Data-Parallel Programming without Arrays

Conal Elliott

Early draft of October 10, 2018

1 Introduction

Despite its overwhelming popularity, the array type has serious drawbacks for parallel programming. In brief, array algorithms are unsafe (subject to out-of-bounds errors), weakly compositional, and brittle to change. A generic functor approach solves these problems, resulting in a programming style that is safe and strongly compositional (for code reuse), while robustly describing infinite families of guaranteed-correct algorithmic variations [Elliott, 2017].

For CPU-based and (especially) hardware implementations, the generic functor style of programming can perform fairly well. GPUs and their supporting programming models, however, have a very strong bias toward array programming. In particular, they efficiently support only "flat" data parallelism, corresponding to computing over arrays of scalar values. For this reason, Guy Blelloch and others have investigated automatic flattening of nested data parallelism, although they kept the array as central data type.

This note describes a design for a programming interface isomorphic to the generic functor composition style and an implementation that maps to efficient (I hope!) GPU-style array computations. The generated computations are guaranteed safe from out-of-bound errors despite using unsafe array operations internally, and the programming model remains elegantly compositional and generic-friendly.

2 Arrays and functors

An array is a sort of memoized version of a function over a finite, linearly ordered domain, usually taken to be $\{0, \ldots, n-1\}$ for some n. Let's assume that we have an efficient implementation of arrays, e.g., from the *vector* package [Leshchinskiy, 2017]:

type $\mathit{Vector}\ a = \ldots$ -- Dynamically sized vectors

$\mathit{index} :: \mathit{Vector}\ a \to \mathit{Int} \to a$
...

This *Vector* type has two major drawbacks:

- It is unsafe, since $\mathit{index}\ v\ i$ can be applied even when i is out of bounds. Although indices can be checked against a recorded size, client code might fail to do so properly and must deal with erroneous indexing when detected.
- It provides no compositional structure to guide algorithm design.

The first problem stems from $\mathit{index}\ v$ being a *partial* function from *Int*. One can instead make indexing be a *total* function from an explicit type of bounded natural numbers, resulting in an interface like that of the *vector-sized* package [Hermaszewski and Gamari, 2017]:

type $\mathit{Vector}\ n\ v$ -- Statically sized vectors

$\mathit{index} :: \mathit{Vector}\ n\ v \to \mathit{Finite}\ n \to v$
...

The type $\mathit{Finite}\ n$ represents the natural numbers less than n, i.e., $\{0, \ldots, n-1\}$. With *Vector*, we at least have a chance of using the type-checker to catch index out-of-bounds errors. Since Haskell doesn't yet have full dependent types, n here is a *type-level* natural number [ref], and we have only weak support for automatically checked safety proofs.

The second problem is subtler. Array-based algorithms typically involve index arithmetic that obscures the essence of an algorithm, is difficult to get correct, and is usually not validated automatically (e.g., by type checking) [Elliott, 2017]. Index arithmetic can be seen as mediating between a more natural data structure (e.g., trees) and its underlying representation (arrays). Choosing a more suitable representation eliminates the encoding and decoding, revealing the essence of the algorithm. The more natural data structures often correspond to memoized forms of functions, with these memoized forms arising from a small algebra of functors, including product and composition of functors and the two corresponding identities [Hinze, 2000; Elliott, 2017].

Programming in the functor vocabulary retains safety while gaining a natural programming style, free of any index calculations. The functor style, however, moves us further from SIMD-style architectures—including GPUs—which work with flat, *Int*-indexed arrays.

We now come to the main idea of this paper. To combine the benefits of the functor style (safety and compositionality) *and* of the flat array style (fast execution on SIMD architectures), combine the functor *interface* with the sized vector *representation* to form a type of "flattened functors":[1]

$$\mathbf{newtype}\ \mathit{Flat}\ f\ a = \mathit{Flat}\ (\mathit{Vector}\ |\mathit{Rep}\ f|\ a)$$

The $|\cdot|$ type family assigns cardinalities to types. The associated type $\mathit{Rep}\ f$ satisfies the property that $\forall a. f\ a \cong \mathit{Rep}\ f \to a$. In words, f is a "representable functor" isomorphic to functions from the associated domain type $\mathit{Rep}\ f$.[2] This associated type comes from the *Representable* class [Kmett, 2018, *Data.Functor.Rep*], which also provides the isomorphism as a pair of methods:

$$\begin{array}{l}
\mathbf{class}\ \mathit{Representable}\ f\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Rep}\ f \\
\quad \mathit{tabulate} :: (\mathit{Rep}\ f \to a) \to f\ a \\
\quad \mathit{index} :: f\ a \to (\mathit{Rep}\ f \to a)
\end{array}$$

The idea here is to think in terms of a representable functor f but represent as $\mathit{Flat}\ f$, which is isomorphic, shown as follows:

$$\begin{array}{ll}
 & f\ a \\
\cong & \{\ \text{the } \mathit{Representable} \text{ isomorphism}\ \} \\
 & \mathit{Rep}\ f \to a \\
\cong & \{\ \text{equal cardinality domains}\ \} \\
 & \mathit{Finite}\ |\mathit{Rep}\ f| \to a \\
\cong & \{\ \forall n. \mathit{Rep}\ (\mathit{Vector}\ n) = \mathit{Finite}\ n\ \} \\
 & \mathit{Vector}\ |\mathit{Rep}\ f|\ a \\
\cong & \{\ \text{definition of } \mathit{Flat}\ \} \\
 & \mathit{Flat}\ f\ a
\end{array}$$

We can define the isomorphism concretely as a pair of mutually-inverse functions between $f\ a$ and $\mathit{Flat}\ f\ a$. First, however, we'll need to look at isomorphisms more generally and isomorphisms with $\mathit{Finite}\ n$ in particular.

3 Isomorphisms

[Rethink the order of subsections below.]

3.1 Basic isomorphisms

An isomorphism between types a and b is witnessed by a pair of functions $f :: a \to b$ and $f' :: b \to a$ such that $f \circ f' = \mathit{id}$ and $f' \circ f = \mathit{id}$.[3]

[1] [Consider an intermediate step of using a type alias: $\mathbf{type}\ \mathit{Flat}\ f\ a = \mathit{Vector}\ (\mathit{Card}\ (\mathit{Rep}\ f))\ a$. We wouldn't get an infinite family of correct algorithms as in generic parallel programming, but we still get safety.]

[2] [Refer also to "Naperian functors".]

[3] The "$\rightleftharpoons$" symbol here is a data constructor, so an $a \cong_k b$ has a field of type $a \to b$ and another of type $b \to a$.

$$\textbf{data}\ a \cong b = (a \to b) \rightleftharpoons (b \to a)$$

This definition extends beyond functions to any category:

$$\textbf{data}\ a \cong_k b = (a \,`k`\, b) \rightleftharpoons (b \,`k`\, a)$$

Then $(\cong)$ then becomes a simple specialization:

$$\textbf{type}\ a \cong b = a \cong_{(\to)} b$$

Although arrow inversion is not computable, we can still use it in a *specification* that relates arrows (e.g., functions) in k to arrows in $(\cong_k)$:

$$\begin{array}{l} \mathit{iso} :: (a \,`k`\, b) \to a \cong_k b \quad \text{-- non-computable specification} \\ \mathit{iso}\ f = f \rightleftharpoons f^{-1} \end{array}$$

It will sometimes be convenient to extract the halves of an isomorphism:

$$\begin{array}{l} \mathit{isoFwd} :: a \cong_k b \to (a \,`k`\, b) \\ \mathit{isoFwd}\ (f \rightleftharpoons \_) = f \end{array}$$

$$\begin{array}{l} \mathit{isoRev} :: a \cong_k b \to (b \,`k`\, a) \\ \mathit{isoRev}\ (\_ \rightleftharpoons f') = f' \end{array}$$

Inversion is trivially definable for $(\cong_k)$:

$$\begin{array}{l} \mathit{inv} :: a \cong_k b \to b \cong_k a \\ \mathit{inv}\ (f \rightleftharpoons f') = f' \rightleftharpoons f \end{array}$$

We will sometimes need not just isomorphisms, but *natural isomorphisms*:

$$\textbf{type}\ f \mathrel{\dot{\cong}} g = \forall a. f\ a \cong g\ a$$

3.2 Composing isomorphisms

We will use *iso* to specify operations on $a \cong_k b$ and calculate correct implementations of those operations.

Theorem 1 (Proved in Appendix A.1)**.** Given the instance definitions in Figure 1, *iso* is a homomorphism with respect to each of the instantiated classes.[4,5]

It will also be convenient to specialize the bifunctor operations from Figure 1 into one-sided versions, as shown in Figure 2.

3.3 Arrow isomorphisms

Although Figure 1 defines instances of many categorical operations for $(\cong_k)$, it omits some important classes, due to non-invertibility:[6]

$$\begin{array}{l} \textbf{class}\ \mathit{MonoidalP}\ k \Rightarrow \mathit{Cartesian}\ k\ \textbf{where} \\ \quad \mathit{exl} :: (a \times b) \,`k`\, a \\ \quad \mathit{exr} :: (a \times b) \,`k`\, b \end{array}$$

$$\begin{array}{l} \textbf{class}\ \mathit{MonoidalS}\ k \Rightarrow \mathit{Cocartesian}\ k\ \textbf{where} \\ \quad \mathit{inl} :: a \,`k`\, (a + b) \end{array}$$

---

[4] [Try combining six classes into three (*Associative*, *Braided*, and *Monoidal*), e.g., **class** *Braided* k $(\odot)$ **where** $\mathit{swap} :: (a \odot b) \,`k`\, (b \odot a)$.]

[5] [Find a different symbol for $(\Rightarrow)$ so as not to clash with Haskell's use for constrained polymorphism.]

