise, with linear work and constant depth.

$$\begin{aligned} W_{twiddle}\ (g\ (f\ \mathbb{C})) &= W_{omegas}\ (g\ (f\ \mathbb{C})) + O(|g \circ f|) \\ &= O(|g \circ f|) + O(|g \circ f|) \\ &= O(|g \circ f|) \\ D_{twiddle}\ (g\ (f\ \mathbb{C})) &= D_{omegas}\ (g\ (f\ \mathbb{C})) + O(1) \\ &= log_2\ |g \circ f| + O(1) \end{aligned}$$

Returning to *fft* $@(g \circ f)$, the first *ffts′* (on $g \circ f$) does $|f|$ many *fft* on g (thanks to *transpose*), in parallel (via *fmap*). The second *ffts′* (on $f \circ g$) does $|g|$ many *fft* on f, also in parallel. (Since *transpose* is optimized away entirely, it is assigned no cost.) Altogether,

$$\begin{aligned} W\ (g \circ f) &= |g| \cdot W\ f + W_{twiddle}\ (g\ (f\ \mathbb{C})) + |f| \cdot W\ g \\ &= |g| \cdot W\ f + O(|g \circ f|) + |f| \cdot W\ g \\ D\ (g \circ f) &= D_{ffts'}\ (g\ (f\ \mathbb{C})) + D_{twiddle}\ (g\ (f\ \mathbb{C})) + D_{ffts'}\ (f\ (g\ \mathbb{C})) \\ &= D\ g + log_2\ |g \circ f| + O(1) + D\ f \end{aligned}$$

Note the symmetry of these results, so that $W\ (g \circ f) = W\ (f \circ g)$ and $D\ (g \circ f) = D\ (f \circ g)$. For this reason, FFT on top-down and bottom-up trees will have the same work and depth complexities.

The definition of *fft* for $g \circ f$ can be simplified (without changing complexity):

$$\begin{aligned} & Comp_1 \circ ffts' \circ transpose \circ twiddle \circ ffts' \circ unComp_1 \\ \equiv\ & \{\text{- definition of } ffts' \text{ (and associativity of } (\circ)) \text{ -}\} \\ & Comp_1 \circ transpose \circ fmap\ fft \circ transpose \circ transpose \circ twiddle \circ transpose \circ fmap\ fft \\ & \quad \circ transpose \circ unComp_1 \\ \equiv\ & \{\text{- } transpose \circ transpose \equiv id \text{ -}\} \\ & Comp_1 \circ transpose \circ fmap\ fft \circ twiddle \circ transpose \circ fmap\ fft \circ transpose \circ unComp_1 \\ \equiv\ & \{\text{- } transpose \circ fmap\ h \equiv traverse\ h \text{ -}\} \\ & Comp_1 \circ traverse\ fft \circ twiddle \circ traverse\ fft \circ transpose \circ unComp_1 \end{aligned}$$

4.3 Comparing Data Types

The top-down and bottom-up tree algorithms correspond to two popular binary FFT variations known as "decimation in time" and "decimation in frequency" ("DIT" and "DIF"), respectively. In the array formulation, these variations arise from choosing N_1 small or N_2 small, respectively (most commonly 2 or 4). Consider top-down trees first, starting with work:

$$\begin{aligned} W\ (RPow\ h\ 0) &= W\ Par_1 = 0 \\ W\ (RPow\ h\ (S\ n)) &= W\ (h \circ RPow\ h\ n) \\ &= |h| \cdot W\ (RPow\ h\ n) + O(|h \circ RPow\ h\ n|) + |RPow\ h\ n| \cdot W\ h \\ &= |h| \cdot W\ (RPow\ h\ n) + O(|h|^{S\ n}) + |RPow\ h\ n| \cdot W\ h \\ &= |h| \cdot W\ (RPow\ h\ n) + O(|RPow\ h\ n|) \end{aligned}$$

