DON'T FEAR THE MONOIDS!
A TUTORIAL ON MULTI-SOURCE QUERY OPTIMIZATION

BRIAN BECKMAN
MICROSOFT CORPORATION
VERSION 002
BBECKMAN@MICROSOFT.COM

ABSTRACT. Queries over all kinds of collections — object graphs, XML trees, relational databases — are really aspects of one kind of thing, a **monoid comprehension**. Reasoning about them as such reveals that they can all be optimized by algebraic transformations such as **join decorrelation**, **unnesting**, and **pushing projections around**, even in the face of arbitrary reference aliases, updates, method calls, and side effects during query execution. This may come as a surprise to database practitioners, who are well acquainted with such optimizations, but only in environments where side effects are squelched. But the mathematics behind this is no more complicated than the familiar relational algebra, which is why I say "Don't fear the monoids!"

This tutorial is targeted to programmers and other computing professionals who need to understand query optimization, and thus monoid comprehensions, but who may not have the required mathematical background at their fingertips.

1. INTRODUCTION

One often needs data from multiple sources, in different forms, and with different styles of relationships. For instance, imagine planning a vacation trip, and needing

- in-memory collections of information about cities and countries
- online relational databases of international travel regulations
- XML-mediated web services for hotel, air, and cruise bookings

Now look for adventure-cruise packages in countries with majority Greek ethnicity and seaside hideaway resorts. Without the ability to query over all these sources at once, we might never find that the perfect spot is in Nicosia, Cyprus, and that prices and the advisory climate are currently favorable for booking a vacation there.

The *monoid* is the single idea that reveals just enough similarity amongst these data sources that we can query them together. The monoid is an idea from abstract mathematics, but that should not cause fear — it is not a difficult idea. The important thing is that, since

Date: December 5, 2011.

it is a *mathematical* idea, it is a *precise* idea, and that is what we need for computers. Despite decades of wishful thinking, we still must tell computers what to do in excruciating mathematical precision, at a level often fearsome for humans.

Contents

2. Gentle Introduction to Monoids

If the following definition causes you the slightest discomfort, then read the rest of this section, where we explain it in elementary terms. Otherwise, it is safe to skip this section.

Definition 1. *A **monoid**, $\mathcal{M}\langle \mathcal{T}, \oplus, \mathcal{Z} \rangle$, is a set of elements of type $\mathcal{T}$, closed under an associative binary operation, $\oplus$, with an **identity**, $\mathcal{Z}$.*

If you have gotten here, you want a gentle introduction requiring no background in abstract algebra. So consider just ordinary whole numbers and arithmetic. Step-by-step, we carefully relate all new concepts to them, explaining all generalizations and all symbolic developments. There is no need to fear the words "abstract" and "algebra" — the concepts are few, easy, and worth the effort to learn, because they lead to much greater clarity and economy of thought about the technology of data, and much greater power in programs because of the greater variety of things one can query using essentially the same methods.

The basic, foundational, archetypical example of a monoid is the set of natural numbers, allowing only ordinary addition; *not* allowing multiplication, subtraction, or division. Let the symbol $\mathbb{N}$ stand for the set of natural numbers, that is, for $\{0, 1, 2, \ldots\}$. Observe the following three facts:

- $\mathbb{N}$ is **closed** under the **binary operation** of addition, meaning that for any two elements a and b of $\mathbb{N}$, $a + b$ is also an element of $\mathbb{N}$.
- Addition is **associative**, meaning that for any *three* elements a, b, and c of $\mathbb{N}$, order of operation does not matter: $((a + b) + c) = (a + (b + c))$.
- There is a special **identity** element of $\mathbb{N}$, namely 0, that, when added to any other element, say a, on the right or on the left, does not change its value. In other words, for any a in $\mathbb{N}$, $a + 0 = a$, and $0 + a = a$.

These are the facts to generalize. Any set of things that satisfies an appropriate generalization of these facts is also a monoid and will have just enough structure to allow us to query it conveniently. The appropriate generalization is over three things:

- The **type** of the elements, that is, the set from which elements of the monoid are drawn. The type generalizes $\mathbb{N}$ in the example above.
- The associative binary operation under which the monoid is closed. This generalizes the $+$ operation in the example above.
- The identity element: an element that does not change its partner under the binary operation. This generalizes 0 in the example above.

Be careful to distinguish between the *type* of the elements in the monoid and the elements of the monoid itself. The monoid's operation places restrictions on the type set. In the present case, the type, $\mathbb{N}$, is a more general collection than the monoid. The natural numbers *can* do things that do not fit in the monoid. For example, allowing both addition and

multiplication gives a much richer *kind* of structure, called a *field*. Allowing subtraction requires negative numbers, not elements of $\mathbb{N}$. Allowing division requires fractions, also not elements of $\mathbb{N}$. Below, we present the example of clock numbers, in which the monoid is a finite set, profoundly smaller than the type, which is still the infinite set $\mathbb{N}$.

From this point on, we always mention the operation and the identity element along with the type of the elements of any monoid. If $\mathbb{N}$ is the type, $+$ the operation, and 0 the identity element, then we write the monoid above as $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$. This notation perfectly captures just the three concepts we need to generalize.

Right away, notice another monoid hiding in the weeds amongst $\mathbb{N}$, namely one with multiplication alone as the operation and 1 as the identity element. Write this monoid as $\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$. Demonstrate that it is a monoid by copying the three facts above, substituting $\times$ for $+$, "multiplication" for "addition," "multiplied by" for "added to," and 1 for 0:

- $\mathbb{N}$ is **closed** under the **binary operation** of multiplication, meaning that for any two elements a and b of $\mathbb{N}$, $a \times b$ is also an element of $\mathbb{N}$.
- Multiplication is **associative**, meaning that for any *three* elements a, b, and c of $\mathbb{N}$, $((a \times b) \times c) = (a \times (b \times c))$.
- There is a special **identity** element of $\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$, namely 1, that, when multiplied by any other element, say a, on the right or on the left, does not change its value. In other words, for any a in $\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$, $a \times 1 = a$, and $1 \times a = a$.

We now have two different monoids just amongst the familiar whole numbers. In each case, to identify the monoid, identify the type of its elements, its binary operation, and its identity element. To demonstrate it *is* a monoid, show that the operation is closed, associative, and honors the identity on both the left and right sides.

Now, consider the numbers on a clock, 1 through 12, which are of type $\mathbb{N}$, that is, elements of the set $\mathbb{N}$. We can build a monoid out of these numbers by defining an operation for the kind of wrap-around addition that clocks demonstrate mechanically: a restricted addition where 1 follows 12, called *addition modulo 12*. Write this monoid as $\mathcal{M}\langle\mathbb{N}, +_{12}, 12\rangle$, denoting the operation by "$+_{12}$" and recording the identity element as 12. The operation is closed — it takes any two numbers between 1 and 12 and returns another number between 1 and 12; it is associative — the order in which addition modulo 12 is applied amongst any three elements does not matter; 12 is the identity — adding 12 to any number on a clock face amounts to moving the hour hand once around, landing on the same number. This is our first example of a monoid that contains fewer elements than the type set, $\mathbb{N}$.

Now, generalize the symbolism. To write out a perfectly general monoid, $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$,

- Use the symbol $\mathcal{T}$ to denote the type set from which elements of the monoid are drawn, realizing that elements of $\mathcal{T}$ may need to be conditioned in some way to bring them into the monoid. This is the generalization of $\mathbb{N}$ in the examples above.

- Use the symbol $\oplus$ for the binary operation, analogous to the ordinary $+$ symbol. Read it aloud as "oh-plus" or "circle plus." This is the generalization of $+$, $\times$, and $+_{12}$ in the examples above.
- Use the symbol $\mathcal{Z}$ to stand for the identity element, analogous to 0, 1, and 12 in the examples above.

Under this generalization scheme, we might say that $\mathbb{N}$ is the $\mathcal{T}$ of $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$, that $+$ is the $\oplus$ of $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$, and that 0 is the $\mathcal{Z}$ of $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$.

The pattern is established, so abstract one last time, copying the facts from above, and making the now-obvious substitutions:

(1) $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$ is closed under the binary operation $\oplus$, meaning that for any two elements a and b of $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$, $a \oplus b$ is also an element of $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$.
(2) $\oplus$ is **associative**, meaning that for any *three* elements a, b, and c, of $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$, $((a \oplus b) \oplus c) = (a \oplus (b \oplus c))$.
(3) For any element a of $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$, $a \oplus \mathcal{Z} = a$, and $\mathcal{Z} \oplus a = a$.

When facts are written in general form like this, they are called **axioms**. To convert them into factual statements about a particular monoid, just substitute a particular set for the type, $\mathcal{T}$, a particular operation for $\oplus$, and a particular element for $\mathcal{Z}$.

Notably missing from the axioms is any direct connection between elements of $\mathcal{T}$ and elements of the monoid. The axioms are dry and parochial without this, but that is intentional. There is much we can deduce about the general monoid without referring to $\mathcal{T}$. Before showing how to process elements of $\mathcal{T}$ to bring them into the monoid, let us see what we can find out.