[6] The projections *exl* and *exr* fail surjectivity, while *inl*, *inr*, and *apply* fail injectivity. [Do isomorphisms *curry* and *uncurry* into isomorphisms?]

$$\begin{array}{l}
\textbf{class } \mathit{Category}\ k \textbf{ where} \\
\quad \mathit{id}\ :: a \mathbin{`k`} a \\
\quad (\circ) :: (b \mathbin{`k`} c) \to (a \mathbin{`k`} b) \to (a \mathbin{`k`} c)
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{Category}\ k \Rightarrow \mathit{Category}\ (\cong_k) \textbf{ where} \\
\quad \mathit{id} = \mathit{id} \rightleftharpoons \mathit{id} \\
\quad (g \rightleftharpoons g') \circ (f \rightleftharpoons f') = (g \circ f) \rightleftharpoons (f' \circ g')
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{AssociativeP}\ k \textbf{ where} \\
\quad \mathit{rassocP} :: ((a \times b) \times c) \mathbin{`k`} (a \times (b \times c)) \\
\quad \mathit{lassocP}\ :: (a \times (b \times c)) \mathbin{`k`} ((a \times b) \times c)
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{AssociativeP}\ k \Rightarrow \mathit{AssociativeP}\ (\cong_k) \textbf{ where} \\
\quad \mathit{lassocP} = \mathit{lassocP} \rightleftharpoons \mathit{rassocP} \\
\quad \mathit{rassocP} = \mathit{rassocP} \rightleftharpoons \mathit{lassocP}
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{BraidedP}\ k \textbf{ where} \\
\quad \mathit{swapP} :: (a \times b) \mathbin{`k`} (b \times a)
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{BraidedP}\ k \Rightarrow \mathit{BraidedP}\ (\cong_k) \textbf{ where} \\
\quad \mathit{swapP} = \mathit{swapP} \rightleftharpoons \mathit{swapP}
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{MonoidalP}\ k \textbf{ where} \\
\quad (\times)\ ::\ (a \mathbin{`k`} c) \to (b \mathbin{`k`} d) \\
\qquad\quad \to ((a \times b) \mathbin{`k`} (c \times d))
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{MonoidalP}\ k \Rightarrow \mathit{MonoidalP}\ (\cong_k) \textbf{ where} \\
\quad (f \rightleftharpoons f') \times (g \rightleftharpoons g') = (f \times g) \rightleftharpoons (f' \times g')
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{AssociativeS}\ k \textbf{ where} \\
\quad \mathit{rassocS} :: ((a + b) + c) \mathbin{`k`} (a + (b + c)) \\
\quad \mathit{lassocS}\ :: (a + (b + c)) \mathbin{`k`} ((a + b) + c)
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{AssociativeS}\ k \Rightarrow \mathit{AssociativeS}\ (\cong_k) \textbf{ where} \\
\quad \mathit{lassocS} = \mathit{lassocS} \rightleftharpoons \mathit{rassocS} \\
\quad \mathit{rassocS} = \mathit{rassocS} \rightleftharpoons \mathit{lassocS}
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{BraidedS}\ k \textbf{ where} \\
\quad \mathit{swapS} :: (a + b) \mathbin{`k`} (b + a)
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{BraidedS}\ k \Rightarrow \mathit{BraidedS}\ (\cong_k) \textbf{ where} \\
\quad \mathit{swapS} = \mathit{swapS} \rightleftharpoons \mathit{swapS}
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{MonoidalS}\ k \textbf{ where} \\
\quad (+)\ ::\ (a \mathbin{`k`} c) \to (b \mathbin{`k`} d) \\
\qquad\quad \to ((a + b) \mathbin{`k`} (c + d))
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{MonoidalS}\ k \Rightarrow \mathit{MonoidalS}\ (\cong_k) \textbf{ where} \\
\quad (f \rightleftharpoons f') + (g \rightleftharpoons g') = (f + g) \rightleftharpoons (f' + g')
\end{array}$$

$$\begin{array}{l}
\textbf{class } \mathit{Closed}\ k \textbf{ where} \\
\quad (\Rightarrow)\ ::\ (d \mathbin{`k`} c) \to (a \mathbin{`k`} b) \\
\qquad\quad \to ((c \Rightarrow a) \mathbin{`k`} (d \Rightarrow b))
\end{array}$$

$$\begin{array}{l}
\textbf{instance } \mathit{Closed}\ k \Rightarrow \mathit{Closed}\ (\cong_k) \textbf{ where} \\
\quad (p \rightleftharpoons p') \Rightarrow (q \rightleftharpoons q') = (p \Rightarrow q) \rightleftharpoons (p' \Rightarrow q')
\end{array}$$

Figure 1: Isomorphisms (calculated in Appendix A.1, specified by *iso* as homomorphism)

$$\begin{array}{l}
\mathit{first} :: \mathit{MonoidalP}\ k \Rightarrow (a \mathbin{`k`} c) \to ((a \times b) \mathbin{`k`} (c \times b)) \\
\mathit{first}\ f = f \times \mathit{id} \\[1ex]
\mathit{second} :: \mathit{MonoidalP}\ k \Rightarrow (b \mathbin{`k`} d) \to ((a \times b) \mathbin{`k`} (a \times d)) \\
\mathit{second}\ g = \mathit{id} \times g \\[1ex]
\mathit{left} :: \mathit{MonoidalS}\ k \Rightarrow (a \mathbin{`k`} c) \to ((a + b) \mathbin{`k`} (c + b)) \\
\mathit{left}\ f = f + \mathit{id} \\[1ex]
\mathit{right} :: \mathit{MonoidalS}\ k \Rightarrow (b \mathbin{`k`} d) \to ((a + b) \mathbin{`k`} (a + d)) \\
\mathit{right}\ g = \mathit{id} + g \\[1ex]
\mathit{dom} :: \mathit{Closed}\ k \Rightarrow (d \mathbin{`k`} c) \to ((c \Rightarrow a) \mathbin{`k`} (d \Rightarrow a)) \\
\mathit{dom}\ f = f \Rightarrow \mathit{id} \\[1ex]
\mathit{cod} :: \mathit{Closed}\ k \Rightarrow (a \mathbin{`k`} b) \to ((c \Rightarrow a) \mathbin{`k`} (c \Rightarrow b)) \\
\mathit{cod}\ g = \mathit{id} \Rightarrow g
\end{array}$$

Figure 2: One-sided specializations of product, coproduct, and exponential bifunctors

$$\quad \mathit{inr} :: b \text{ `}k\text{` } (a + b)$$

$$\begin{array}{l}\textbf{class } (\mathit{MonoidalP}\ k, \mathit{Closed}\ k) \Rightarrow \mathit{MonoidalClosed}\ k \textbf{ where}\\ \quad \mathit{apply} :: ((a \Rightarrow b) \times a) \text{ `}k\text{` } b\\ \quad \mathit{curry} :: ((a \times b) \text{ `}k\text{` } c) \to (a \text{ `}k\text{` } (b \Rightarrow c))\\ \quad \mathit{uncurry} :: (a \text{ `}k\text{` } (b \Rightarrow c)) \to ((a \times b) \text{ `}k\text{` } c)\end{array}$$

These *Cartesian* and *Cocartesian* instances give rise to two useful derived operations:

$$\begin{array}{l}(\triangle) :: \mathit{Cartesian}\ \ \ k \Rightarrow (a \text{ `}k\text{` } c) \to (a \text{ `}k\text{` } d) \to (a \to (c \times d))\\ (\triangledown) :: \mathit{Cocartesian}\ k \Rightarrow (c \text{ `}k\text{` } a) \to (d \text{ `}k\text{` } a) \to ((c + d) \text{ `}k\text{` } a)\end{array}$$

In uncurried form,

$$\begin{array}{l}\mathit{fork} :: \mathit{Cartesian}\ \ \ k \Rightarrow (a \text{ `}k\text{` } c) \times (a \text{ `}k\text{` } d) \to (a \to (c \times d))\\ \mathit{fork} = \mathit{uncurry}\ (\triangle)\end{array}$$

$$\begin{array}{l}\mathit{join} :: \mathit{Cocartesian}\ k \Rightarrow (c \text{ `}k\text{` } a) \times (d \text{ `}k\text{` } a) \to ((c + d) \text{ `}k\text{` } a)\\ \mathit{join} = \mathit{uncurry}\ (\triangledown)\end{array}$$

These uncurried versions have inverses:

$$\begin{array}{l}\mathit{unfork} :: \mathit{Cartesian}\ \ \ k \Rightarrow (a \text{ `}k\text{` } c) \times (a \text{ `}k\text{` } d) \to (a \to (c \times d))\\ \mathit{unfork}\ f = (\mathit{exl} \circ f, \mathit{exr} \circ f)\end{array}$$

$$\begin{array}{l}\mathit{unjoin} :: \mathit{Cocartesian}\ k \Rightarrow (c \text{ `}k\text{` } a) \times (d \text{ `}k\text{` } a) \to ((c + d) \text{ `}k\text{` } a)\\ \mathit{unjoin}\ f = (f \circ \mathit{inl}, f \circ \mathit{inr})\end{array}$$

Lemma 2 (Proved in Appendix A.2)**.** *fork* and *unfork* are inverses, as are *join* and *unjoin*.

We can thus package these pairs of inverses into isomorphisms:

$$\begin{array}{l}\mathit{forkIso} :: \mathit{Cocartesian}\ k \Rightarrow (a \text{ `}k\text{` } c) \times (a \text{ `}k\text{` } d) \cong (a \text{ `}k\text{` } (c \times d))\\ \mathit{forkIso} = \mathit{fork} \rightleftharpoons \mathit{unfork}\end{array}$$

$$\begin{array}{l}\mathit{joinIso} :: \mathit{Cartesian}\ k \ \ \ \Rightarrow (c \text{ `}k\text{` } a) \times (d \text{ `}k\text{` } a) \cong ((c + d) \text{ `}k\text{` } a)\\ \mathit{joinIso} = \mathit{join} \rightleftharpoons \mathit{unjoin}\end{array}$$

Likewise, *curry* and *uncurry* are always inverses:

$$\begin{array}{l}\mathit{curryIso} :: \mathit{MonoidalClosed}\ k \Rightarrow ((a \times b) \text{ `}k\text{` } c) \cong (a \text{ `}k\text{` } (b \to c))\\ \mathit{curryIso} = \mathit{curry} \rightleftharpoons \mathit{uncurry}\end{array}$$

3.4 Natural number isomorphisms

The notions of cardinality and isomorphism are tightly connected. Georg Cantor defined cardinality in terms of injections and isomorphisms [ref]. In particular, $|A| \leqslant |B|$ exactly when there is an injection from A to B, and $|A| = |B|$ exactly when there is an bijection from A to B. These definitions apply not only to finite sets and so laid the foundation for comparing cardinalities of infinite (even uncountably infinite) set, including Cantor's seminal result that there are strictly more real numbers than natural numbers.