By the Master Theorem,

$$W\ (RPow\ h\ n) = O(|RPow\ h\ n| \cdot log\ |RPow\ h\ n|)$$

Next, depth:

$$\begin{aligned} D\ (RPow\ h\ 0) &= D\ Par_1 = 0 \\ D\ (RPow\ h\ (S\ n)) &= D\ (h \circ RPow\ h\ n) \\ &= D\ h + D\ (RPow\ h\ n) + log_2\ |h \circ RPow\ h\ n| + O(1) \\ &= D\ h + D\ (RPow\ h\ n) + log_2\ (|h|^{S\ n}) + O(1) \\ &= D\ (RPow\ h\ n) + O(n) \end{aligned}$$

Thus,

$$D\ (RPow\ h\ n) = O(n^2) = O(log^2\ |RPow\ h\ n|)$$

As mentioned above, $W\ (g \circ f) = W\ (f \circ g)$ and $D\ (g \circ f) = D\ (f \circ g)$, so top-down and bottom-up trees have the same work and depth complexities.

Next, consider bushes.

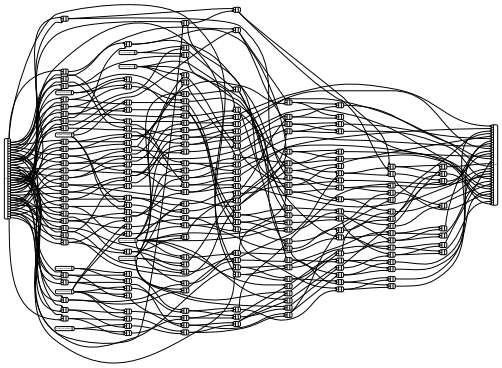

Fig. 27. *fft* @(*RBin* 4) *[W=197, D=8]*

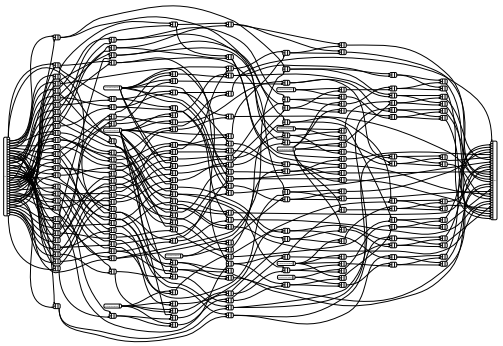

Fig. 28. *fft* @(*LBin* 4) *[W=197, D=8]*

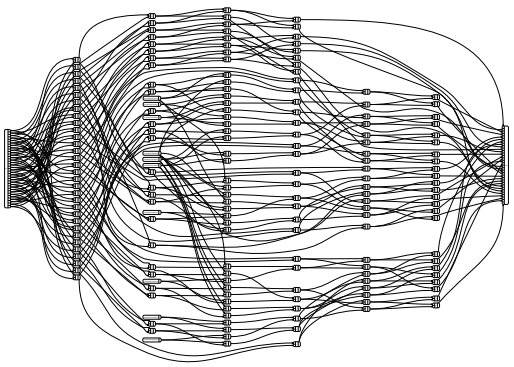

Fig. 29. *fft* @(*Bush* 2) *[W=186, D=6]*

	+	−	×	total	depth
RBin 4	74	74	40	197	8
LBin 4	74	74	40	197	8
Bush 2	72	72	32	186	6

Fig. 30. FFT for 16 complex values

	+	−	×	total	depth
RBin 8	2690	2690	2582	8241	20
LBin 8	2690	2690	2582	8241	20
Bush 3	2528	2528	1922	7310	14

Fig. 31. FFT for 256 complex values

$$
\begin{aligned}
W\ (Bush\ 0) &= W\ Pair = 2 \\
W\ (Bush\ (S\ n)) &= W\ (Bush\ n \circ Bush\ n) \\
&= |Bush\ n| \cdot W\ (Bush\ n) + O(|Bush\ n \circ Bush\ n|) + |Bush\ n| \cdot W\ (Bush\ n) \\
&= 2 \cdot |Bush\ n| \cdot W\ (Bush\ n) + O(|Bush\ (S\ n)|) \\
&= 2 \cdot 2^{2^n} \cdot W\ (Bush\ n) + O(2^{2^{n+1}}) \\
D\ (Bush\ 0) &= D\ Pair = 1 \\
D\ (Bush\ (S\ n)) &= D\ (Bush\ n \circ Bush\ n) \\
&= D\ (Bush\ n) + log_2\ |Bush\ n \circ Bush\ n| + O(1) + D\ (Bush\ n) \\
&= 2\ D\ (Bush\ n) + 2^{n+1} + O(1)
\end{aligned}
$$

Closed form solutions are left for later work.