To satisfy the third axiom, we must show that $\mathcal{Z}$ works on both the left and the right. Suppose our monoid has *two* identities, $\mathcal{Z}_{\text{left}}$ and $\mathcal{Z}_{\text{right}}$, that satisfy the following equations: for any a in $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$,

(1) $$\mathcal{Z}_{\text{left}} \oplus a = a$$
(2) $$a \oplus \mathcal{Z}_{\text{right}} = a$$

Is there anything else in the axioms that forces $\mathcal{Z}_{\text{left}} = \mathcal{Z}_{\text{right}}$? Equation 1 certainly implies that $\mathcal{Z}_{\text{left}} \oplus \mathcal{Z}_{\text{right}} = \mathcal{Z}_{\text{right}}$, just by substituting $\mathcal{Z}_{\text{right}}$ for a. Now, however, substitute $(\mathcal{Z}_{\text{left}} \oplus \mathcal{Z}_{\text{right}})$ for $\mathcal{Z}_{\text{right}}$ in equation 2:

$$a \oplus (\mathcal{Z}_{\text{left}} \oplus \mathcal{Z}_{\text{right}}) = a$$

By associativity, rewrite this as

$$(a \oplus \mathcal{Z}_{\text{left}}) \oplus \mathcal{Z}_{\text{right}} = a$$

But equation 2 asserts that *anything* to the left of $\mathcal{Z}_{\text{right}}$ is preserved by $\oplus$, so deduce that

$$(a \oplus \mathcal{Z}_{\text{left}}) \oplus \mathcal{Z}_{\text{right}} = (a \oplus \mathcal{Z}_{\text{left}}) = a$$

So, $\mathcal{Z}_{\text{left}}$ preserves values even when it appears to the *right* of $\oplus$ with an arbitrary element, a, to the left. In other words, it acts exactly like $\mathcal{Z}_{\text{right}}$ of equation 2 in all circumstances. There is no conceivable way in which $\mathcal{Z}_{\text{left}}$ can be distinguished from $\mathcal{Z}_{\text{right}}$, so we just say that $\mathcal{Z}_{\text{left}} = \mathcal{Z}_{\text{right}}$; that is, we have proved

Lemma 1. *Associativity implies that the left and right identities in a monoid are identical.*

Notice that this lemma only applies to the identity. We cannot deduce that $a \oplus b$ is the same as $b \oplus a$ in general. This will turn out to be germane with data monoids like lists and sets. However, these considerations invite us to introduce the terms **commute** and **commutative** now:

Definition 2. *Consider a particular element* b *of monoid* $\mathcal{M}\langle \mathcal{T}, \oplus, \mathcal{Z} \rangle$. *If* $a \oplus b = b \oplus a$ *for any* a *in* $\mathcal{M}\langle \mathcal{T}, \oplus, \mathcal{Z} \rangle$, *then* b ***commutes*** *under* $\oplus$.

Definition 3. *A monoid* $\mathcal{M}\langle \mathcal{T}, \oplus, \mathcal{Z} \rangle$ *is* ***commutative*** *if all its elements commute under* $\oplus$. *Otherwise, the monoid is not commutative.*

For an example of a non-commutative monoid, consider 2×2 matrices of whole numbers under ordinary matrix multiplication:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{pmatrix} \tag{3}$$

Since the right-hand side is a matrix of whole numbers so long as each factor matrix on the left-hand side contains only whole numbers, then the set is closed under matrix multiplication. It is easy, if a bit lengthy, to check that matrix multiplication is associative. It is also easy to see that

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

is an identity. So, we have a monoid! But matrix multiplication is *not* commutative, as reversing the order of the factors shows:

$$\begin{aligned} \begin{pmatrix} e & f \\ g & h \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} &= \begin{pmatrix} ea + fc & eb + fd \\ ga + hc & gb + hd \end{pmatrix} \\ &= \begin{pmatrix} ae + cf & be + df \\ ag + ch & bg + dh \end{pmatrix} \end{aligned} \tag{4}$$

The second line in (4) is justified because the underlying operation of integer multiplication is commutative, so we may rewrite $ea + fc$ as $ae + cf$, but that is as close as we can get to the corresponding element in (3). No rearrangement of the terms will make $ae + cf$ equal to $ae + bg$ in general. For the vast majority of choices of values of the eight variables a through h from $\mathbb{N}$, the two matrix products in equations 3 and 4 above have different values.

Is the identity of any monoid unique? Suppose we have two elements, $\mathcal{Z}_1$ and $\mathcal{Z}_2$, that satisfy, for any a in $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$

$$\mathcal{Z}_1 \oplus a = a \oplus \mathcal{Z}_1 = a$$
$$\mathcal{Z}_2 \oplus a = a \oplus \mathcal{Z}_2 = a$$

So, in particular, $\mathcal{Z}_1 \oplus \mathcal{Z}_2 = \mathcal{Z}_1$, and $\mathcal{Z}_1 \oplus \mathcal{Z}_2 = \mathcal{Z}_2$, so we have proved

Lemma 2. *The identity of a monoid is unique.*

These are powerful results. If we can identify *anything* as a monoid, then we know immediately that its identity is unique and commutes. We do *not*, in general, know whether the entire monoid is commutative.

At this point, we can reiterate definition 1:

> *A **monoid**, $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$, is a set of elements of type $\mathcal{T}$, closed under an associative binary operation, $\oplus$, with an **identity**, $\mathcal{Z}$.*

and hope that you are totally comfortable with it. The only missing topic is a precise discussion of the connection between $\mathcal{T}$ and the monoid, but we address this below, in the context of collection monoids like lists and sets.

3. A Zoo of Monoids

We have seen that $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$, $\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$, and $\mathcal{M}\langle\mathbb{N}, +_{12}, 12\rangle$ are monoids. Square matrices of whole numbers under matrix multiplication also form monoids. We can devise many more such numerical examples, but the monoids of interest to data professionals are the ones that represent collections. Let us define a list structure and show the monoidal structure:

Definition 4. *A **list** is an ordered sequence of zero or more items of some type $\mathcal{T}$. A monoid of lists has the associative, non-commutative, binary operation, $+\!\!+$, for concatenating lists and the empty list as identity.*

As defined, a set of lists can be a monoid! The definition is a bit austere, though: we need notation. Write the empty list as $[\,]$ and the general list as $[a_1, a_2, \ldots]$, making sure that every element a_i is of the same type, that is, is an element of $\mathcal{T}$. Write that last assertion symbolically as $\forall i, a_i \in \mathcal{T}$, reading "$\forall$" as "for all" and "$\in$" as "in" or "is an element of." The term **ordered** means that we only need to know a number or **index** i to find an element a_i of the list. Later, we encounter some unordered monoids, wherefrom we *cannot* find an element given only its index.

The indices i come from the natural numbers excluding zero, written $\mathbb{N} - \{0\}$. This is *1-based indexing* and is the usual convention in mathematics. Occasionally, it is more convenient to start indices at 0, drawing them from all of $\mathbb{N}$, writing the general list as $[a_0, a_1, \ldots]$. This is *0-based indexing*, is an equally valid convention to 1-based indexing, and is the norm for programming languages.

We might have defined list monoids without restricting their items to come from a type set $\mathcal{T}$. For instance, we might want a list to contain both integers and ASCII characters, or nested lists. That sort of definition is called a *heterogeneous list*, like lists in the programming language Lisp. However, such is a bit out of vogue nowadays, partly due to the ascendence of strongly typed programming languages and partly due to applications to relational databases to follow. On balance, it is beneficial to name and restrict the types of elements that a particular list may contain.

The empty list as identity means that $\mathcal{L} +\!\!+ [\,] = \mathcal{L}$, $[\,] +\!\!+ \mathcal{L} = \mathcal{L}$ for any list $\mathcal{L}$.

Now, consider a couple of particular lists of integers, say $\mathcal{L}_1 = [6, 7, 8]$ and $\mathcal{L}_2 = [4, 5]$. Invoke the operation and construct

$$\mathcal{L}_1 +\!\!+ \mathcal{L}_2 = [6, 7, 8, 4, 5]$$
$$\mathcal{L}_2 +\!\!+ \mathcal{L}_1 = [4, 5, 6, 7, 8]$$

Observation 1. *Lists and $+\!\!+$ are not commutative.*

Is every set of lists a monoid? No. Consider the set of lists of no more than three integers. By concatenating suitably large elements of this set, we get lists of more than three integers, thus, the set is not closed under concatenation, and is therefore not a monoid.

But also consider the set of lists of integers where the first element of every list is the number 1. This set *is* a monoid, since there is no way to get rid of an initial 1 by concatenation! Concatenate a list beginning with a 1 to either the front or the back of a list beginning with a 1 and you will still have a list beginning with a 1. The identity element is the empty list. The first element of the empty list is a 1 by a strange but valid technicality of logic. I cannot show you any element of the empty list that is *not* a 1, therefore, the statement that the first element of the empty list is a 1 must be correct. It is impossible to prove it false, and the Law of the Excluded Middle insists that it be either true or false.

Imagine a dictionary of words considered as lists of characters. The entire dictionary might be a list monoid, and by analogy to the monoid of lists of integers with first element 1 above, each alphabetical section might also be a list monoid, but there is a stipulation. Each alphabetical section must, in fact, be infinite in length, just as must be the entire dictionary, since, given any two words, I can always create a longer word by concatenating the two. Even though each entry in the dictionary is finite in length, the entire dictionary is infinite in length, and so, in fact, is every subsection of the dictionary. Redefine a dictionary subsection to mean "the set of words with any finite prefix whatever." So, the 'a's form one subsection, as before, but so do the 'aa's and the 'ab's and the 'qu's and the 'str's and so on. This is the kind of dictionary that Jorge Luis Borges might have liked [bab].[1]

The trivial list monoid contains just the empty list and no other lists. No other list monoid can be finite under concatenation, since, given any two elements, we can always create another, longer element. Even the monoid of lists of just 1's contains the elements $[\,]$, $[1]$, $[1,1]$, and so on, without end.