Cardinality also relates the notions of sums, products, and exponentials on sets to sums, products, and exponentials on natural numbers:[7]

$$\begin{array}{l}|a + b| = |a| + |b|\\ |a \times b| = |a| \times |b|\\ |a \uparrow b| = |a| \uparrow |b|\end{array}$$

[7] [Either complete or remove the discussion of exponentials.]

$$
\begin{aligned}
&\mathbf{type}\ \mathit{KnownNat}_2\ m\ n = (\mathit{KnownNat}\ m, \mathit{KnownNat}\ n)\\[6pt]
&\mathit{finU}_1 :: \mathbf{0} \cong \mathit{Finite}\ 0\\
&\mathit{finU}_1 = \mathit{combineZero} \rightleftharpoons \mathit{separateZero}\\[6pt]
&\mathit{finPar}_1 :: \mathbf{1} \cong \mathit{Finite}\ 1\\
&\mathit{finPar}_1 = \mathit{combineOne} \rightleftharpoons \mathit{separateOne}\\[6pt]
&\mathit{finSum} :: \mathit{KnownNat}_2\ m\ n \Rightarrow \mathit{Finite}\ m + \mathit{Finite}\ n \cong \mathit{Finite}\ (m+n)\\
&\mathit{finSum} = \mathit{combineSum} \rightleftharpoons \mathit{separateSum}\\[6pt]
&\mathit{finProd} :: \mathit{KnownNat}_2\ m\ n \Rightarrow \mathit{Finite}\ m \times \mathit{Finite}\ n \cong \mathit{Finite}\ (m \times n)\\
&\mathit{finProd} = \mathit{combineProd} \rightleftharpoons \mathit{separateProd}\\[6pt]
&\mathit{finExp} :: \mathit{KnownNat}_2\ m\ n \Rightarrow \mathit{Finite}\ m \uparrow \mathit{Finite}\ n \cong \mathit{Finite}\ (m \uparrow n)\\
&\mathit{finExp} = \mathit{combineExp} \rightleftharpoons \mathit{separateExp}\\[6pt]
&\quad \text{-- ...}
\end{aligned}
$$

Figure 3: Isomorphisms involving *Finite*

where "$(\uparrow)$" refers to exponentiation on types in the LHS and on numbers in the RHS; and $a \uparrow b$ on types is more commonly written as "$b \rightarrow a$" or "$b \Rightarrow a$" ("exponentials" or "internal homs"). Let's focus on finite sets, and particularly $\mathit{Finite}\ n$, i.e., the natural numbers $\{0, \ldots, n-1\}$, reasoning as follows:

$$
\begin{aligned}
&\ |\mathit{Finite}\ (m+n)|\\
=&\ \{\ \text{defining property of } \mathit{Finite}\ \}\\
&\ m+n\\
=&\ \{\ \text{defining property of } \mathit{Finite}\ \}\\
&\ |\mathit{Finite}\ m| + |\mathit{Finite}\ n|\\
=&\ \{\ \text{above}\ \}\\
&\ |\mathit{Finite}\ m + \mathit{Finite}\ n|
\end{aligned}
$$

Likewise for products and exponentials as well as for 0 and 1. Summarizing,

$$
\begin{aligned}
|\mathit{Finite}\ 0| &= |\mathbf{0}|\\
|\mathit{Finite}\ 1| &= |\mathbf{1}|\\[6pt]
|\mathit{Finite}\ (m+n)| &= |\mathit{Finite}\ m + \mathit{Finite}\ n|\\
|\mathit{Finite}\ (m \times n)| &= |\mathit{Finite}\ m \times \mathit{Finite}\ n|\\
|\mathit{Finite}\ (m \uparrow n)| &= |\mathit{Finite}\ m \uparrow \mathit{Finite}\ n|
\end{aligned}
$$

where $\mathbf{0}$ is the empty type, and $\mathbf{1}$ is the unit type (usually written "*Void*" and "()" in Haskell). Equivalently,

$$
\begin{aligned}
\mathit{Finite}\ 0 &\cong \mathbf{0}\\
\mathit{Finite}\ 1 &\cong \mathbf{1}\\[6pt]
\mathit{Finite}\ (m+n) &\cong \mathit{Finite}\ m + \mathit{Finite}\ n\\
\mathit{Finite}\ (m \times n) &\cong \mathit{Finite}\ m \times \mathit{Finite}\ n\\
\mathit{Finite}\ (m \uparrow n) &\cong \mathit{Finite}\ m \uparrow \mathit{Finite}\ n
\end{aligned}
$$

Figure 3 defines these isomorphisms for later use.[8] Figure 4 defines the operations used to construct isomorphisms in Figure 3. These operations correspond to functionality in the *finite-typelits* package [mniip, 2017].[9]

[8] [Fill in for exponentiation. Also, I may want to reverse the sense of these isomorphisms. If so, change the code and then the paper.]

[9] [Explain some of the Haskellisms. Also my overloading of $(+)$ and $(\times)$ for sum and product of natural numbers and types. I'll also need to explain the *Finite* constructor. Maybe revisit my attempt to redefine the *Finite* type.]

$$
\begin{array}{l}
\mathit{combineZero} :: \mathbf{0} \rightarrow \mathit{Finite}\ 0 \\
\mathit{combineZero} = \mathit{absurd}
\end{array}
$$

$$
\begin{array}{l}
\mathit{separateZero} :: \mathit{Finite}\ 0 \rightarrow \mathbf{0} \\
\mathit{separateZero} = \mathit{error}\ \texttt{"no Finite 0"}\ \ \text{-- } \mathit{Revisit.}
\end{array}
$$

$$
\begin{array}{l}
\mathit{combineOne}\ :: \mathbf{1} \rightarrow \mathit{Finite}\ 1 \\
\mathit{combineOne} = \mathit{const}\ (\mathit{Finite}\ 0)
\end{array}
$$

$$
\begin{array}{l}
\mathit{separateOne}\ :: \mathit{Finite}\ 1 \rightarrow \mathbf{1} \\
\mathit{separateOne} = \mathit{const}\ ()
\end{array}
$$

$$
\begin{array}{l}
\mathit{combineSum} :: \forall m\ n. \mathit{KnownNat}_2\ m\ n \Rightarrow (\mathit{Finite}\ m + \mathit{Finite}\ n) \rightarrow \mathit{Finite}\ (m + n) \\
\mathit{combineSum}\ (\mathit{Left}\ \ \ (\mathit{Finite}\ l\,)) = \mathit{Finite}\ l \\
\mathit{combineSum}\ (\mathit{Right}\ (\mathit{Finite}\ k)) = \mathit{Finite}\ (\mathit{nat}\ @m + k)
\end{array}
$$

$$
\begin{array}{l}
\mathit{separateSum} :: \forall m\ n. \mathit{KnownNat}_2\ m\ n \Rightarrow \mathit{Finite}\ (m + n) \rightarrow (\mathit{Finite}\ m + \mathit{Finite}\ n) \\
\mathit{separateSum}\ (\mathit{Finite}\ l) \mid l < m \qquad\ \ = \mathit{Left}\ \ \ (\mathit{Finite}\ l) \\
\qquad\qquad\qquad\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Right}\ (\mathit{Finite}\ (l - m)) \\
\quad \mathbf{where} \\
\qquad m = \mathit{nat}\ @m
\end{array}
$$

$$
\begin{array}{l}
\mathit{combineProd} :: \forall m\ n. \mathit{KnownNat}_2\ m\ n \Rightarrow (\mathit{Finite}\ m \times \mathit{Finite}\ n) \rightarrow \mathit{Finite}\ (m \times n) \\
\mathit{combineProd}\ (\mathit{Finite}\ l, \mathit{Finite}\ k) = \mathit{Finite}\ (\mathit{nat}\ @n \times l + k)
\end{array}
$$

$$
\begin{array}{l}
\mathit{separateProd} :: \forall m\ n. \mathit{KnownNat}_2\ m\ n \Rightarrow \mathit{Finite}\ (m \times n) \rightarrow (\mathit{Finite}\ m \times \mathit{Finite}\ n) \\
\mathit{separateProd}\ (\mathit{Finite}\ l) = (\mathit{Finite}\ q, \mathit{Finite}\ r)\ \mathbf{where}\ (q, r) = l\ \text{`}\mathit{divMod}\text{`}\ \mathit{nat}\ @n
\end{array}
$$

Figure 4: Sum and product isomorphisms

Lemma 3. The functions defined in Figure 4 are pairs of inverses (*combineZero* with *separateZero*, etc), justifying their use in Figure 3.[10]

Another property will turn out to be very useful:

Lemma 4. The functions defined in Figure 4 are strictly monotonic.

Monotonic isomorphisms are also referred to as "order isomorphisms". While there may be many isomorphisms between two types, for finite, linearly ordered types, there is only one order isomorphism.[11,12] Together with invertibility, this monotonicity property thus *uniquely* determines the functions defined in Figure 4. The linear orderings assumed in these definitions agree with Haskell's *Ord* type class and standard instances, in which left injections are smaller than right injections, and products are ordered lexicographically:

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Ord}\ (a + b)\ \textbf{where} \\
\quad \mathit{Left}\ \ a < \mathit{Left}\ \ a' = a < a' \\
\quad \mathit{Left}\ \ a < \mathit{Right}\ b' = \mathit{True} \\
\quad \mathit{Right}\ b < \mathit{Left}\ \ a' = \mathit{False} \\
\quad \mathit{Right}\ b < \mathit{Right}\ b' = b < b'
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Ord}\ (a \times b)\ \textbf{where} \\
\quad (a, b) < (a', b') = a < a' \vee (a = a' \wedge b < b')
\end{array}
$$

Lemma 5 (Proved in Appendix A.3)**.** Monotonic functions on linear orders form a category that is monoidal under sums and products.[13,14]

3.5 Some other useful isomorphisms

While Figures 1 and 2 shows how to construct and compose arrows with a standard vocabulary, we will also need some primitives, which we can easily build whenever we have a pair of inverses. We've already seen one example in Section 2, namely the representability isomorphism:

$$
\begin{array}{l}
\mathit{repIso} :: \mathit{Representable}\ f \Rightarrow f\ a \cong (\mathit{Rep}\ f \to a) \\
\mathit{repIso} = \mathit{index} \rightleftharpoons \mathit{tabulate}
\end{array}
$$

Another example is found in the *Newtype* class from the package *newtype-generics* [Jahandarie et al., 2018]:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Newtype}\ n\ \textbf{where} \\
\quad \textbf{type}\ O\ n \\
\quad \mathit{pack} \quad\ :: O\ n \to n \\
\quad \mathit{unpack} :: n \to O\ n
\end{array}
$$

This class serves as a shared interface for the isomorphism between a **newtype**-defined data type and its underlying representation. We can wrap up instances:

$$
\begin{array}{l}
\mathit{newIso} :: \mathit{Newtype}\ n \Rightarrow n \cong O\ n \\
\mathit{newIso} = \mathit{unpack} \rightleftharpoons \mathit{pack}
\end{array}
$$

Closely related to *Newtype* is the *Coercible* class, which provides a way to perform safe, zero-cost conversions between types that share a common underlying representation [Breitner et al., 2014]. The safe coercion primitive is

$$
\mathit{coerce} :: \mathit{Coercible}\ a\ b \Rightarrow a \to b
$$

[10] [Can I calculate half of these functions from the others? The *combineX* functions are simpler, so start with them.]
[11] Proof sketch: the smallest value of one type must map to the smallest of the other, the next smallest to the next smallest, etc.
[12] [In what other settings are order isomorphisms unique? At least for well-ordered sets [proof].]
[13] [Move this lemma to a later section, and expand it.]
[14] [What about under exponentials?]