Figures 27 and 28 show *fft* for top-down and bottom-up binary trees of depth four, and Figure 29 for bushes of depth two and three, all three of which types contain 16 elements. Each complex number appears as its real and imaginary components. Figures 30 and 31 give an empirical comparison. The total counts include literals, many of which are non-zero only accidentally, due to numerical inexactness. Pleasantly, the *Bush* instance of generic FFT appears to improve over the classic DIT and DIF algorithms in both work and depth.

5 RELATED WORK

Much has been written about parallel scan from a functional perspective. Blelloch [1996, Figure 11] gave a functional implementation of work-efficient of the algorithm of Ladner and Fischer [1980] in the functional parallel language NESL. O'Donnell [1994] presented an implementation in Haskell of what appears to the algorithm of Sklansky [1960], along with an equational correctness proof. Sheeran [2007, 2011] reconstructed the algorithms of Sklansky [1960], Ladner and Fischer [1980], and Brent and Kung [1982], generalized the latter two algorithms, and used dynamic programming to search the space defined by the generalized Ladner-Fischer algorithm, leading to a marked improvement in efficiency. (One can speculate on how to set up a search problem in the context of the generic, type-directed scan formulation given in the present paper, perhaps searching among functors isomorphic to arrays of statically known size.) Hinze [2004] developed an elegant algebra of scans, noting that "using only two basic building blocks and four combinators all standard designs can be described succinctly and rigorously." Moreover, the algebra is shown to be amenable to proving and deriving circuit designs. All of the work mentioned in this paragraph so far formulate scan exclusively in terms of lists, unlike the generic approach explored in the present paper. In contrast, Gibbons [1993, 2000] generalized to other data types, including trees, and reconstructed scan as a combination of the two more general operations of upward and downward accumulations. Keller and Chakravarty [1999] described a distributed scan algorithm similar to some of those emerging from the generic algorithm of Section 3 above, pointing out the additional scan and adjustment required to combine results of scanned segments.

FFT has also been studied through a functional lens, using lists or arrays. de Vries [1988] developed an implementation of fast polynomial multiplication based on binary FFT. Hartel and Vree [1992] assessed the convenience and efficiency of lazy functional array programming. Keller et al. [2010] gave a binary FFT implementation in terms of shape-polymorphic, parallel arrays, using index manipulations. Jones [1989, 1991] derived the Cooley/Tukey FFT algorithm from the DFT (discrete Fourier transform) definition, using lists of lists, which were assumed rectangular. (Perhaps such a derivation could be simplified by using type structure in place of lists and arithmetic.) Jay [1993] explored a categorical basis for tracking the static sizes of lists (and hence list-of-lists rectangularity) involved in computations like FFT. Berthold et al. [2009] investigated use of skeletons for parallel, distributed memory implementation of list-based FFT, mainly binary versions, though also mentioning other uniform and mixed radices. Various skeletons defined strategies for distributing work. Gorlatch [1998] applied his notion of "distributable homomorphisms" specialized to the FFT problem, reproducing common FFT algorithms. Sharp and Cripps [1993] transformed a DFT implementation to efficient an FFT in the functional language Hope$^+$. One transformation path led to a general functional execution platform, while other paths partially evaluated with respect to the problem size and generated feed-forward static process networks for execution on various static architectures. Bjesse et al. [1998] formulated the decimation-in-time and decimation-in-frequency FFT algorithms in the Haskell-embedded hardware description language Lava, producing circuits, executions, and correctness proofs. Frigo [1999] developed a code generator in OCAML for highly efficient FFT implementations for any size (not just powers of two or even composite). Similarly, Kiselyov et al. [2004] developed an FFT algorithm in MetaOCAML for static input sizes, using explicit staging and sharing.