There is an alternative, recursive definition for list, equivalent in every way to definition 4, above, but perhaps more familiar to programmers:

Definition 5. *A **list** of elements of type $\mathcal{T}$ is either the empty list or a singleton $[\mathfrak{a}]$ concatenated to a list, where $\mathfrak{a} \in \mathcal{T}$.*

A **singleton** is just a list containing a single element. A better way of writing this is in BNF (Backus-Naur Form):

$$\begin{array}{lcll} \mathit{List}\langle\mathcal{T}\rangle & ::= & [\,] & \textit{a list of elements of type } \mathcal{T} \textit{ is either the empty list} \\ & | & [\mathfrak{a} \in \mathcal{T}] +\!\!+ \mathit{List}\langle\mathcal{T}\rangle & \textit{or a singleton concatenated to another list} \end{array}$$

emphasizing that $\mathfrak{a} \in \mathcal{T}$.

[1] Borge's library consisted of 410-page books of 40×80 lines, from a 25-character alphabet. There are $25^{40\times80\times410} \approxeq 1.96 \times 10^{1,834,097}$ different such books. While a fairly large number, this is, existentially finite.

As before, write the monoid as $\mathcal{M}\langle \mathcal{T}, +\!\!+, [] \rangle$. The interesting bit is the first slot, the type set of the monoid, $\mathcal{T}$. The recursive version of the definition reveals that lists are built from singletons of the form $[a]$ rather than directly from elements of $\mathcal{T}$. Does this technical detail call into question our definition of a monoid, definition 1? There, we wrote that a monoid is a set of elements 'of type $\mathcal{T}$.' There is just enough wiggle room to accommodate lists as elements of a monoid if we allow 'of type $\mathcal{T}$' to mean 'elements of the set $\mathcal{T}$ or elements of the set of singleton lists made from elements of the set $\mathcal{T}$.' Let us write this latter set, the set of singleton lists, as $[\mathcal{T}]$ for short. Each element of this latter set is a special kind of element of the monoid of lists. We convert elements of the monoid's type set $\mathcal{T}$ into the derivative type set $[\mathcal{T}]$ before they can participate in the monoid. We must introduce an auxiliary operation, called **unit**, that just takes an arbitrary element of $\mathcal{T}$ and returns a singleton. Symbolize *unit* for lists as $\mathcal{U}_{[]}$, and denote the fact that it maps elements of the type set $\mathcal{T}$ to singleton elements of the monoid as follows:

$$\begin{array}{ll} \mathcal{U}_{[]} : \mathcal{T} \to [\mathcal{T}] & \mathcal{U}_{[]} \textit{ maps elements of } \mathcal{T} \textit{ to elements of } [\mathcal{T}] \\ \forall a \in \mathcal{T}, \mathcal{U}_{[]}(a) = [a] & \textit{for any } a \textit{ in } \mathcal{T}, \mathcal{U}_{[]}(a) \textit{ is the singleton } [a] \end{array}$$

3.1. **Where was the Unit?** We didn't have to go through all this with the numerical monoids $\mathcal{M}\langle \mathbb{N}, +, 0 \rangle$ and $\mathcal{M}\langle \mathbb{N}, \times, 1 \rangle$, or even with the more complicated monoid of square matrices. What gives? Why is this different? The answer is that we actually did go through all this with the numerical monoids, just under the covers, without talking about it. Now we must.

Revisit the clock-arithmetic example, $\mathcal{M}\langle \mathbb{N}, +_{12}, 12 \rangle$, the monoid of whole numbers under addition modulo 12. Now we can see that we confused the type set, $\mathbb{N}$, and the set of 'singleton' elements of the monoid, which really must be the finite set $\{1, 2, \ldots, 12\}$, henceforth written $\mathbb{N}_{12}$. There were no consequences to the confusion because this monoid contains *only* singleton elements and because the unit function is so trivial in this case. The list monoid, though, has shown us that, in general, we must map type-set elements into singletons, so now we must go back and reconsider all the prior examples and identify those unit functions.

Before an element of $a \in \mathbb{N}$ can participate in the clock-arithmetic monoid, a must undergo conditioning — be tweaked — be brought into the set of singletons, $\mathbb{N}_{12}$, by computing its residual mod 12, that is, its remainder when divided by 12. So, the unit function for $\mathcal{M}\langle \mathbb{N}, +_{12}, 12 \rangle$ is

$$\begin{array}{l} \mathcal{U}_{12} : \mathbb{N} \to \mathbb{N}_{12} \\ \forall a \in \mathbb{N}, \mathcal{U}_{12}(a) = a \bmod 12 \end{array}$$

Even the monoids $\mathcal{M}\langle \mathbb{N}, +, 0 \rangle$ and $\mathcal{M}\langle \mathbb{N}, \times, 1 \rangle$ have unit functions:

$$\begin{array}{ll} \mathcal{U}_{+,0} : \mathbb{N} \to \mathbb{N} & \mathcal{U}_{\times,1} : \mathbb{N} \to \mathbb{N} \\ \forall a \in \mathbb{N}, \mathcal{U}_{+,0}(a) = a & \forall a \in \mathbb{N}, \mathcal{U}_{\times,1}(a) = a \end{array}$$

All these unit functions are pretty trivial, but these last ones are about as trivial as it gets: they just return their arguments without change. The function that does such is called *id*, the **identity function**, not to be confused with the identity *element* of a monoid. It is useful to symbolize this function in **lambda notation**. Read λx as "the function of x that…":

$$\mathrm{id} = \lambda x.x, \quad \textit{id is the function of } x \textit{ that returns } x \textit{ for any } x$$

The point is that every monoid must have a unit function to convert elements of the type set $\mathcal{T}$ into singleton elements of the monoid. We *might*, at this point, force ourselves to be so pedantic as to rewrite every monoid with its unit function and set of singletons, like this:

$$\mathcal{M}\langle \mathbb{N}, \lambda x.(x \bmod 12), \mathbb{N}_{12}, +_{12}, 12\rangle$$

But that is really going too far, because, although the unit function and sets of singletons *exist* for every monoid, they are so trivial that usually we just need to note them once and then forget them. They are a little like the door to a house: not, by any means, the most interesting part of the house, and not worth talking about at all once one has passed through it. But it is an essential part. Later, we find that careful accounting for *unit* leads us from monoids to monads, their practical cousin in programming languages.

3.2. **The Collection Monoids.** Lists are our first example of a **collection monoid**. There are three other kinds: *set*, *bag*, and *permutation*. They differ in whether they are ordered and whether they are *idempotent*, and that is why there are just four: two choices for 'ordered or not' and two, independent choices for 'idempotent or not.' We have already explained the concept of *ordered*. Let us define idempotent:

Definition 6. *An operation, $\oplus$, is **idempotent** if $x \oplus x = x$ for any x in the domain of $\oplus$.*

Definition 7. *A collection is **idempotent** if no duplicate elements are allowed.*

We see below that idempotency of an operation does not suffice, in general, to guarantee idempotency of a collection. So, the two definitions, while related, are distinct.

A list monoid is neither commutative nor idempotent, so, in some sense, it is the simplest of the four kinds. A *permutation* monoid is idempotent and non-commutative. Consider the ordered sequence $(\!|1, 2, 3|\!)$. Because it is ordered, it is distinct from $(\!|2, 1, 3|\!)$, and, in fact, from all four other permutations or orderings of the numbers 1, 2, and 3. Because $(\!|1, 2, 3|\!)$ is idempotent, we cannot add, prepend, or append another copy of 1, 2, or 3 to it.

Trial Definition 1. *A **permutation** is an ordered sequence of zero or more items of some type $\mathcal{T}$. A monoid of permutations has the associative, non-commutative, idempotent, binary operation, $\boxplus$ (pronounced 'box-plus'), for concatenating permutations, and the empty permutation, $(\!|\,|\!)$ (pronounced 'bananas'),[2] as identity.*

[2]It is with some regret that I use banana brackets for permutations, since Meijer, Trust, and others use them for catamorphisms. However, I was not able to find another viable bracket form for permutations amongst the mutually compatible font sets I have at hand.

As stated, a collection of permutations can be a monoid, but we must be more precise concerning idempotency. The first task is to 'design' $\boxplus$: to write out its operational rules and see if we can build with it a collection that contains no duplicate elements. Let s and t be distinct elements of $\mathcal{T}$, and let $(\!|\,|\!)$ be the empty permutation. Write the type of *permutations of items of type* $\mathcal{T}$ as $(\!|\mathcal{T}|\!)$. The unit function will map elements of $\mathcal{T}$ to singleton elements of $(\!|\mathcal{T}|\!)$:

$$\begin{array}{ll} \mathcal{U}_{(\!|\,|\!)} : \mathcal{T} \to (\!|\mathcal{T}|\!) & \mathcal{U}_{(\!|\,|\!)} \textit{ maps elements of } \mathcal{T} \textit{ to elements of } (\!|\mathcal{T}|\!) \\ \forall s \in \mathcal{T}, \mathcal{U}_{(\!|\,|\!)}(s) = (\!|s|\!) & \textit{for any } s \textit{ in } \mathcal{T},\ \mathcal{U}_{(\!|\,|\!)}(s) \textit{ is the singleton } (\!|s|\!) \end{array}$$

Now, build up some base cases. We want $\boxplus$ to satisfy the following equations.

$$\begin{aligned} (\!|s|\!) \boxplus (\!|\,|\!) &= (\!|s|\!) \\ (\!|\,|\!) \boxplus (\!|s|\!) &= (\!|s|\!) \\ (\!|s|\!) \boxplus (\!|t|\!) &= (\!|s, t|\!) \\ (\!|t|\!) \boxplus (\!|s|\!) &= (\!|t, s|\!) \end{aligned}$$

Since permutations are ordered, then $(\!|s, t|\!) \neq (\!|t, s|\!)$ when $s \neq t$:

Observation 2. *Permutations are not commutative and* $\boxplus$ *is not commutative.*

just as with lists. Still working with singletons, we want

$$(\!|s|\!) \boxplus (\!|s|\!) = (\!|s|\!)$$

This is idempotency of $\boxplus$, but it is not quite enough to guarantee no duplicates in an arbitrary permutation. So far, we only know (1) how to deal with the identity element and (2) how to create doubletons from singletons. We do not know how to apply $\boxplus$ to an element like $(\!|s, t|\!)$ with more than one item in it. There are several ways to proceed, but let us try the recursive style, just as with lists:

Trial Definition 2. *A permutation* $\mathcal{P}$ *of elements of type* $\mathcal{T}$ *is either the empty permutation* $(\!|\,|\!)$ *or a singleton permutation,* $(\!|s|\!)$*, where* $s \in \mathcal{T}$*, associatively, non-commutatively, and idempotently concatenated to another permutation, that is,* $\mathcal{P}_1 = (\!|s|\!) \boxplus \mathcal{P}_2$*.*

Now, consider an arbitrary non-empty permutation, $(\!|s, t, \ldots|\!)$=$(\!|s|\!) \boxplus \mathcal{P}$, and try to concatenate $(\!|s|\!)$ to it. Due to associativity and binary idempotency, we get

$$\begin{aligned} (\!|s|\!) \boxplus \Big((\!|s|\!) \boxplus \mathcal{P} \Big) &= \Big((\!|s|\!) \boxplus (\!|s|\!) \Big) \boxplus \mathcal{P} \\ &= (\!|s|\!) \boxplus \mathcal{P} \end{aligned}$$

So, adding an element $(\!|s|\!)$ to the front of a permutation $\mathcal{P}$ that already has $(\!|s|\!)$ at the front does not change $\mathcal{P}$. But suppose $(\!|s|\!)$ is buried inside $\mathcal{P}$. Nothing in the design of $\boxplus$ so far prevents us from building, say, $(\!|s, t, s|\!)$, which would have an unacceptable duplicate element. We must conclude:

Observation 3. *Idempotency of an operation does not guarantee idempotency of a collection.*

We are forced to redesign $\boxplus$. Write the arbitrary finite permutation of length $n \geqslant 1$ as $\mathcal{P}=(\!|s_1, s_2, \ldots, s_n|\!)$, where all the s_i are distinct, that is, no two of the elements are equal to one another. Now concatenate a singleton permutation, $(\!|s|\!)$, to the front or back of $\mathcal{P}$:

$$(\!|s|\!) \boxplus (\!|s_1, s_2, \ldots, s_n|\!) = \begin{cases} (\!|s_1, s_2, \ldots, s_n|\!) & \text{if } s \text{ equals any of the } s_i,\ i = 1 \text{ to } n, \\ (\!|s, s_1, s_2, \ldots, s_n|\!) & \text{if } s \text{ equals none of the } s_i. \end{cases} \tag{5}$$

$$(\!|s_1, s_2, \ldots, s_n|\!) \boxplus (\!|s|\!) = \begin{cases} (\!|s_1, s_2, \ldots, s_n|\!) & \text{if } s \text{ equals any of the } s_i,\ i = 1 \text{ to } n, \\ (\!|s_1, s_2, \ldots, s_n, s|\!) & \text{if } s \text{ equals none of the } s_i. \end{cases} \tag{6}$$

If the starting permutation $\mathcal{P}$ has no duplicates, the rules above make it impossible for there *ever* to be any duplicates, since $\boxplus$ does not change $\mathcal{P}$ in that case.

The two rules above let us concatenate singletons to the front or to the back of any permutation. Letting $n = 1$ and $s_1 = s$ in the above shows the new $\boxplus$ is an idempotent operation, under definition 6, that also preserves the idempotency of a permutation. But it is not yet complete as the operation of a permutation monoid, because it can only concatenate singletons to elements of a monoid. Fixing this is the final detail in the new design of $\boxplus$. First, to see pointedly why it needs fixing, which of the following two is true when $s \neq t$?

$$\begin{aligned} (\!|s, t|\!) \boxplus (\!|t, s|\!) &= (\!|s, t|\!) \\ (\!|s, t|\!) \boxplus (\!|t, s|\!) &= (\!|t, s|\!) \end{aligned} \tag{7}$$

They cannot both be true since we require all monoid operations to give deterministic results, that is, we require them to be single-valued mappings or *functions* in the ordinary sense. We are forced to make an arbitrary decision, and that will be as follows:

> *If both the left-hand and right-hand arguments of $\boxplus$ are larger than singletons, then $\boxplus$ will* ***deconstruct*** *its right-hand argument $(\!|s, s_1, s_2, \ldots, s_n|\!)$ into $(\!|s|\!) \boxplus (\!|s_1, s_2, \ldots, s_n|\!)$. By associativity, repeated applications of equation 6 will concatenate the resulting singletons into the right-hand end of the left-hand argument of the original $\boxplus$ operation, yielding an unambiguous result.*

With this final rule, resolve the ambiguity above as follows:

$$\begin{aligned} (\!|s, t|\!) \boxplus (\!|t, s|\!) &\to (\!|s, t|\!) \boxplus \Big((\!|t|\!) \boxplus (\!|s|\!) \Big) && \textit{deconstruct the right-hand argument} \\ &= \Big((\!|s, t|\!) \boxplus (\!|t|\!) \Big) \boxplus (\!|s|\!) && \textit{associativity} \\ &= (\!|s, t|\!) \boxplus (\!|s|\!) = (\!|s, t|\!) \end{aligned}$$

At first glance, this deconstruction rule may seem weird, but it really is just another case of *déjà vu*. The recursive definition of lists (definition 5) implied it, in effect. That definition showed *only* how to construct lists with $+\!\!+$ by concatenating singletons to the front of a pre-existing list. So, actually, to concatenate two lists larger than singletons requires deconstructing the right-hand argument to the form $[s]+\!\!+[s_1, s_2, \ldots, s_n]$. We did not have

to sweat the details with lists because without idempotency, there was no chance of ambiguity like equation 7 under any detailed operational definition of ⧺. But, this kind of deconstruction is implicit in the all-but-mandatory recursive programming patterns for lists in functional programming — in Scheme and Haskell, for instance.

In Scheme, a non-empty list is written as

$$\begin{aligned} \textit{list}_1 &= (\texttt{cons}\ \textit{singleton-preimage}\ \textit{list}_2) \\ \textit{singleton-preimage} &= (\texttt{car}\ \textit{list}_1) \\ \textit{list}_2 &= (\texttt{cdr}\ \textit{list}_1) \end{aligned}$$

which effectively defines the primitive functions `cons`, `car`, and `cdr`. The *singleton-preimage* is simply an element of the type set, $\mathcal{T}$, which, when subjected implicitly to the unit function, $\mathcal{U}_{[]}$, becomes a singleton list. Alternatively, we could explicitly create the singleton using Scheme's `list` primitive and concatenate it to the front of the list via Scheme's version of the list-monoidal operation ⧺, written `append`.

Cons builds lists from singleton preimages rather than from explicit singletons. This design reflects a representation of a list as a *monad* rather than as a *monoid*. The reason is to finesse the use of the pesky unit function. It is annoying, when writing a program in a practical programming language, to have to write the unit function every single time an element of $\mathcal{T}$ is *cons*ed to the front of a list. *Append* is the Scheme primitive function supporting monoidal list construction; *cons* is the primitive function supporting monadic list construction. There is more to say on this in the section on monoid homomorphisms.

In Haskell, deconstruction appears directly in pattern-matching forms. Monadic lists are often written similarly to Scheme, as `cons` pairs, `(x:xs)`, where `x` is a singleton preimage; `xs` is, recursively, another list; and `:` is the *deconstruction* operator, in a pattern-matching context, and is the *construction* operator in other contexts.

The following shows how to write the *length* function in these two languages. Later, we see that these are monadic forms of *monoid homomorphisms*. These particular monoid homomorphisms are functions from list monoids to $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$ — functions that will convert any member of a list monoid to a natural number:

```
(define (length list)
  (cond
    ((null? list) 0)
    (else (+ 1 (length (cdr list))))))
```

```
length [] = 0
length (x:xs) = 1 + length xs
```

Finally, notice that equation 5 is not necessary, since, with a right-hand deconstruction rule, all permutation concatenations with ⊞ resolve into a form matching 6.

Let us generalize equation 6 beyond just permutations and $\boxplus$. We already have a generic stand-in for a monoidal operation, namely $\oplus$. We use square brackets $[\ldots]$ for lists and banana brackets $(\!|\ldots|\!)$ for permutations; later we use curly braces $\{\ldots\}$ for sets and bag brackets $\wr\ldots\int$ for, well, bags. We need a generic stand-in for brackets of all kinds to generalize 6, and ordinary parentheses fit the bill, reserving angle brackets for generic type arguments as with our monoid notation. So, just let $(s_1, s_2, \ldots, s_n)$ represent the generic collection, which may be ordered or not and idempotent or not, and consider:

Definition 8. *An associative operation, $\oplus$, is **recursively idempotent** if, for any collection $\mathcal{C} = (s_1, s_2, \ldots, s_n)$, $n \geqslant 1$, and any singleton (s),*

$$(s_1, s_2, \ldots, s_n) \oplus (s) = \begin{cases} (s_1, s_2, \ldots, s_n) & \text{if } s \text{ equals one of the } s_i,\ i = 1 \text{ to } n, \\ (s_1, s_2, \ldots, s_n, s) & \text{if } s \text{ equals none of the } s_i. \end{cases}$$

$\oplus$ must also satisfy $(s) \oplus () = () \oplus (s) = (s)$ and operationally enforce a right-deconstruction rule, whereby operands on the right-hand side are deconstructed into a stream of singletons.■

By letting $n = 1$ and $s_1 = s$, we see immediately that

Lemma 3. *A recursively idempotent operation is also idempotent under definition 6.*

though the reverse is not always true. The following is also evident:

Lemma 4. *A collection under a recursively idempotent operation is an idempotent collection under definition 7.*

Now, we have enough to write:

Definition 9. *A **permutation** is an ordered, idempotent collection of zero or more items of some type $\mathcal{T}$. A monoid of permutations has the associative, non-commutative,* recursively *idempotent, binary operation, $\boxplus$, for concatenating permutations, and the empty permutation, $(\!|\,|\!)$, for identity.*

In other words, a monoid of permutations is a set of permutations under $\boxplus$ and $(\!|\,|\!)$, just as a monoid of lists is a set of lists under $+\!\!+$ and $[\,]$. The operations, properly designed, imply the presence or lack of idempotency.