Instances of the *Coercible* class are synthesized automatically as needed by the compiler. One source of these instances is **newtype** definitions, but others include congruence rules so that, for instance, if $Coercible\ a\ b$ then $Coercible\ (f\ a)\ (f\ b)$, and so on for conversions involving arbitrarily nested **newtype** coercions. (There are some restrictions depending on the "role" of type parameters). Coercibility is also an equivalence relation (reflexive, symmetric, and transitive). Packaging as an isomorphism is straightforward:

$$\begin{aligned}&coerceIso :: Coercible\ a\ b \Rightarrow a \cong b\\&coerceIso = coerce \rightleftharpoons coerce\end{aligned}$$

While *coerceIso* is much more flexible, *newIso* requires fewer type annotations.

3.6 Reindexing representable functors

Let's now use our isomorphism vocabulary to form *reindexing* isomorphisms. Given representable functors f and g, suppose we have an isomorphism $h :: Rep\ g \cong Rep\ f$, converting between indices of g and f. Then f and g are also (naturally) isomorphic, witnessed as follows:[15]

$$\begin{aligned}&reindex :: (Representable\ f, Representable\ g) \Rightarrow (Rep\ g \cong Rep\ f) \to (f \mathrel{\dot{\cong}} g)\\&reindex\ h = inv\ repIso \circ dom\ h \circ repIso\end{aligned}$$

where *dom* is defined in Figure 2.[16] The types involved:

$$\begin{aligned}&repIso &&:: f\ a \cong (Rep\ f \to a)\\&dom\ h &&:: (Rep\ f \to a) \cong (Rep\ g \to a)\\&inv\ repIso &&:: (Rep\ g \to a) \cong g\ a\end{aligned}$$

This reindexing isomorphism will be exactly what we need to allow us to program in the style of generic functors but implement via flat data parallelism for SIMD execution.

Exercise 6. Show that *reindex* is a contravariant functor, i.e., $reindex\ id = id$, and $reindex\ (k \circ h) = reindex\ h \circ reindex\ k$ whenever these equations are type-correct. (When might they not be type-correct?)

There are some generally useful representable functors defined in *GHC.Generics* [Magalhães et al., 2011], shown with their *Representable* instances [Kmett, 2018] in Figure 5.[17,18]

3.7 Reshaping vectors

Figure 6 applies reindexing from Section 3.6 with index isomorphisms involving *Finite* from Section 3.4, recalling that $Rep\ (Vector\ n) = Finite\ n$. The RHSs of these isomorphisms involve the generic functor building blocks. The types involved:

$$\begin{aligned}finU_1 &:: \mathbf{0} \cong Finite\ 0\\&:: Rep\ U_1 \cong Rep\ (Vector\ 0)\\reindex\ finU_1 &:: U_1 \mathrel{\dot{\cong}} Vector\ 0\end{aligned}$$

$$\begin{aligned}finPar_1 &:: \mathbf{1} \cong Finite\ 1\\&:: Rep\ Par_1 \cong Rep\ (Vector\ 1)\\reindex\ finPar_1 &:: Par_1 \mathrel{\dot{\cong}} Vector\ 1\end{aligned}$$

$$\begin{aligned}finSum &:: Finite\ m + Finite\ n \cong Finite\ (m+n)\\&:: Rep\ (Vector\ m) + Rep\ (Vector\ n) \cong Rep\ (Vector\ (m+n))\end{aligned}$$

[15] [I think *reindex* is a contravariant functorial. I first made it covariant, but I think contravariant fits the intent better. It also saves me double inversion of *dom h*.]

[16] [Maybe note some specializations of *reindex* at this point, including $h = id$, $f = (\to)\ a$, and $g = (\to)\ b$. In these cases, $dom\ h = id$, $repIso = id$, and $inv\ repIso = id$, respectively.]

[17] [Drop a hint about why these definitions.]

[18] [Rewrite each *index* and *tabulate* pair via a single isomorphism than handles both elegantly? How to present nicely, considering that *Representable* has two methods instead of one isomorphism-valued method?]

```
newtype U1       a = U1                  -- unit
newtype Par1     a = Par1 a              -- singleton
data    (f × g)  a = f a × g a           -- product
newtype (g ∘ f)  a = Comp1 (g (f a))     -- composition

instance Representable U1 where
  type Rep U1 = 0
  index U1 = absurd
  tabulate _ = U1

instance Representable Par1 where
  type Rep Par1 = ()
  index (Par1 a) () = a
  tabulate f = Par1 (f ())

instance (Representable f, Representable g) ⇒ Representable (f × g) where
  type Rep (f × g) = Rep f + Rep g
  index (a × _) (Left  i) = index a i
  index (_ × b) (Right j) = index b j
  tabulate f = tabulate (f ∘ Left) × tabulate (f ∘ Right)

instance (Representable f, Representable g) ⇒ Representable (g ∘ f) where
  type Rep (g ∘ f) = Rep g × Rep f
  index (Comp1 gf) (j, i) = index (index gf j) i
  tabulate = Comp1 ∘ tabulate ∘ fmap tabulate ∘ curry
```

Figure 5: Some generic functors and their associated *Representable* instances

```
vecU1 :: Vector 0 ≅̇ U1
vecU1 = reindex finU1

vecPar1 :: Vector 1 ≅̇ Par1
vecPar1 = reindex finPar1

vecProd :: KnownNat2 m n ⇒ Vector (m + n) ≅̇ Vector m × Vector n
vecProd = reindex finSum

vecComp :: KnownNat2 m n ⇒ Vector (m × n) ≅̇ Vector m ∘ Vector n
vecComp = reindex finProd
```

Figure 6: Reshaping vectors

$$\begin{aligned} &:: \mathit{Rep}\ (\mathit{Vector}\ m \times \mathit{Vector}\ n) \cong \mathit{Rep}\ (\mathit{Vector}\ (m+n)) \\ \mathit{reindex}\ \mathit{finSum} &:: \mathit{Vector}\ (m+n) \mathrel{\dot{\cong}} \mathit{Vector}\ m \times \mathit{Vector}\ n \end{aligned}$$

$$\begin{aligned} \mathit{finProd} &:: \mathit{Finite}\ m \times \mathit{Finite}\ n \cong \mathit{Finite}\ (m \times n) \\ &:: \mathit{Rep}\ (\mathit{Vector}\ m) \times \mathit{Rep}\ (\mathit{Vector}\ n) \cong \mathit{Rep}\ (\mathit{Vector}\ (m \times n)) \\ &:: \mathit{Rep}\ (\mathit{Vector}\ m \circ \mathit{Vector}\ n) \cong \mathit{Rep}\ (\mathit{Vector}\ (m \times n)) \\ \mathit{reindex}\ \mathit{finProd} &:: \mathit{Vector}\ (m \times n) \mathrel{\dot{\cong}} \mathit{Vector}\ m \circ \mathit{Vector}\ n \end{aligned}$$

In the forward direction, *vecProd* slices a vector into two pieces, while *vecComp* slices a vector into a two-dimensional array.[19,20]

3.8 Finite isomorphisms

For each type a with finitely many distinct values n, we can form an isomorphism between a and $\mathit{Finite}\ n$ (representing the natural numbers $\{0, \ldots, n-1\}$ and introduced briefly in Section 2). We will then use the latter as safe array indices:[21]

$$\textbf{type}\ \mathit{KnownCard}\ a = \mathit{KnownNat}\ |a|$$

$$\begin{aligned} &\textbf{class}\ \mathit{KnownCard}\ a \Rightarrow \mathit{HasFin}\ a\ \textbf{where} \\ &\quad \textbf{type}\ |a| :: \mathit{Nat} \\ &\quad \mathit{fin} :: a \cong \mathit{Finite}\ |a| \end{aligned}$$

The *KnownNat* constraint is part of GHC's support for type-level natural numbers [ref]. Figure 7 shows *HasFin* instances for some standard types and type constructions.[22] These instances come from a simple specification:

Lemma 7. Each *fin* defined in Figure 7 is an order isomorphism.

Proof. Follows from Lemmas 4 and 5. □

It will sometimes be convenient to extract the halves of the *fin* isomorphism:

$$\begin{aligned} &\mathit{toFin} :: \mathit{HasFin}\ a \Rightarrow a \to \mathit{Finite}\ |a| \\ &\mathit{toFin} = \mathit{isoFwd}\ \mathit{fin} \end{aligned}$$

$$\begin{aligned} &\mathit{unFin} :: \mathit{HasFin}\ a \Rightarrow \mathit{Finite}\ |a| \to a \\ &\mathit{unFin} = \mathit{isoRev}\ \mathit{fin} \end{aligned}$$

4 Working with flattened functors

Section 2 defined a type $\mathit{Flat}\ f\ a$ and showed it to be isomorphic to $f\ a$ for any representable functor f having a finite associated index type. The vocabulary defined above suffices to define this isomorphism concretely:

$$\begin{aligned} &\mathit{flat} :: \mathit{HasFlat}\ f \Rightarrow f \mathrel{\dot{\cong}} \mathit{Flat}\ f \\ &\mathit{flat} = \mathit{inv}\ \mathit{newIso} \circ \mathit{inv}\ \mathit{repIso} \circ \mathit{dom}\ (\mathit{inv}\ \mathit{fin}) \circ \mathit{repIso} \end{aligned}$$

This definition mirrors the type isomorphism chain appearing in Section 2.