6 REFLECTIONS

The techniques and examples in this paper illustrate programming parallel algorithms in terms of six simple, fundamental functor building blocks (sum, product, composition, and the three corresponding identities). This "generic" style has several advantages over the conventional practice of

designing and implementing parallel algorithms in terms of arrays. Banishing arrays does away with index calculations that obscure most presentations and open the door to run-time errors. Those dynamic errors are instead prevented by static typing, and the consequent index-free formulations more simply and directly capture the essential idea of the algorithm. The standard functor building blocks also invite use of the functionality of standard type classes such as *Functor*, *Foldable*, *Traversable*, and *Applicative*, along with the elegant and familiar programming and reasoning tools available for those patterns of computation, again sweeping away details to reveal essence. In contrast, array-based formulations involve indirect and error-prone emulations of operations on implicit compositions of the simpler types hiding behind index calculations for reading and writing array elements.

A strength of the generic approach to algorithms is that it is much easier to formulate data types than correct algorithms. As long as a data type can be modeled in terms of generic components having instances for the problem being solved, a correct, custom algorithm is assembled for that type automatically. The result may or may not be very parallel, but it is easy to experiment. Moreover, the same recipes that assemble data types and algorithms, also assemble analyses of work and depth complexity in the form of recurrences to be solved.

Of the six generic building blocks, the star of the show in this paper is functor composition, where the hearts of scan and FFT are both to be found. By using just compositions of uniform pairs, we are led to rediscover two well-known, parallel-friendly algorithms for each of scan and FFT. While functor composition is associative up to isomorphism, different associations give rise to different performance properties. Consistent right association leads to the common "top-down" form of perfect binary leaf trees, while consistent left association leads to a less common "bottom-up" form. For generic scan, the purely right-associated compositions followed by simple automatic optimizations become the well-known algorithm first discovered by Sklansky [1960], while the purely left-associated compositions and automatic optimizations become the more work-efficient algorithm of Ladner and Fischer [1980]. Conventional formulations of these algorithms center on arrays and, in retrospect, contain optimizations that obscure their essential natures and the simple, deep duality between them. Sklansky's scan algorithm splits an array of size N into two, performs two recursive scans, and adjusts the second resulting array. Ladner and Fischer's scan algorithm sums adjacent pairs, performs *one* recursive scan, and then interleaves the one resulting array with a modified version of it. In both cases, the post-recursion adjustment step turns out to be optimized versions of additional recursive scans, followed by the same kind of simple, uniform adjustment. Making these extra, hidden scans explicit reveals the close relationship between these two algorithms. The applied optimization is merely removal of zero additions (more generally combinations with monoid identity) and is easily automated. The duality between Sklansky's parallel scan and Ladner and Fischer's is exactly mirrored in the duality between two of the FFT algorithms, commonly known as "decimation in time" (right-associated functor composition) and "decimation in frequency" (left-associated functor composition).

Not only do we see the elegant essence and common connections between known algorithms—satisfying enough in its own right—but this insight also points the way to many infinitely many variations of these algorithms by varying the functors being composed beyond uniform pairs *and* varying the pattern of composition beyond *uniform* right or left association. This paper merely scratches the surface of the possible additional variations in the form of fully balanced compositions of the pair functor, as a type of uniform "bushes". Even this simple and perhaps obvious idea appears to provide a useful alternative. For scan, bushes offer a different compromise between top-down trees (best in work and worst in depth) vs bottom-up trees (best in depth and worst in work), coming in second place for both work and depth. With FFT, the complexity story seems to be uniformly positive, besting top-down and bottom-up in both work and depth, though at the cost

of less flexibility in data set size, since bushes of have sizes of the form 2^{2^n}, compared with 2^n for binary trees.

There are many more interesting questions to explore. Which other known scan and FFT algorithms emerge from the generic versions defined in this paper, specialized to other data types? Are there different instances for the *generic* functor combinators that lead to different algorithms for the data types used above? How does generic scan relate to the scan algebra of Hinze [2004], which is another systematic way to generate scan algorithms? What other problems are amenable to the sort of generic formulation in this paper? What other data types (functor assembly patterns) explain known algorithms and point to new ones?

REFERENCES

Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring. *Datatype-Generic Programming: International Spring School, Revised Lectures*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, April 2007.