Permutation monoids, unlike list monoids, can be finite. For instance, the monoid of permutations of the integers 1 through 3 has sixteen elements:

$$\begin{array}{cccccc} & (\!|\,|\!) & (\!|1|\!) & (\!|2|\!) & (\!|3|\!) & \\ (\!|1,2|\!) & (\!|1,3|\!) & (\!|2,1|\!) & (\!|2,3|\!) & (\!|3,1|\!) & (\!|3,1|\!) \\ (\!|1,2,3|\!) & (\!|2,1,3|\!) & (\!|2,3,1|\!) & (\!|3,2,1|\!) & (\!|3,1,2|\!) & (\!|1,3,2|\!) \end{array}$$

In fact, we can write a general formula for the size of any permutation monoid built on a finite type set, $\mathcal{T}$. If $\mathcal{T}$ contains n elements, then a permutation monoid built on it contains elements of all lengths from 0 to n, as in the example above, where $n = 3$. The number of elements of length 0 is just 1, because it is the identity element of the monoid and it is unique. The number of elements of length 1, the number of singletons, is just n, because

there are n choices for the sole element of each singleton. Notice that the term 'element' appears twice in the last sentence, once to mean 'element of the permutation monoid,' and once to mean 'element of the element of the monoid,' which is, in turn, an element of the type set $\mathcal{T}$. Such stacking of terms is unavoidable when discussing unit functions and collection monoids. The number of doubletons is $n(n-1)$ because there are n choices for the first element of the doubleton and $n-1$ independent choices for the second element. We can see that the number of permutations of length m, for every $m \geqslant 0$ and $m \leqslant n$, must be $n!/(n-m)!$, and the total number of elements in the monoid must be

$$|\mathcal{P}_n| = \sum_{m=0}^{n} \frac{n!}{(n-m)!} \tag{8}$$

This is not trivial to solve off-the-cuff, meaning that it is not easy to write an expression in n simpler than the overt sum. But notice that if we exclude the identity element, labeled with $m = 0$, then write the sum as

$$\begin{aligned}
S_n &\stackrel{\text{def}}{=} n(n-1)(n-2)\ldots 2\cdot 1 \\
&+ n(n-1)(n-2)\ldots 2 \\
&+ \cdots \\
&+ n(n-1)(n-2) \\
&+ n(n-1) \\
&+ n \\
&= n \left(\begin{array}{l} (n-1)(n-2)\ldots 2\cdot 1 \; + \\ (n-1)(n-2)\ldots 2 \; + \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ (n-1)(n-2) \; + \\ (n-1) \; + \\ 1 \end{array} \right) \\
S_n &= n(1 + S_{n-1})
\end{aligned}$$

we see that S_n is a simple function of the next lower S_{n-1}. Expressions of this form are *recurrence relations* and are natural for linear-time computation in functional programming languages. While perhaps not as pretty as a closed-form formula in n, they are certainly practical and ubiquitous.

We must be careful to set the base of the recurrence correctly. S_n is one less than the size of a permutation monoid drawn from a base-type set of length n, so $S_1 = 1$ because the permutation monoid drawn from a base-type set of one element, say $\mathcal{T} = \{1\}$, has two elements, namely $(\!|\,|\!)$ and $(\!|1|\!)$. Starting off this way, we see that the first few values of S_n are

$$[1, 4, 15, 64, 325, 1956, 13699, 109600, \ldots]$$

Each term of this is one less than the overt sum in equation 8, as expected. This series is currently known as Sloan number A007526, and the overt sum is known as A000522. These sequences have have a long history going back more than three centuries to the

golden age of combinatorics. Some literature on it can be looked up at the Online Encyclopedia of Integer Sequences [oei], specifically [A26] and [A22].

Should we mistakenly start our recurrence relation at $S_0 = 1$, however, we get

$$[2, 6, 21, 88, 445, 2676, 18739, 149920, \ldots]$$

a series with a very different history, usage, meaning, and background [A40].[3]

S_n is a bit larger than factorial, which has recurrence relation $S_n = n(S_{n-1})$, and as the following table and logarithmic plot of numerical values shows

n	S_n	$n!$
1	1	1
2	4	2
3	15	6
4	64	24
5	325	120
6	1956	720

The plot suggests that the ratio of S_n and $n!$ converges to a constant, and a little experimentation, in lieu of analytical proof, reveals that constant to be $e = 2.71828...$ In other words, as n gets larger, S_n approaches $e \times n!$, a convenient factoid.

Recall the imaginary dictionary containing an infinite number of alphabetically arranged subsections, each infinite in length. This dictionary was a mental image of the monoid of *lists* of alphabetical characters, also known as the monoid of character strings under list concatenation. Should we artificially limit the size of words in such a dictionary, we would no longer have a monoid, because the limited dictionary would not be closed under concatenation. But, we *can* build a finite dictionary out of permutations. Such a dictionary would have the stupendous size

$$1 + S_{26} = 1,096,259,850,353,149,530,222,034,277$$

[3] As of late 2005, there is some sort of known problem with the http interface to the Online Encyclopedia of Integer Sequences, and the cite owners recommend sending an empty email to 'sequences@research.att.com' for instructions.

ignoring differences between upper and lower case. This is a little larger than 1 billion billion billion. To set an intuition of the magnitude of this number, it is approximately the number of kilograms in the mass of the Earth. But it is finite.

Of course, this dictionary would suffer from the limitation that no duplicate letters would be permitted. This limitation is easily relieved, again in a finite fashion. Suppose we decide that words with more than four instances of a given letter are not interesting. Then, we can augment our alphabet, the type set $\mathcal{T}$, with three additional copies of each letter, artificially distinguished from one another by a subscript:

$$\{a, b, \ldots, z\} \to \{a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4, \ldots, z_1, z_2, z_3, z_4\}$$

This device is just an application of *disjoint union* from set theory. Now, our dictionary will have words with repeated letters, and will be of the more stupendous size:

$$1 + S_{104} \approxeq 2.8 \times 10^{166}$$

which is approximately the square of the number of electrons in the observable Universe. So, while this new dictionary is technically finite, and is vastly smaller than Borges' library, it could never be physically written down, so, it is purely conceptual, once again.

The design of $\boxplus$ opens up can of worms of practical ramifications. A computer program implementing the operation for idempotent collection monoids must have a way of detecting whether a candidate element is already in a collection. The straightforward linear search — iterating over the entire collection — is only practical for small collections. Other search structures, like hash tables or B-trees, are a practical necessity in any scalable implementation.

The non-idempotent (infinite) collection monoids do not escape scot-free. If they are unordered, they must also have search optimizations, not for insertion, but for finding and grouping. Only lists can scalably perform both insertion and finding, at least finding-by-numerical-index, without search optimizations (though scalable finding requires array implementation with other ramifications on recursive programming patterns).

Let us round out the zoo of collection monoids with the following definitions:

Definition 10. *A **set** is an unordered, idempotent collection of zero or more items of some type $\mathcal{T}$. A monoid of sets has the associative, commutative, recursively idempotent, binary operation, $\cup$, for combining sets, and the empty set, $\{\} = \emptyset$, for identity.*

Just as a monoid of permutations can be finite, so can a monoid of sets. Again, suppose a base type set $\mathcal{T}$ with n elements. The number of elements of a set monoid drawn from $\mathcal{T}$ is the sum of the number of sets of length m for all $m \geqslant 0$ to $m \leqslant n$. The number of sets of length m turns out to be

$$\frac{n!}{(n-m)!m!} \stackrel{\text{def}}{=} \binom{n}{m}$$

which is the binomial coefficient for n and m. This is the same as the number of permutations of length m divided by $m!$ because now, the order does not matter, and there are

$m!$ different ways to rearrange the elements of a permutation of length m. The sum of the number of sets for all m is the same as the binomial $(x+y)^n$ evaluated at $x=1, y=1$. This is

$$\sum_{m=0}^{n} \frac{n!}{(n-m)!m!} = \binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n-1} + \binom{n}{n}$$

$$= (x+y)^n|_{x=1,y=1} = 2^n$$

Another way of seeing the result is that each set in the monoid is the result of independently choosing whether to include each element of $\mathcal{T}$. Since for each element of $\mathcal{T}$, there are two independent choices — to include or not — and there are n elements of $\mathcal{T}$, there must be exactly 2^n sets. The monoid is just the *power set* of $\mathcal{T}$, that is, the set of all subsets of $\mathcal{T}$.

To relate this to databases, consider a set of employees:

```
John Smith
Jean Green
Joe Snow
```

Think of this set as a type set, $\mathcal{T}$, and think of the monoid of sets drawn from this type set. This monoid would be the collection of all subsets of $\mathcal{T}$ — the power set of $\mathcal{T}$, and would have eight elements. Its monoid combination operation would be set union, $\cup$. The next section of this tutorial exhibits transformations from such a monoid to others, and how such transformations effect various 'database queries' over this table of employees.