To make for simpler calculations, let's now slightly refactor the definition of *Flat*. Let $\mathit{Arr}\ a\ b$ be a type of "domain-indexed safe arrays"[23], indexed by a with elements in b:

$$\textbf{newtype}\ \mathit{Arr}\ a\ b = \mathit{Arr}\ (\mathit{Vector}\ |a|\ b)$$

We can easily redefine *Flat* via *Arr*:

[19] [Add another isomorphism for exponentiation.]
[20] [Use these vector-reshaping definitions as *specifications*, but at least allude an efficient, no-copy implementation.]
[21] [I think I use the terms "vector" and "array" interchangeably. Perhaps pick one and stick to it.]
[22] [Add a $\mathit{HasFin}\ (a \to b)$ instance.]
[23] [Look for a different description and matching name.]

$$\begin{array}{l}
\mathbf{type}\ \mathit{KnownCard}\ a = \mathit{KnownNat}\ |a| \\[1ex]
\mathbf{class}\ \mathit{KnownCard}\ a \Rightarrow \mathit{HasFin}\ a\ \mathbf{where} \\
\quad \mathbf{type}\ |a| :: \mathit{Nat} \\
\quad \mathit{fin} :: a \cong \mathit{Finite}\ |a| \\[1ex]
\mathbf{instance}\ \mathit{HasFin}\ \mathbf{0}\ \mathbf{where} \\
\quad \mathbf{type}\ |\mathbf{0}| = 0 \\
\quad \mathit{fin} = \mathit{finU}_1 \\[1ex]
\mathbf{instance}\ \mathit{HasFin}\ \mathbf{1}\ \mathbf{where} \\
\quad \mathbf{type}\ |\mathbf{1}| = 1 \\
\quad \mathit{fin} = \mathit{finPar}_1 \\[1ex]
\mathbf{instance}\ \mathit{KnownNat}\ n \Rightarrow \mathit{HasFin}\ (\mathit{Finite}\ n)\ \mathbf{where} \\
\quad \mathbf{type}\ |\mathit{Finite}\ n| = n \\
\quad \mathit{fin} = \mathit{id} \\[1ex]
\mathbf{instance}\ (\mathit{HasFin}\ a, \mathit{HasFin}\ b) \Rightarrow \mathit{HasFin}\ (a + b)\ \mathbf{where} \\
\quad \mathbf{type}\ |a + b| = |a| + |b| \\
\quad \mathit{fin} = \mathit{finSum} \circ (\mathit{fin} + \mathit{fin}) \\[1ex]
\mathbf{instance}\ (\mathit{HasFin}\ a, \mathit{HasFin}\ b) \Rightarrow \mathit{HasFin}\ (a \times b)\ \mathbf{where} \\
\quad \mathbf{type}\ |a \times b| = |a| \times |b| \\
\quad \mathit{fin} = \mathit{finProd} \circ (\mathit{fin} \times \mathit{fin})
\end{array}$$

Figure 7: *HasFin* instances

$$\mathbf{type}\ \mathit{Flat}\ f = \mathit{Arr}\ (\mathit{Rep}\ f)$$

The type $\mathit{Arr}\ a\ b$ is isomorphic to $a \to b$:

$$\begin{array}{l}
\mathit{arrFun} :: \mathit{HasFin}\ a \Rightarrow \mathit{Arr}\ a\ b \cong (a \to b) \\
\mathit{arrFun} = \mathit{dom}\ \mathit{fin} \circ \mathit{repIso} \circ \mathit{newIso}
\end{array}$$

Use this isomorphism to define a *Representable* instance such that $\mathit{repIso} = \mathit{arrFun}$:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{HasFin}\ a \Rightarrow \mathit{Representable}\ (\mathit{Arr}\ a)\ \mathbf{where} \\
\quad \mathbf{type}\ \mathit{Rep}\ (\mathit{Arr}\ a) = a \\
\quad \mathit{index} = \mathit{isoFwd}\ \mathit{arrFun} \\
\quad \mathit{tabulate} = \mathit{isoRev}\ \mathit{arrFun}
\end{array}$$

Consequently, for $\mathit{Arr}\ a$,

$$\begin{array}{l}
\mathit{index} :: \mathit{HasFin}\ a \Rightarrow \mathit{Arr}\ a\ b \to (a \to b) \\
\mathit{index} = \mathit{isoFwd}\ \mathit{arrFun} \\
\qquad\ \ = \mathit{dom}\ \mathit{toFin} \circ \mathit{index} \circ \mathit{unpack} \\
\qquad\ \ = \lambda(\mathit{Arr}\ \mathit{xs}) \to \mathit{index}\ \mathit{xs} \circ \mathit{toFin} \\[1ex]
\mathit{tabulate} :: \mathit{HasFin}\ a \Rightarrow (a \to b) \to \mathit{Arr}\ a\ b \\
\mathit{tabulate} = \mathit{isoRev}\ \mathit{arrFun} \\
\qquad\qquad = \mathit{pack} \circ \mathit{tabulate} \circ \mathit{dom}\ \mathit{unFin} \\
\qquad\qquad = \lambda f \to \mathit{Arr}\ (\mathit{tabulate}\ (f \circ \mathit{unFin}))
\end{array}$$

We can adapt the vector-reshaping isomorphisms from Section 3.7 to reshape *Arr* instead, as shown in Figure 8.

Lemma 8 (Proved in Appendix A.4)**.** Each of the definitions in Figure 8 is equal to (a type instance of) *reindex id*.

$$
\begin{aligned}
&\textbf{type}\ \mathit{KnownCard}_2\ a\ b = (\mathit{KnownCard}\ a, \mathit{KnownCard}\ b)\\[1ex]
&\mathit{arrU}_1 :: \mathit{Arr}\ \mathbf{0} \mathrel{\dot{\cong}} U_1\\
&\mathit{arrU}_1 = \mathit{vecU}_1 \circ \mathit{newIso}\\[1ex]
&\mathit{arrPar}_1 :: \mathit{Arr}\ \mathbf{1} \mathrel{\dot{\cong}} \mathit{Par}_1\\
&\mathit{arrPar}_1 = \mathit{vecPar}_1 \circ \mathit{newIso}\\[1ex]
&\mathit{arrProd} :: \mathit{KnownCard}_2\ a\ b \Rightarrow \mathit{Arr}\ (a+b) \mathrel{\dot{\cong}} \mathit{Arr}\ a \times \mathit{Arr}\ b\\
&\mathit{arrProd} = \mathit{coerceIso} \circ \mathit{vecProd} \circ \mathit{newIso}\\[1ex]
&\mathit{arrComp} :: \mathit{KnownCard}_2\ a\ b \Rightarrow \mathit{Arr}\ (a\times b) \mathrel{\dot{\cong}} \mathit{Arr}\ a \circ \mathit{Arr}\ b\\
&\mathit{arrComp} = \mathit{coerceIso} \circ \mathit{vecComp} \circ \mathit{newIso}
\end{aligned}
$$

Figure 8: *Arr* reshaping isomorphisms

Note that the only operations besides the corresponding vector reshapers are *newIso* and *coerceIso*, which will likely vanish at compile time.[24,25,26] The *arrFun* isomorphism does most of the work for *flat*:[27]

$$
\begin{aligned}
&\mathit{flat} :: \mathit{HasFlat}\ f \Rightarrow f\ a \cong \mathit{Flat}\ f\ a\\
&\mathit{flat} = \mathit{reindex}\ \mathit{id}\\
&\quad\ = \mathit{inv}\ \mathit{repIso} \circ \mathit{repIso}\\[1ex]
&\mathit{toFlat} :: \mathit{HasFlat}\ f \Rightarrow f\ a \to \mathit{Flat}\ f\ a\\
&\mathit{toFlat} = \mathit{isoFwd}\ \mathit{flat}\\
&\qquad\ = \mathit{tabulate} \circ \mathit{index}
\end{aligned}
$$

These last two definitions look like they would simplify to *id*, but they do not, because the argument and result types differ. This pattern occurs whenever converting between functors via a common index type (here $\mathit{Rep}\ f$).[28]

Now, note that homomorphisms compose (into homomorphisms), so to guarantee that *toFlat* is homomorphic, it suffices to guarantee that *tabulate* or *index* on f is homomorphic. We will *assume* the latter as a reasonable expectation on the representable functors involved.[29]

Theorem 9 (Proved in Appendix A.5)**.** Given the instance definitions below, *tabulate* for $\mathit{Arr}\ a$ is a homomorphism with respect to *Functor* and *Applicative*:[30]

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (\mathit{Arr}\ a)\ \textbf{where}\\
&\quad \mathit{fmap}\ h\ (\mathit{Arr}\ \mathit{bs}) = \mathit{Arr}\ (\mathit{fmap}\ h\ \mathit{bs})\\[1ex]
&\textbf{instance}\ \mathit{KnownCard}\ a \Rightarrow \mathit{Applicative}\ (\mathit{Arr}\ a)\ \textbf{where}\\
&\quad \mathit{pure}\ a = \mathit{Arr}\ (\mathit{pure}\ a)\\
&\quad \mathit{Arr}\ \mathit{fs} \circledast \mathit{Arr}\ \mathit{xs} = \mathit{Arr}\ (\mathit{fs} \circledast \mathit{xs})
\end{aligned}
$$

In Haskell, these two instance definitions can be written more succinctly:

$$
\begin{aligned}
&\textbf{deriving instance}\ \mathit{Functor}\ (\mathit{Arr}\ a)\\
&\textbf{deriving instance}\ \mathit{KnownCard}\ a \Rightarrow \mathit{Applicative}\ (\mathit{Arr}\ a)
\end{aligned}
$$

[24] [Do they?]
[25] [Move this paragraph and figure somewhere more sensible.]
[26] [I think each of these *Arr* reshapers is a special case of *reindex id*. Prove and exploit.]
[27] [Revisit this part, since *arrFun* is no longer apparent here.]
[28] Give a name to *reindex id*, and use it here.
[29] [Return to this assumption.]
[30] [Hence *index* is as well. Explain somewhere clearly and simply that inverses of homomorphisms and compositions of homomorphisms are also homomorphisms.]