Jost Berthold, Mischa Dieterle, Oleg Lobachev, and Rita Loogen. Parallel FFT with Eden skeletons. In *International Conference on Parallel Computing Technologies*, PaCT '09, pages 73–83, 2009.

Richard Bird and Lambert Meertens. Nested Datatypes. In *Mathematics of Program Construction*, pages 52–67, 1998.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *International Conference on Functional Programming*, pages 174–184, 1998.

Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, 1996.

R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, 31(3), March 1982.

James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

G.C. Danielson and C. Lanczos. Some improvements in practical Fourier analysis and their application to X-ray scattering from liquids. *Journal of the Franklin Institute*, 233(5):435–452, 1942.

Fer-Jan de Vries. A functional program for the Fast Fourier Transform. *SIGPLAN Notices*, pages 67–74, January 1988.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, pages 671–684, 2014.

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. Visible type application. In *European Symposium on Programming Languages and Systems*, pages 229–254, 2016.

Conal Elliott. Denotational design with type class morphisms. Technical Report 2009-01, LambdaPix, March 2009.

Conal Elliott. Compiling to categories. *Proc. ACM Program. Lang.*, 1(ICFP), September 2017.

Matteo Frigo. A fast Fourier transform compiler. In *PLDI*, volume 34, pages 169–180. ACM, May 1999.

Jeremy Gibbons. Upwards and downwards accumulations on trees. In *Mathematics of Program Construction*, 1993.

Jeremy Gibbons. Generic downwards accumulations. *Science of Computer Programming*, 37(1-3):37–65, May 2000.

Sergei Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *The Journal of Supercomputing*, 1998.

Pieter H. Hartel and Willem G. Vree. Arrays in a lazy functional language — a case study: The fast Fourier transform. In *2nd Arrays, functional languages, and parallel systems (ATABLE)*, 1992.

Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, October 1984.

Ralf Hinze. Memo functions, polytypically! In *2nd Workshop on Generic Programming*, pages 17–32, 2000.

Ralf Hinze. An algebra of scans. In *International Conference on Mathematics of Program Construction*, pages 186–210, 2004.

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

John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

Darius Jahandarie, Conor McBride, and João Cristóvão. newtype-generics, 2014. Haskell library.

C. Barry Jay. Matrices, monads and the fast Fourier transform. Technical Report UTSSOCS-93.13, University of Technology, Sydney, 1993.

Steven G. Johnson. Diagram to illustrate the general Cooley-Tukey FFT algorithm, 2010. URL https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm#General_factorizations.

Geraint Jones. Deriving the fast Fourier algorithm by calculation. In *Glasgow Workshop on Functional Programming*, 1989.

Geraint Jones. A fast flutter by the Fourier transform. In *IV Higher Order Workshop, Banff 1990*, pages 77–84, 1991.

Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In *Parallel and Distributed Processing*, pages 108–122, 1999.

Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, and Simon Peyton. Regular, shape-polymorphic, parallel arrays in Haskell. In *International Conference on Functional Programming*, 2010.

Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *EMSOFT*, 2004.

Richard E Ladner and Michael J Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–838, 1980.

José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. A generic deriving mechanism for Haskell. In *Haskell Symposium*, pages 37–48, 2010.

José Pedro Magalhães and Andres Löh. A formal comparison of approaches to datatype-generic programming. In *Workshop on Mathematically Structured Functional Programming*, pages 50–67, March 2012.

José Pedro Magalhães et al. GHC.Generics, 2011. URL https://wiki.haskell.org/GHC.Generics. Haskell wiki page.

Conor McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract), 2001. Unpublished.

Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

John T. O'Donnell. A correctness proof of parallel scan. *Parallel Processing Letters*, 04(03):329–338, 1994.

David Sharp and Martin Cripps. Synthesis of the fast Fourier transform algorithm by functional language program transformation. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 136–143, January 1993.

Mary Sheeran. Parallel prefix network generation: An application of functional programming. In *Hardware Design and Functional Languages*, 2007.

Mary Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *Journal of Functional Programming*, 21(1):59–114, January 2011.

J. Sklansky. Conditional-sum addition logic. *IRE Transactions on Electronic Computers*, EC-9(2):226–231, June 1960.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Workshop on types in language design and implementation*, 2012.