We might examine the idea of a 'dictionary' built from sets of characters, just as we did for lists and permutations, but this would be a boring one, since for every choice of characters, no matter what their order, there would be only one entry. Since the essence of words built of characters is order, the monoid of sets of characters would hardly rate the moniker 'dictionary.'

Definition 11. *A **bag** or **multiset** is an unordered collection of zero or more items of some type $\mathcal{T}$. A monoid of bags has the associative, commutative, non-idempotent, binary operation, $\uplus$, for combining bags, and the empty bag, $\{\!\{\}\!\}$, for identity.*

A bag is like a list, except that order does not matter. So, $\{\!\{1,1,2,3\}\!\}$ is the same as $\{\!\{2,1,3,1\}\!\}$ and the same as all twenty-two other arrangements of 1, 1, 2, and 3. Just as with lists, there are no finite bag monoids other than the trivial bag monoid, which contains only the empty bag. Just as with lists, the reason is that from any two bags, we can form a larger bag by merging the two *via* the bag-combination operation, $\uplus$.

Observation 4. *All non-idempotent collection monoids are infinite.*

In the database context, *bag* is the default type, because databases permit duplicates. So, a table of employees, without further conditioning, might look like this:

```
John Smith
Jean Green
John Smith
Joe Snow
```

This table is obviously not the full monoid of bags of such names — such a monoid is infinite. It is just a particular element of the monoid — a particular bag of those names.

The two copies of `John Smith` are most probably not an error. It would not be surprising to have two employees with the same name, but it is incumbent on the database designer to do something to distinguish them. The usual thing is to add a column containing an arbitrary but unique employee number for each employee, and then to set a *database constraint* declaring that number to be a *primary key* in the table. This is an operational constraint that must be maintained at run time, incurring computational cost, as discussed above. It effectively transforms the *bag* of employees to a *set* of employees through a device equivalent to the disjoint union, also discussed above. There is much more about such transformations in the next section of this tutorial, but, suffice it to say, while *bag* is the default in databases, *set* is quite often the desired behavior.

3.3. **Summary.** The following tables summarize everything established so far and submits a few more non-collection samples for your consideration ($\mathbb{Z}$ is the set of all integers, positive, negative, and zero; in the last column of the second table, C and I stand for commutative and idempotent, respectively). While perusing the tables, remember that, especially in the case of the collection monoids, the interpretation of the definition of a monoid, "$\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$ ... a set of elements of type $\mathcal{T}$ ...", must include subjecting the elements of $\mathcal{T}$ to the unit function to bring them into the monoid as singletons.

$\mathcal{M}\langle\mathbb{N}, +, 0\rangle$	Natural numbers under addition only
$\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$	Natural numbers under multiplication only
$\mathcal{M}\langle\mathbb{N}, \max_{\mathbb{N}}, 0\rangle$	Natural numbers with a binary *max* operation
$\mathcal{M}\langle\mathbb{N} \cup \{\infty\}, \min_{\mathbb{N}}, \infty\rangle$	Natural numbers and Infinity with a binary *min* operation
$\mathcal{M}\langle\mathbb{Z} \cup \{-\infty\}, \max_{\mathbb{Z}}, -\infty\rangle$	All whole numbers and –Infinity & binary *max* operation
$\mathcal{M}\langle\mathbb{Z} \cup \{\infty\}, \min_{\mathbb{Z}}, \infty\rangle$	All whole numbers and Infinity & binary *min* operation
$\mathcal{M}\langle\text{bool}, \vee, \text{false}\rangle$	The set bool $=\{\text{true}, \text{false}\}$ with logical *or* operation, $\vee$
$\mathcal{M}\langle\text{bool}, \wedge, \text{true}\rangle$	The set bool with logical *and* operation, $\wedge$
$\mathcal{M}\langle\mathcal{T}, +\!\!+, [\,]\rangle$	lists of elements that are of type $\mathcal{T}$
$\mathcal{M}\langle\mathcal{T}, \boxplus, (\!\mid\!)\rangle$	permutations of elements that are of type $\mathcal{T}$
$\mathcal{M}\langle\mathcal{T}, \cup, \emptyset\rangle$	sets of elements that are of type $\mathcal{T}$
$\mathcal{M}\langle\mathcal{T}, \uplus, \{\!\!\{\}\!\!\}\rangle$	bags of elements that are of type $\mathcal{T}$

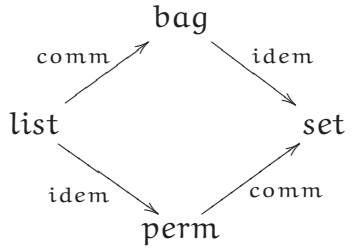
Notice that the operation, alone, suffices to identify each kind of monoid. In later sections, we may feel free to abbreviate the notation by saying, for instance, that $\boxplus$ denotes a monoid of permutations or that $\vee$ denotes the *or* monoid.

monoid	singleton type	$\oplus$	$\mathcal{Z}$	$\mathcal{U}$	C/I
$\mathcal{M}\langle\mathbb{N}, +, 0\rangle$	$\mathbb{N}$	$+$	0	$\mathrm{id} = \lambda x.x$	C
$\mathcal{M}\langle\mathbb{N}, \times, 1\rangle$	$\mathbb{N}$	$\times$	1	id	C
$\mathcal{M}\langle\mathbb{N}, \max_{\mathbb{N}}, 0\rangle$	$\mathbb{N}$	$\max_{\mathbb{N}}$	0	id	CI
$\mathcal{M}\langle\mathbb{N} \cup \{\infty\}, \min_{\mathbb{N}}, \infty\rangle$	$\mathbb{N} \cup \{\infty\}$	$\min_{\mathbb{N}}$	∞	id	CI
$\mathcal{M}\langle\mathbb{Z} \cup \{-\infty\}, \max_{\mathbb{Z}}, -\infty\rangle$	$\mathbb{Z} \cup \{-\infty\}$	$\max_{\mathbb{Z}}$	$-\infty$	id	CI
$\mathcal{M}\langle\mathbb{Z} \cup \{\infty\}, \min_{\mathbb{Z}}, \infty\rangle$	$\mathbb{Z} \cup \{\infty\}$	$\min_{\mathbb{Z}}$	∞	id	CI
$\mathcal{M}\langle\mathrm{bool}, \vee, \mathrm{false}\rangle$	bool	$\vee$	false	id	CI
$\mathcal{M}\langle\mathrm{bool}, \wedge, \mathrm{true}\rangle$	bool	$\wedge$	true	id	CI
lists	$[\mathcal{T}]$	$+\!\!+$	$[\,]$	$\lambda x.[x]$	neither
permutations	$(\!\|\mathcal{T}\|\!)$	$\boxplus$	$(\!\|\,\|\!)$	$\lambda x.(\!\|x\|\!)$	I
sets	$\{\mathcal{T}\}$	$\cup$	$\{\} = \emptyset$	$\lambda x.\{x\}$	CI
bags	$\wr\mathcal{T}\wr$	$\uplus$	$\wr\,\wr$	$\lambda x.\wr x\wr$	C

Here is another table that categorizes the kinds of collection monoids, making it obvious why there are just four kinds:

	idempotent	**not idempotent**
commutative	set $\{\mathcal{T}\}$	bag $\wr\mathcal{T}\wr$
not commutative	permutation $(\!\|\mathcal{T}\|\!)$	list $[\mathcal{T}]$

Another effective way to look at this information is as follows:



List is the most primitive kind of collection monoid, because it has neither of the qualifications of commutativity (order-independence) or idempotency (lack of duplicates). Add commutativity to a list monoid, and get a bag monoid. Add idempotency to a bag monoid, get a set monoid. Add idempotency to a list monoid, get a permutation monoid. Add commutativity to a permutation monoid, get a set monoid. Add both commutativity and idempotency to a list monoid, by either pathway, get a set monoid.

And the final way to look at it is to say the two kinds of commutative collection monoid are *set* and *bag* and the two kinds of idempotent collection monoid are *set* and *permutation*.

As another abbreviation, it can be convenient to denote the types or kinds of collection monoids using just the type symbols for their singleton elements, namely $[\mathcal{T}]$, $(\!|\mathcal{T}|\!)$, $\{\mathcal{T}\}$, and $\wr\mathcal{T}\int$, and to write $(\mathcal{T})$ to be a generic catch-all symbol meaning any of the four kinds of collection monoid. This abbreviation does run the risk of confounding the type of a collection monoid with the type of the subset containing just singleton members, but at this point, we hope the distinction is so familiar that the overloading of the type symbol does not cause heartburn. We feel free to use these abbreviations in the sequel.

4. Mapping One Monoid to Another

Once we figure out how to convert elements of one monoid to elements of another, we have everything we need for all kinds of queries. The queries familiar from relational databases *are* nothing more nor less than transformations from one monoid to another. Think of a query that gets a list of distinct salaries from an employee database. Database practitioners will recognize this as

```
SELECT DISTINCT e.salary
FROM employees e
```

and we will recognize it as transformation from a collection monoid of employees to a monoid of sets of salaries. Given *any* particular collection of employees — an element of a collection monoid of employees, the transformation will return the correct set of salaries — an element of a monoid of sets of salaries. If we had left off the `DISTINCT`, we would have gotten an element of a monoid of *bags* of salaries, and the statement above would have been a transformation from a collection monoid of employees to a monoid of bags of salaries.

In the following, we show how to construct arbitrary monoid-to-monoid transformations, and argue, by example, that any query one would care to make over database tables, over XML streams, or over structured graphs of objects in memory is such a transformation.

Let $\mathcal{M}_1 = \mathcal{M}\langle \mathcal{T}, \oplus_1, \mathcal{Z}_1 \rangle$ and $\mathcal{M}_2 = \mathcal{M}\langle \mathcal{S}, \oplus_2, \mathcal{Z}_2 \rangle$ be two monoids.