$$
\begin{array}{l}
\textbf{instance } \mathit{Foldable}\ ((\to)\ \mathbf{0})\ \textbf{where} \\
\quad \mathit{fold}\ \_ = \emptyset \\[1ex]
\textbf{instance } \mathit{Foldable}\ ((\to)\ \mathbf{1})\ \textbf{where} \\
\quad \mathit{fold}\ \mathit{as} = \mathit{as}\ () \\[1ex]
\textbf{instance } (\mathit{Foldable}\ ((\to)\ a), \mathit{Foldable}\ ((\to)\ b)) \Rightarrow \mathit{Foldable}\ ((\to)\ (a+b))\ \textbf{where} \\
\quad \mathit{fold}\ \mathit{as} = \mathit{fold}\ (\mathit{as} \circ \mathit{Left}) \oplus \mathit{fold}\ (\mathit{as} \circ \mathit{Right}) \\[1ex]
\textbf{instance } (\mathit{Foldable}\ ((\to)\ a), \mathit{Foldable}\ ((\to)\ b)) \Rightarrow \mathit{Foldable}\ ((\to)\ (a \times b))\ \textbf{where} \\
\quad \mathit{fold}\ \mathit{as} = \mathit{fold}\ (\mathit{fold} \circ \mathit{curry}\ \mathit{as}) \\[1ex]
\textbf{instance } \mathit{KnownNat}\ n \Rightarrow \mathit{Foldable}\ ((\to)\ (\mathit{Finite}\ n))\ \textbf{where} \\
\quad \ldots
\end{array}
$$

Figure 9: Folding functions

What's remarkable about these definitions is that the conversions between the index type a and its numeric counterpart $\mathit{Finite}\ |a|$ (via the fin isomorphism) have disappeared during calculation. The index type a therefore plays no role in the structure of the algorithms used for fmap, pure, and $(\circledast)$ on $\mathit{Arr}\ a$. As the instances above show, these operations are implemented directly as the corresponding operations on $\mathit{Vector}\ n$.[31] Those vector operations correspond to what SIMD-style parallel processors—including GPUs—do best, namely map k-ary functions over k vectors of arguments. We will see that the structure of the algorithms used for other operations *does* depend on the functor f. By fixing the *Functor* and *Applicative* implementations always to use the full SIMD style, we exploit the high-performance parallelism of the GPU architecture. By varying the implementations of other classes (*Foldable* etc) according to choice of functor f, we embrace a variety of parallel algorithms for solving the same problem with different sequential-vs-parallel trade-offs.[32]

5 Folds

While the *Functor* and *Applicative* operations correspond to fully parallel computations, folds do not. Let's now examine how to structure them by a combination of sequential and parallel composition. An important requirement to keep in mind is that GPU-style architectures support "flat" data parallelism, i.e., sequential compositions of fully parallel passes [ref].

5.1 Folding functions

Unlike fmap, pure, and $(\circledast)$ (from *Functor* and *Applicative*), we will not simply delegate fold on $\mathit{Arr}\ a$ (or $\mathit{Flat}\ f$) to the same operation on vectors. Instead, we will imitate folds on *functions*. The standard Haskell libraries do not define these instances, but they could, as shown in Figure 9.[33,34] These instances reflect the view of a function $f :: a \to b$ as an a-indexed collection of b values. To avoid ambiguity, the index type a must linearly ordered, and the fold definitions must respect that ordering. In particular, for $a + b$, $\mathit{Left}\ a < \mathit{Right}\ b$ for all a and b, and for $a \times b$, ordering is lexicographic, as in Section 3.4.

Another way to justify these instances is to relate them to *Foldable* instances on the representable functors isomorphic to these functions, as shown in Figure 10.[35] The reverse might be more satisfying, however,

[31] The instances also involve removing and adding the **newtype** wrapper (Arr), but even those simple operations disappear during compilation.

[32] [I think there is a lovely principle here to be highlighted. Datatype-indexed families of generic parallel algorithms are great [say why], but they don't operate on arrays and so are difficult to map well to flat data parallelism. The *Arr* (or *Flat*) data type retains the type-driven nature of the families of generic algorithms, while mapping well to flat data parallelism. Revisit Guy Blelloch's flattening transformation to determine how my techniques relate to it.]

[33] [Explain Haskellisms, especially $(\to)\ a$.]

[34] [Start using "*Fun*" in place of $(\to)$ when given only one argument.]

[35] [State and prove a theorem here. Maybe specify by enumerating the inhabitants of each functor in index order into a list and then folding over the list.]

$$
\begin{array}{l}
\textbf{instance } \mathit{Foldable}\ U_1 \textbf{ where} \\
\quad \mathit{fold}\ U_1 = \emptyset \\[1ex]
\textbf{instance } \mathit{Foldable}\ \mathit{Par}_1 \textbf{ where} \\
\quad \mathit{fold}\ (\mathit{Par}_1\ a) = a \\[1ex]
\textbf{instance } (\mathit{Foldable}\ f, \mathit{Foldable}\ g) \Rightarrow \mathit{Foldable}\ (f \times g) \textbf{ where} \\
\quad \mathit{fold}\ (\mathit{fa} \times \mathit{ga}) = \mathit{fold}\ \mathit{fa} \oplus \mathit{fold}\ \mathit{ga} \\[1ex]
\textbf{instance } (\mathit{Foldable}\ f, \mathit{Foldable}\ g) \Rightarrow \mathit{Foldable}\ (g \circ f) \textbf{ where} \\
\quad \mathit{fold}\ (\mathit{Comp}_1\ \mathit{gfa}) = \mathit{fold}\ (\mathit{fmap}\ \mathit{fold}\ \mathit{gfa}) \\[1ex]
\textbf{instance } \mathit{KnownNat}\ n \Rightarrow \mathit{Foldable}\ (\mathit{Vector}\ n) \textbf{ where} \\
\quad \ldots
\end{array}
$$

Figure 10: Folding representable functors

calculating *Foldable* instances for representable functors by appealing to the instances on functions in Figure 9.[36]

Note that in Haskell's current *Foldable* class, a *fold* definition does not suffice for a complete instance definition, so the *fold* definitions in Figure 9 would have to be replaced or augmented by *foldMap* definitions. Semantically, $\mathit{foldMap}\ f = \mathit{fold} \circ \mathit{fmap}\ f$, and that equality could be captured as a default definition for *foldMap*, though currently it is not. This current situation is unfortunate for parallel SIMD performance. As mentioned above, $\mathit{fmap}\ f$ is a pure SIMD operation and so every f application in $\mathit{fmap}\ f$ can be evaluated in a single parallel pass (assuming sufficient hardware resoures), to then be followed by a sequential composition of passes for the remaining *fold*. Taking $\mathit{foldMap}\ f$ as primitive moves those f applications *into* the fold, where they are no longer all evaluated in parallel.[37]

We will sometimes want to re-parametrize functions monotonically for convenient and efficient folding:

Lemma 10 (Proved in Appendix A.6)**.** For types a and b and any *order isomorphism* $h :: a \to b$, $\mathit{dom}\ h$ is a *Foldable* isomorphism, i.e., have $\mathit{fold} = \mathit{fold} \circ \mathit{dom}\ h$. (The LHS *fold* is on $a \to x$, while the RHS *fold* is on $b \to x$.)

We can generalize Lemma 10 to reindexing of representable functors:

Lemma 11 (Proved in Appendix A.7)**.** For functors f and g and any *order isomorphism* $h :: \mathit{Rep}\ g \cong \mathit{Rep}\ f$,

$$\mathit{fold} \circ \mathit{index} = \mathit{fold} \circ \mathit{index} \circ \mathit{isoFwd}\ (\mathit{reindex}\ h :: f \mathrel{\dot{\cong}} g)$$

5.2 Folding flattened functors

[Maybe move function folds to an earlier section, so we can move the $\mathit{Foldable}\ (\mathit{Arr}\ a)$ instances earlier.]

Theorem 9 in Section 4 gives simple instances of *Functor* and *Applicative* for $\mathit{Arr}\ a$, calculated from the usual specification that *tabulate* (or *index*) on $\mathit{Arr}\ a$ is a homomorphism:

Theorem 12 (Proved in Appendix A.8)**.** Given the following instance definition, *index* for $\mathit{Arr}\ a$ is a *Foldable* homomorphism:

$$
\begin{array}{l}
\textbf{instance } \mathit{Foldable}\ (\mathit{Arr}\ a) \textbf{ where} \\
\quad \mathit{fold} = \mathit{fold} \circ \mathit{unpack}
\end{array}
$$

Equivalently,

$$\textbf{deriving instance } \mathit{Foldable}\ (\mathit{Arr}\ a)$$

[36] [Maybe do some, and leave the rest for exercises.]
[37] [Return to this point later.]

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Foldable}\ (\mathit{Arr}\ \mathbf{0})\ \mathbf{where} \\
\quad \mathit{fold} = \mathit{fold} \circ \mathit{isoFwd}\ \mathit{arrU}_1 \\[1ex]
\mathbf{instance}\ \mathit{Foldable}\ (\mathit{Arr}\ \mathbf{1})\ \mathbf{where} \\
\quad \mathit{fold} = \mathit{fold} \circ \mathit{isoFwd}\ \mathit{arrPar}_1 \\[1ex]
\mathbf{instance}\ (\mathit{Foldable}\ (\mathit{Arr}\ a), \mathit{Foldable}\ (\mathit{Arr}\ b), \mathit{KnownCard}_2\ a\ b) \Rightarrow \mathit{Foldable}\ (\mathit{Arr}\ (a + b))\ \mathbf{where} \\
\quad \mathit{fold} = \mathit{fold} \circ \mathit{isoFwd}\ \mathit{arrProd} \\[1ex]
\mathbf{instance}\ (\mathit{Foldable}\ (\mathit{Arr}\ a), \mathit{Foldable}\ (\mathit{Arr}\ b), \mathit{KnownCard}_2\ a\ b) \Rightarrow \mathit{Foldable}\ (\mathit{Arr}\ (a \times b))\ \mathbf{where} \\
\quad \mathit{fold} = \mathit{fold} \circ \mathit{isoFwd}\ \mathit{arrComp}
\end{array}
$$

Figure 11: Specialized and optimized *Foldable* instances for *Arr*

As with *Functor* and *Applicative*, this *Foldable* instance for $\mathit{Arr}\ a$ simply defers to the corresponding instance for $\mathit{Vector}\ |a|$. Although this instance is correct, it is somewhat dissatisfying. If we defer all instances for $\mathit{Arr}\ a$ to the corresponding instances for $\mathit{Vector}\ |a|$, then we will have achieved our goal of safety, but not of type-directed algorithm design.[38] For *Functor* and *Applicative*, the *Vector* instances are compelling in that they directly exploit SIMD-style architecture. Since *fold* is not a SIMD operation in itself (due to data dependencies), it must be decomposed into a pattern of SIMD computations with at least some sequentiality. An entirely sequential *fold* would have a much longer parallel computation time than necessary. Instead, we can use the specific nature of a (not just its cardinality) into account.[39]

Theorem 13 (Proved in Appendix A.9)**.** The specialized instance definitions in Figure 11 below agree with the general instance above. With these specialized definitions, therefore, *index* for $\mathit{Arr}\ a$ is a *Foldable* homomorphism.[40,41]

[Return to *fold* vs $\mathit{foldMap}\ f$. Maybe I should start with $\mathit{foldMap}\ f$ and then explain (or better, *show*) how $\mathit{fold} \circ \mathit{fmap}\ f$ gives better parallelism.]

6 What else?

[

- Idea: use $\mathit{foldMap}\ f$ at first, and then discover the loss of parallelism, motivating factoring $\mathit{foldMap}\ f = \mathit{fold} \circ \mathit{fmap}\ f$.
- *LScan* and *FFT*. There might not be much direct value in this work for just *Functor*, *Applicative*, and *Foldable*, since vector-specific versions don't specialize for the first two and don't might not be worth varying for the latter. I think I'll have to tackle a whole new issue, which is how to *construct* $\mathit{Arr}\ a$ in a compositional and functional manner but still map to efficient data-parallel code. I think the underlying implementation will imperatively update output arrays. I guess I'll have to manage non-interference proofs for parallel computations, hopefully in an elegantly rigorous way.
- In-place update.
- Add an isomorphism version of *fold*/*unfold*, given an algebra/coalgebra isomorphism. Are there interesting and/or useful examples of invertible algebras?

]

[38] [Connect these remarks to an earlier discussion of goals.]

[39] [Maybe *fold* is a poor use of this flexibility. It may be perfectly fine to have a single *fold* strategy based *only* on cardinality. On the other hand, *scan* and *fft* make good use of additional flexibility.]

[40] [Resolve *fold* vs $\mathit{foldMap}\ f$.]

[41] [Maybe refactor: add a class with a method that chooses an order isomorphism for reindexing. Then I could give a single instance for $\mathit{Foldable}\ (\mathit{Arr}\ a)$ that defers to the new class and method.]

7 Related work

- The flattening transformation for nested data parallelism.
- Reversible computing?
- [Gibbons, 2017]
- [Hinze and James, 2010]

A Proofs

A.1 Theorem 1

First, let's require that *iso* is a *functor*, i.e., a homomorphism for the *Category* interface shown in Figure 1. The corresponding homomorphism properties for *iso*:

$$id = iso\ id$$

$$iso\ g \circ iso\ f = iso\ (g \circ f)$$

Start with the *id* homomorphism, and simplify the RHS:

$$\begin{array}{ll} & iso\ id \\ = & \{\ \text{definition of } iso\ \} \\ & id \rightleftharpoons id^{-1} \\ = & \{\ id \text{ is its own inverse (i.e., } id \circ id = id)\ \} \\ & id \rightleftharpoons id \end{array}$$

The *id* homomorphism for *iso* is thus equivalent to

$$id = id \rightleftharpoons id$$

We can thus satisfy the homomorphism requirement by using this version of as a *definition*.

Next consider the $(\circ)$ homomorphism and simplify the LHS:

$$\begin{array}{ll} & iso\ g \circ iso\ f \\ = & \{\ \text{definition of } iso\ \} \\ & (g \rightleftharpoons g^{-1}) \circ (f \rightleftharpoons f^{-1}) \end{array}$$

Then the RHS:

$$\begin{array}{ll} & iso\ (g \circ f) \\ = & \{\ \text{definition of } iso\ \} \\ & (g \circ f) \rightleftharpoons (g \circ f)^{-1} \\ = & \{\ \text{property of inversion and composition}\ \} \\ & (g \circ f) \rightleftharpoons (f^{-1} \circ g^{-1}) \end{array}$$

The $(\circ)$ homomorphism is thus equivalent to

$$((g \rightleftharpoons g^{-1}) \circ (f \rightleftharpoons f^{-1})) = (g \circ f \rightleftharpoons f^{-1} \circ g^{-1})$$

Now *strengthen* this requirement by generalizing from f^{-1} and g^{-1} to arbitrary f' and g' (having the required types):

$$((g \rightleftharpoons g') \circ (f \rightleftharpoons f')) = (g \circ f \rightleftharpoons f' \circ g')$$

This strengthened (hence sufficient) condition is also in solved form and so can be satisfied by definition.

We can set up and solve similar homomorphism equations for the operations of the other categorical classes, leading to the class instances in Figure 1. For instance, for *MonoidalP*, the crucial insight is as follows:

Lemma 14. The product, coproduct, and exponential bifunctors invert as follows:

$$
\begin{aligned}
(f \times g)^{-1} &= f^{-1} \times g^{-1} \\
(f + g)^{-1} &= f^{-1} + g^{-1} \\
(f \Rightarrow g)^{-1} &= f^{-1} \Rightarrow g^{-1}
\end{aligned}
$$

Proof:

$$
\begin{aligned}
& (f^{-1} \times g^{-1}) \circ (f \times g) \\
=\; & \{\; (f \times g) \circ (h \times k) = (f \circ h) \times (g \circ k) \text{ [Gibbons, 2002, Section 1.5.1] } \} \\
& f^{-1} \circ f \times g^{-1} \circ g \\
=\; & \{ \text{ fundamental property of inverses } \} \\
& id \times id \\
=\; & \{ \text{ [Gibbons, 2002, Section 1.5.1] } \} \\
& id
\end{aligned}
$$

Likewise $(f \times g) \circ (f^{-1} \times g^{-1}) = id$. Similarly for $f + g$, while $f \Rightarrow g$ differs slightly due to contravariance:

$$
\begin{aligned}
& (f^{-1} \Rightarrow g^{-1}) \circ (f \Rightarrow g) \\
=\; & \{\; (f \Rightarrow g) \circ (h \Rightarrow k) = (h \circ f) \Rightarrow (g \circ k) \;\} \\
& f \circ f^{-1} \Rightarrow g^{-1} \circ g \\
=\; & \{ \text{ fundamental property of inverses } \} \\
& id \Rightarrow id \\
=\; & \{ \text{ [cite or prove] } \} \\
& id
\end{aligned}
$$

□

A.2 Lemma 2

$$
\begin{aligned}
& unfork \circ fork \\
=\; & \{ \; \eta\text{-expansion } \} \\
& \lambda(f, g) \to unfork\ (fork\ (f, g)) \\
=\; & \{ \text{ definition of } fork \; \} \\
& \lambda(f, g) \to unfork\ (f \vartriangle g) \\
=\; & \{ \text{ definition of } unfork \; \} \\
& \lambda(f, g) \to (exl \circ (f \vartriangle g), exr \circ (f \vartriangle g)) \\
=\; & \{ \text{ [Gibbons, 2002, Section 1.5.1] } \} \\
& \lambda(f, g) \to (f, g) \\
=\; & \{ \text{ definition of } id \text{ for functions } \} \\
& id
\end{aligned}
$$

$$
\begin{aligned}
& unjoin \circ join \\
=\; & \{ \; \eta\text{-expansion } \} \\
& \lambda(f, g) \to unjoin\ (join\ (f, g)) \\
=\; & \{ \text{ definition of } join \; \} \\
& \lambda(f, g) \to unjoin\ (f \triangledown g) \\
=\; & \{ \text{ definition of } unjoin \; \} \\
& \lambda(f, g) \to ((f \triangledown g) \circ inl, (f \triangledown g) \circ inr) \\
=\; & \{ \text{ [Gibbons, 2002, Section 1.5.2] } \} \\
& \lambda(f, g) \to (f, g) \\
=\; & \{ \text{ definition of } id \text{ for functions } \} \\
& id
\end{aligned}
$$

$$\begin{array}{ll}
& \mathit{fork} \circ \mathit{unfork} \\
= & \{\ \eta\text{-expansion}\ \} \\
& \lambda f \rightarrow \mathit{fork}\ (\mathit{unfork}\ f) \\
= & \{\ \text{definition of } \mathit{unfork}\ \} \\
& \lambda f \rightarrow \mathit{fork}\ (\mathit{exl} \circ f, \mathit{exr} \circ f) \\
= & \{\ \text{definition of } \mathit{fork}\ \} \\
& \lambda f \rightarrow (\mathit{exl} \circ f) \vartriangle (\mathit{exr} \circ f) \\
= & \{\ \text{[Gibbons, 2002, Section 1.5.1]}\ \} \\
& \lambda f \rightarrow (\mathit{exl} \vartriangle \mathit{exr}) \circ f \\
= & \{\ \text{[Gibbons, 2002, Section 1.5.1]}\ \} \\
& \lambda f \rightarrow \mathit{id} \circ f \\
= & \{\ \text{property of } \mathit{id} \text{ and } (\circ)\ \} \\
& \lambda f \rightarrow f \\
= & \{\ \text{definition of } \mathit{id} \text{ for functions}\ \} \\
& \mathit{id}
\end{array}$$

$$\begin{array}{ll}
& \mathit{join} \circ \mathit{unjoin} \\
= & \{\ \eta\text{-expansion}\ \} \\
& \lambda f \rightarrow \mathit{join}\ (\mathit{unjoin}\ f) \\
= & \{\ \text{definition of } \mathit{unjoin}\ \} \\
& \lambda f \rightarrow \mathit{join}\ (f \circ \mathit{inl}, f \circ \mathit{inr}) \\
= & \{\ \text{definition of } \mathit{join}\ \} \\
& \lambda f \rightarrow (f \circ \mathit{inl}) \triangledown (f \circ \mathit{inr}) \\
= & \{\ \text{[Gibbons, 2002, Section 1.5.2]}\ \} \\
& \lambda f \rightarrow f \circ (\mathit{inl} \triangledown \mathit{inr}) \\
= & \{\ \text{[Gibbons, 2002, Section 1.5.2]}\ \} \\
& \lambda f \rightarrow f \circ \mathit{id} \\
= & \{\ \text{property of } \mathit{id} \text{ and } (\circ)\ \} \\
& \lambda f \rightarrow f \\
= & \{\ \text{definition of } \mathit{id} \text{ for functions}\ \} \\
& \mathit{id}
\end{array}$$

A.3 Lemma 5

[To do. See my notes from 2018-08-01.]

A.4 Lemma 8

[See my notes from 2018-08-11. I'm looking for much simpler proofs.]