Definition 12. *A **monoid homomorphism** is a transformation from $\mathcal{M}_1$ to $\mathcal{M}_2$ — a prescription for converting elements of $\mathcal{M}_1$ to elements of $\mathcal{M}_2$ — that preserves the monoid operations.*

The rest of this section explains this definition. Note the following fine points:

(1) A **mapping** is a generic prescription for converting inputs, drawn from a **domain** set, to outputs, elements of a **range** or **codomain** set. A mapping may not have an output defined for every conceivable input, and it may have more than one output defined for some particular inputs.
(2) A **partial function** is a mapping that does not have an output for every input, but has a single output when it has any output.
(3) A **function** is a mapping that has a single output value for every input value in its domain. A function **covers** its domain. The fact may be emphasized by calling it a **total function**.
(4) A **surjection** is a function that covers the codomain, meaning that for every value in the codomain there is at least one element of the domain that maps to it. Synonyms are **surjective function** or **onto function**, with the word 'onto' being abused as an adjective.

(5) An **injection** is a function with the property that any particular output value corresponds to at most one input value. Synonyms are **injective function** or **one-to-one function**.
(6) A **bijection** or a **one-to-one correspondence** is a function that is both an injection and a surjection.
(7) A **multi-valued function** or **non-deterministic function** is a mapping that has more than one output for at least one particular input. It is not a function, so the English, here, is contradictory. But this is the standard nomenclature.
(8) The inverse of a surjection is either a multi-valued injection or an injection.
(9) The inverse of an injection is either a partial surjection or a surjection.
(10) The inverse of a bijection is a bijection.
(11) An **isomorphism** is a bijection that preserves all operations and structure. The domain and codomain of an isomorphism are **isomorphic**. For all intents and purposes, isomorphic sets are identical.
(12) A **transformation** is a total function with some additional proviso. In the case of monoid homomorphisms, the proviso is 'preserving the monoid operations.'

Start off with a concrete example: let $\mathcal{M}_1$ be $\mathcal{M}\langle\mathcal{T}, +\!\!+, [\,]\rangle$, the monoid of lists of elements of any type $\mathcal{T}$, and $\mathcal{M}_2$ be $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$, the monoid of integers under addition. Let c be the particular homomorphism that counts the number of members of a list, defined recursively as follows:

$$\begin{aligned} c([\,]) &= 0 \\ c([x]+\!\!+\mathcal{L}) &= 1 + c(\mathcal{L}) \end{aligned}$$

where $[x]$ is a singleton list and $\mathcal{L}$ is any list in $\mathcal{M}_1$. It is clear that c has $\mathcal{M}_1$ covered, meaning that no matter what list from $\mathcal{M}_1$ you give to c, it will give an answer. It is also clear that every application of c will yield just one answer, so c is, indeed, a function from $\mathcal{M}_1$ to $\mathcal{M}_2$.

The most important thing to note is how $+\!\!+$ operations in $\mathcal{M}_1$ become $+$ operations in $\mathcal{M}_2$. The homomorphism, c, not only converts elements of $\mathcal{M}_1$ to elements of $\mathcal{M}_2$, but converts the *operation* of $\mathcal{M}_1$, in parallel, to the *operation* of $\mathcal{M}_2$. Thus, c, while it is a function, is not *just* a function, but a transformation. The mapping of operations holds not just for singletons, but for arbitrary lists in $\mathcal{M}_1$ Let us deduce, from the above, that

$$c(\mathcal{L}_1+\!\!+\mathcal{L}_2) = c(\mathcal{L}_1) + c(\mathcal{L}_2) \tag{9}$$

where $\mathcal{L}_1$ and $\mathcal{L}_2$ are any lists whatever. How? By induction. The next paragraph covers the argument in detail. However, this kind of argument follows such a clear and frequent pattern that, in the rest of this tutorial, we simply say "by induction" and assume the point made.

Equation 9 is certainly true when $\mathcal{L}_1$ is a singleton, by the definition of c, or when $\mathcal{L}_1$ is the empty list, which is the identity of the list monoid, so that

$$\begin{aligned} c([]+\!\!+\mathcal{L}_2) &= c(\mathcal{L}_2) \\ &= 0 + c(\mathcal{L}_2) = c([]) + c(\mathcal{L}_2) \end{aligned}$$

The only remaining case is where $\mathcal{L}_1$ has more than one element. Write it as $[x]+\!\!+\mathcal{L}_0$, with $\mathcal{L}_0$ arbitrary non-empty.[4] The definition of c guarantees that $c(\mathcal{L}_1) = 1 + c(\mathcal{L}_0)$. Assume, hypothetically (for the sake of argument), that

$$c(\mathcal{L}_0+\!\!+\mathcal{L}_2) = c(\mathcal{L}_0) + c(\mathcal{L}_2)$$

Now consider

$$\mathcal{L}_1+\!\!+\mathcal{L}_2 = [x]+\!\!+\mathcal{L}_0+\!\!+\mathcal{L}_2$$

By associativity, which holds for any monoid operation, write

$$\begin{aligned} c([x]+\!\!+(\mathcal{L}_0+\!\!+\mathcal{L}_2)) &= 1 + c(\mathcal{L}_0+\!\!+\mathcal{L}_2) \\ &= 1 + c(\mathcal{L}_0) + c(\mathcal{L}_2) \\ &= c(\mathcal{L}_1) + c(\mathcal{L}_2) \\ = c(([x]+\!\!+\mathcal{L}_0)+\!\!+\mathcal{L}_2) &= c(\mathcal{L}_1+\!\!+\mathcal{L}_2) \end{aligned}$$

We have proved equation 9 for the empty and singleton cases, and we have proved that *if* it is true for a list of any length *then* it is true for a list of length one greater. Therefore it must be true for lists of any length. ■

Generalizing, for ϕ to be a transformation from $\mathcal{M}_1$ to $\mathcal{M}_2$ means that $\phi(a_1) = a_2$ and $\phi(b_1) = b_2$ are definite elements of $\mathcal{M}_2$ just when a_1 and b_1 are elements of $\mathcal{M}_1$ because the transformation is a function with a proviso. The proviso is 'preserve the monoid operations,' but what does that mean in general? It has to be that usage of $\oplus_1$ gets turned into usage of $\oplus_2$ roughly as follows:

$$\phi(a_1 \oplus_1 b_1) = \phi(a_1) \oplus_2 \phi(b_1) = a_2 \oplus_2 b_2 \tag{10}$$

but this causes a problem. Suppose $\mathcal{M}_1$ is an idempotent monoid, meaning that

$$a_1 \oplus_1 a_1 = a_1$$

Then, the attempt at preserving operations, equation 10, implies that

$$\phi(a_1 \oplus_1 a_1) = \phi(a_1) \oplus_2 \phi(a_1) = a_2 \oplus_2 a_2$$

but also that

$$\phi(a_1 \oplus_1 a_1) = \phi(a_1) = a_2$$

All is well if $\mathcal{M}_2$ is also idempotent, but, if not, then $a_2 \oplus_2 a_2 = a_2$ could be wrong, and this calculation could lead to a contradiction. In particular, this makes it impossible to count

[4] The argument is stronger than necessary: we could allow $\mathcal{L}_0$ to be empty, but it is perhaps more clear this way.

the number of members in a set with a monoid homomorphism:

$$\phi(\{a\} \cup \{a\}) = \phi(\{a\}) + \phi(\{a\}) = 2$$
$$\phi(\{a\} \cup \{a\}) = \phi(\{a\}) = 1$$

We must introduce an auxiliary binary operation of *difference* or *minus*. This operation is not allowed within the monoid, but prevents duplicate operations when mapping idempotent monoids to non-idempotent ones. Let us develop *minus* by comparison to set union. The monoid operation for sets is $\cup$ or *union*, not formally defined up to this point; let us do that here:

Definition 13. *If* A *and* B *are sets, then* $A \cup B$*, read "*A union B*," is the set of elements that appear either in* A *or* B*.*

So, $\{1,2\} \cup \{2,3\} = \{1,2,3\}$, for instance.

Now, given two sets, here is their difference

Definition 14. *If* A *and* B *are sets, then* $A - B$*, read "*A minus B*," is the set of elements that appear in* A *and not in* B*.*

So, $\{1,2\} - \{2,3\} = \{1\}$. This sort of definition works for the other idempotent monoid, permutations, because the order in which one takes elements out of a permutation does not matter.

$$(\!|1,2,3|\!) - (\!|3,2|\!) = (\!|1,2,3|\!) - (\!|2,3|\!)$$

Definition 15. *If* A *and* B *are permutations, then* $A - B$*, read "*A minus B*," is a permutation consisting of the elements in* A*, but not in* B*, without changing the order in* A*.*

We are now equipped, with *minus* defined for both kinds of idempotent monoids, to resolve the conundrum by writing

Definition 16. *A monoid homomorphism* $\phi : \mathcal{M}_1 \to \mathcal{M}_2$ *(definition 12)* ***preserves the monoid operations*** *through the following device:*

$$\phi(a \oplus_1 b) = \phi(a) \oplus_2 \psi(\phi, b, a) \tag{11}$$

where

$$\psi(\phi, b, a) = \begin{cases} \phi(b - a) & \text{if } \mathcal{M}_1 \text{ is idempotent} \\ \phi(b) & \text{otherwise} \end{cases} \tag{12}$$

■

Redoing the problematic computation of set length, it will no longer give out the pesky 2:

$$\begin{aligned}\phi(\{a\} \cup \{a\}) &= \phi(\{a\}) + \psi(\phi, \{a\}, \{a\}) \\ &= \phi(\{a\}) + \phi(\{a\} - \{a\}) \\ &= 1 + \phi(\emptyset) = 1 + 0 = 1\end{aligned}$$

We *could* have made this definition ever-so-slightly smaller by simply defining *minus* to be *no-operation* or *nop* for non-idempotent monoids. But the definition is more clear as stated above. Sometimes, clarity trades off against conciseness; usually they work together.