A.5 Theorem 9

[Add some explanation here.]

$$\begin{array}{ll}
& \mathit{tabulate} \circ \mathit{fmap}\ f \circ \mathit{index} \\
= & \{\ \text{definition of } \mathit{tabulate} \text{ and } \mathit{index} \text{ for } \mathit{Arr}\ a\ \} \\
& \mathit{pack} \circ \mathit{tabulate} \circ \mathit{dom}\ \mathit{unFin} \circ \mathit{fmap}\ f \circ \mathit{dom}\ \mathit{toFin} \circ \mathit{index} \circ \mathit{unpack} \\
= & \{\ \mathit{tabulate} \circ \mathit{dom}\ \mathit{unFin} \text{ is a } \mathit{Functor} \text{ homomorphism}\ \} \\
& \mathit{pack} \circ \mathit{fmap}\ f \circ \mathit{unpack}
\end{array}$$

$$\begin{array}{ll}
& \mathit{tabulate}\ (\mathit{pure}\ a) \\
= & \{\ \text{definition of } \mathit{tabulate} \text{ for } \mathit{Arr}\ a\ \} \\
& \mathit{pack}\ (\mathit{tabulate}\ (\mathit{dom}\ \mathit{unFin}\ (\mathit{pure}\ a))) \\
= & \{\ \mathit{tabulate} \circ \mathit{dom}\ \mathit{unFin} \text{ is an } \mathit{Applicative} \text{ homomorphism}\ \} \\
& \mathit{pack}\ (\mathit{pure}\ a)
\end{array}$$

$$\begin{array}{ll}
& \mathit{tabulate}\ (\mathit{index}\ \mathit{fs} \circledast \mathit{index}\ \mathit{xs}) \\
= & \{\ \text{definition of } \mathit{tabulate} \text{ and } \mathit{index} \text{ for } \mathit{Arr}\ a\ \} \\
& \mathit{pack}\ (\mathit{tabulate}\ (\mathit{dom}\ \mathit{unFin}\ (\mathit{dom}\ \mathit{toFin}\ (\mathit{index}\ (\mathit{unpack}\ \mathit{fs})) \circledast \mathit{dom}\ \mathit{toFin}\ (\mathit{index}\ (\mathit{unpack}\ \mathit{xs}))))) \\
= & \{\ \mathit{tabulate} \circ \mathit{dom}\ \mathit{unFin} \text{ is an } \mathit{Applicative} \text{ homomorphism}\ \} \\
& \mathit{pack}\ (\mathit{unpack}\ \mathit{fs} \circledast \mathit{unpack}\ \mathit{xs})
\end{array}$$

A.6 Lemma 10

Proof.

$$\begin{array}{ll}
& \mathit{fold} \circ \mathit{dom}\ h \\
= & \{\ \eta\text{-expansion}\ \} \\
& \lambda \mathit{xs} \rightarrow \mathit{fold}\ (\mathit{dom}\ h\ \mathit{xs}) \\
= & \{\ \text{definition of } \mathit{dom} \text{ on functions}\ \} \\
& \lambda \mathit{xs} \rightarrow \mathit{fold}\ (\mathit{xs} \circ h)
\end{array}$$

[How to finish this proof? Maybe via a new lemma in Section 5.1. Still, I think I'll need a new angle in order to make general claims about folds on functions.] □

Lemma 15. For any function $f :: u \to v$, $\mathit{dom}\ f$ is a *Functor* and *Applicative* homomorphism.

Proof.

$$
\begin{aligned}
& \mathit{dom}\ f\ (\mathit{fmap}\ h\ \mathit{xs}) \\
=\ & \{\ \text{definition of } \mathit{fmap} \text{ on functions}\ \} \\
& \mathit{dom}\ f\ (h \circ \mathit{xs}) \\
=\ & \{\ \text{definition of } \mathit{dom}\ \} \\
& (h \circ \mathit{xs}) \circ f \\
=\ & \{\ \text{associativity of } (\circ)\ \} \\
& h \circ (\mathit{xs} \circ f) \\
=\ & \{\ \text{definition of } \mathit{dom}\ \} \\
& h \circ \mathit{dom}\ f\ \mathit{xs} \\
=\ & \{\ \text{definition of } \mathit{fmap} \text{ on functions}\ \} \\
& \mathit{fmap}\ h\ (\mathit{dom}\ f\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{dom}\ f\ (\mathit{pure}\ a) \\
=\ & \{\ \text{definition of } \mathit{pure} \text{ on functions}\ \} \\
& \mathit{dom}\ f\ (\mathit{const}\ a) \\
=\ & \{\ \text{definition of } \mathit{dom}\ \} \\
& \mathit{const}\ a \circ f \\
=\ & \{\ \text{property of } \mathit{const} \text{ and } (\circ)\ \} \\
& \mathit{const}\ a \\
=\ & \{\ \text{definition of } \mathit{pure} \text{ on functions}\ \} \\
& \mathit{pure}\ a
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{dom}\ f\ (\mathit{hs} \circledast \mathit{xs}) \\
=\ & \{\ \text{definition of } (\circledast) \text{ on functions}\ \} \\
& \mathit{dom}\ f\ (\lambda u \to \mathit{hs}\ u \circledast \mathit{xs}\ u) \\
=\ & \{\ \text{definition of } \mathit{dom}\ \} \\
& (\lambda u \to \mathit{hs}\ u \circledast \mathit{xs}\ u) \circ f \\
=\ & \{\ \text{definition of } (\circ) \text{ on functions}\ \} \\
& (\lambda u \to (\mathit{hs}\ (f\ u))\ (\mathit{xs}\ (f\ u))) \\
=\ & \{\ \text{definition of } (\circ) \text{ on functions}\ \} \\
& (\lambda u \to ((\mathit{hs} \circ f)\ u)\ ((\mathit{xs} \circ f)\ u)) \\
=\ & \{\ \text{definition of } (\circledast) \text{ on functions}\ \} \\
& (\mathit{hs} \circ f) \circledast (\mathit{xs} \circ f) \\
=\ & \{\ \text{definition of } \mathit{dom}\ \} \\
& \mathit{dom}\ f\ \mathit{hs} \circledast \mathit{dom}\ f\ \mathit{xs}
\end{aligned}
$$

□

A.7 Lemma 11

Proof.

$$
\begin{aligned}
& \mathit{fold} \circ \mathit{index} \circ \mathit{isoFwd}\ (\mathit{reindex}\ h) \\
=\ & \{\ \text{definition of } \mathit{reindex}\ \} \\
& \mathit{fold} \circ \mathit{index} \circ \mathit{isoFwd}\ (\mathit{inv}\ \mathit{repIso} \circ \mathit{dom}\ h \circ \mathit{repIso}) \\
=\ & \{\ \text{definitions of } (\circ) \text{ on IsoT } k,\ \mathit{inv}, \text{ etc}\ \} \\
& \mathit{fold} \circ \mathit{index} \circ \mathit{tabulate} \circ \mathit{dom}\ (\mathit{isoFwd}\ h) \circ \mathit{index} \\
=\ & \{\ \mathit{index} \circ \mathit{tabulate} = \mathit{id}\ \} \\
& \mathit{fold} \circ \mathit{dom}\ (\mathit{isoFwd}\ h) \circ \mathit{index}
\end{aligned}
$$

$$\begin{aligned}
&= \{ \text{ Lemma 10 } \} \\
&\quad \mathit{fold} \circ \mathit{index}
\end{aligned}$$

□

A.8 Theorem 12

We want to show that $\mathit{fold} \circ \mathit{index} = \mathit{fold} \circ \mathit{unpack}$, where the equation is on $\mathit{Arr}\ a$. (The LHS *fold* is on $(\rightarrow)\ a$ and the RHS *fold* is on $\mathit{Vector}\ |a|$.)

Proof.

$$\begin{aligned}
&\quad \mathit{fold} \circ \mathit{index} \\
&= \{ \text{ definition of } \mathit{index} \text{ for } \mathit{Arr}\ a\ \} \\
&\quad \mathit{fold} \circ \mathit{dom}\ \mathit{toFin} \circ \mathit{index} \circ \mathit{unpack} \\
&= \{\ \mathit{dom}\ \mathit{toFin} \circ \mathit{index} \text{ is a } \mathit{Foldable} \text{ homomorphism } \} \\
&\quad \mathit{fold} \circ \mathit{unpack}
\end{aligned}$$

□

The last step depends on *dom toFin* being a *Foldable* homomorphism, which follows from Theorem 7 and Lemma 11.

A.9 Theorem 13

Proof. Let p be one of the *Arr*-reshaping isomorphisms in Figure 8. By Lemma 8, we know that $p = \mathit{reindex}\ h$ where h is monotonic (and in fact $h = \mathit{id}$).

$$\begin{aligned}
&\quad \mathit{fold} \\
&= \{ \text{ definition of } \mathit{Foldable} \text{ instance (Figure 8) } \} \\
&\quad \mathit{fold} \circ \mathit{isoFwd}\ (\mathit{reindex}\ h) \\
&= \{ \text{ induction on the index type } \} \\
&\quad \mathit{fold} \circ \mathit{index} \circ \mathit{isoFwd}\ (\mathit{reindex}\ h) \\
&= \{ \text{ Lemma 11 } \} \\
&\quad \mathit{fold} \circ \mathit{index}
\end{aligned}$$

□

References

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe Zero-cost Coercions for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, 2014.

Conal Elliott. Generic functional parallel algorithms: Scan and FFT. *Proceedings of the ACM Programming Languages (ICFP)*, 1(ICFP), September 2017.

Jeremy Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

Jeremy Gibbons. APLicative Programming with Naperian Functors. In Hongseok Yang, editor, *European Symposium on Programming*, 2017.

Joe Hermaszewski and Ben Gamari. vector-sized, 2017. URL `http://github.com/expipiplus1/vector-sized`. Haskell library.

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

Ralf Hinze and Daniel W. H. James. Reason isomorphically! In *ICFP-WGP*, 2010.

Darius Jahandarie, Conor McBride, João Cristóvão, and Simon Jakobi. newtype-generics (version 0.5.3), 2018. URL `http://hackage.haskell.org/package/newtype-generics`. Haskell library.

Edward Kmett. adjunctions (version 4.4), 2018. URL `http://hackage.haskell.org/package/adjunctions`. Haskell library.

Roman Leshchinskiy. vector (version 0.12), 2017. URL `https://github.com/haskell/vector`. Haskell library.

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

mniip. finite-typelits, 2017. URL `https://github.com/mniip/finite-typelits`. Haskell library.