The similar problem of converting elements of a commutative (unordered) monoid into elements of a non-commutative (ordered) one is harder. Consider converting the set $1, 2, 3$ into a list. Can such a mapping be a homomorphism? Does it result in $[1, 2, 3]$ or $[2, 1, 3]$? What about the other four possible orders of 1, 2, and 3? Since, without further information, the mapping does not have a unique result, it is not a properly defined function — it is *non-deterministic*. So it fails the first criterion for being a homomorphism, that of being a function, yielding a single output for every input. In this case, it is better to solve the problem in the particular, when it arises, than to attempt a general definition. So, we conclude:

Observation 5. *Homomorphisms, in general, from commutative (unordered) collection monoids to non-commutative (ordered) collection monoids, do not exist.*

A natural question is whether the identity element of the source monoid always corresponds to the identity element of the destination monoid under any homomorphism. The answer is that it must, for, suppose it did not. Then,

$$\phi(\mathcal{Z}_1) = x_2 \neq \mathcal{Z}_2$$

would be some non-identity element. Now, pick some m_1 in $\mathcal{M}_1$ and consider the following calculation, which can lead to a contradiction:

$$\begin{aligned}m_1 \oplus_1 \mathcal{Z}_1 &= m_1 \\ \phi(m_1 \oplus_1 \mathcal{Z}_1) &= \phi(m_1) \stackrel{\text{def}}{=} m_2 \\ &= \phi(m_1) \oplus_2 \psi(\phi, \mathcal{Z}_1, m_1) \\ &= m_2 \oplus_2 x_2 \neq m_2 \text{ in general}\end{aligned}$$

Lemma 5. *A monoid homomorphism preserves the identity element.*

4.1. **A Zoo of Homomorphisms.** Consider the following idioms — *map*, *filter*, and *fold* — from functional programming and their relationships with monoid homomorphisms. Let $\mathcal{A}\langle\mathcal{T}\rangle = (a_1, a_2, \ldots)$ be a representative element of any of the four kinds of collection monoid, with a_i its constituent elements of type $\mathcal{T}$, $\mathcal{U}_\mathcal{A} = \mathcal{U}_{()}$ its unit function, $\oplus$ its monoid operation, and $\mathcal{Z}$ its identity. Write the type of $\mathcal{A}\langle\mathcal{T}\rangle$ itself as $(\mathcal{T})$.

4.1.1. *The* map *idiom.* *Map* is a function of two arguments. The first argument is another function, $\mathtt{f}$, that converts elements $\mathtt{t}$ of type $\mathcal{T}$ to elements, $\mathtt{s} = \mathtt{f}(\mathtt{t})$, of another type set, $\mathcal{S}$. The second argument of map is an element of a collection monoid, say $\mathcal{A}\langle\mathcal{T}\rangle$. The final output is an element of the same kind of collection monoid, but with a potentially different type set. Here is *map*, operating on $\mathcal{A}\langle\mathcal{T}\rangle$:

$$\mathtt{map}\ \mathtt{f}\ \mathcal{A}\langle\mathcal{T}\rangle = \mathtt{map}\ \mathtt{f}\ (a_1, a_2, \ldots) = (\mathtt{f}(a_1), \mathtt{f}(a_2), \ldots) = \mathcal{A}\langle\mathcal{S}\rangle$$

This does not change the kind of collection monoid to which $\mathcal{A}$ belongs, just the base type of the monoid's elements' elements, and, therefore, implicitly, the unit function that converts base-type elements to singletons in the monoid.

We might say that if we feed *map* its first argument only, that this combination, written $\mathtt{map}\ \mathtt{f}$, is a function from $\mathcal{A}\langle\mathcal{T}\rangle$ to $\mathcal{A}\langle\mathcal{S}\rangle$, *i.e.*, a function that converts elements of a collection monoid with base type $\mathcal{T}$ to elements of the same kind of collection monoid with base type switched out to $\mathcal{S}$. Write the type of this new combination function as follows:

$$\mathtt{map}\ \mathtt{f} : \mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle \to \mathcal{M}\langle\mathcal{S}, \oplus, \mathcal{Z}\rangle$$

There it is: the monoid homomorphism is $\mathtt{map}\ \mathtt{f}$. It *preserves* the monoid operation because it does not *change* the monoid operation. The identity element of the two collection monoids is the same because it pertains to the monoid rather than to the base sets, $\mathcal{T}$ and $\mathcal{S}$. The unit function also pertains to the monoid, so doesn't change in any meaningful way.

To derive a recursive form of the definition, we begin with a base case:

$$\mathtt{map}\ \mathtt{f}\ (\,) \stackrel{\text{def}}{=} (\,) \tag{13}$$

In other words, map has no effect on the empty collection. For the non-base case, as discussed at length concerning permutations, we apply a *deconstruction rule* to the monoid-element argument:

$$\begin{aligned}
\mathtt{map}\ \mathtt{f}\ \mathcal{A}\langle\mathcal{T}\rangle &= \mathtt{map}\ \mathtt{f}\ (a_1, a_2, \ldots) \\
&= \mathtt{map}\ \mathtt{f}\ \left((a_1) \oplus (a_2, \ldots)\right) \\
&= \mathtt{map}\ \mathtt{f}\ \left(\mathcal{U}(a_1) \oplus (a_2, \ldots)\right) \\
&= \mathtt{map}\ \mathtt{f}\ \left(\mathcal{U}\,(a_1) \oplus \mathcal{A}'\langle\mathcal{T}\rangle\right) \\
&\stackrel{\text{def}}{=} \mathcal{U}\ \left(\mathtt{f} \circ \mathcal{U}^{-1}\,(\mathcal{U}\,(a_1))\right) \oplus \left(\mathtt{map}\ \mathtt{f}\ \mathcal{A}'\langle\mathcal{T}\rangle\right) \qquad (14) \\
&\stackrel{\text{def}}{=} \mathcal{U}\,(\mathtt{f}\,(a_1)) \oplus \left(\mathtt{map}\ \mathtt{f}\ \mathcal{A}'\langle\mathcal{T}\rangle\right) \qquad (15)
\end{aligned}$$

where $\mathcal{A} = (a_1) \oplus \mathcal{A}'$. There is quite a bit of subtle 'junk' math going on, here, with the unit function, just to get consituent elements of $\mathcal{A}$ out of the collection monoid so $\mathtt{f}$ can operate on them, then subjected to the unit again so they can get $\oplus$'ed back on to the front of the recursive application of $\mathtt{map}\ \mathtt{f}$ to $\mathcal{A}'$. In the definition 14, the notation $\mathtt{f} \circ \mathcal{U}^{-1}$ means *function composition*, introduced at length below under the *fold* idiom. In this case, *map* composes $\mathtt{f}$ with the *inverse* of $\mathcal{U}$, written $\mathcal{U}^{-1}$. *Map* may need to know how

to *decondition* singletons with the inverse of the unit function, to take them out of their collection monoid. We know from the preliminary points of this section that not every function has an inverse that is, itself, a function. Fortunately, the unit functions of all the kinds of colletion monoids are invertible. Such is *not* the case for unit functions of non-collection monoids — those may have multi-valued inverses. It turns out that, when used under *map* as above, any value of a multi-valued inverse may be taken since the unit is immediately applied, a fact reflected in the second, alternative definition 15.

Keep these remarks in mind as we proceed through the *filter* and *fold* idioms, and as we endeavor to extend each of the idioms to non-collection monoids. Ultimately, these considerations will lead us to *monads* over *monoids*, since monads provide a clean, general technique for finessing away 'junk' math over the unit functions.

Here is an example of *map* at work in a Haskell-like programming language. Suppose a function, `intToChar`, which converts integers to their ASCII character equivalents. Now, suppose a list of integers, $[97, 98, 99]$. Convert the entire list of integers to their character equivalents as follows:

$$\texttt{map intToChar [97,98,99]} \rightarrow \texttt{['a','b','c']="abc"}$$

So, `map intToChar` will convert any list of integers — any element of the monoid of lists of integers — to a list of characters — an element of the monoid of lists of characters. The final equality obtains because functional programming languages usually represent character strings as lists of individual characters.

The definition of *map* also works for idempotent collection monoids like set. Suppose a function `nextPrime` that maps integers to the next higher prime number, and map it over a set of integers, for instance, $1..10 = \{1, 2, \ldots 10\}$ (in general, abbreviate the set of integers from $\mathfrak{m}$ to $\mathfrak{n}$ with double dots, as $\{\mathfrak{m}..\mathfrak{n}\}$):

$$\texttt{map nextPrime \{1..10\}} \rightarrow \texttt{\{2,3,5,7,11\}}$$

By this application, a set containing ten members becomes a set containing five members, because the duplicates get eliminated. In mathematics, elimination of duplicates is automatic, effortless, and almost magical. In real-world computation, we humans must do some work to program a computer to do some more work to get rid of them.

Back to the generalities, write the type of the function $\mathtt{f}$ as $\mathcal{T} \rightarrow \mathcal{S}$, and the type of *map*, itself as something that transforms a function of type $\mathcal{T} \rightarrow \mathcal{S}$ to something of type $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle \rightarrow \mathcal{M}\langle\mathcal{S}, \oplus, \mathcal{Z}\rangle$:

$$\texttt{map} : (\mathcal{T} \rightarrow \mathcal{S}) \rightarrow (\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle \rightarrow \mathcal{M}\langle\mathcal{S}, \oplus, \mathcal{Z}\rangle) \tag{16}$$

So, *map* is a *homomorphism-maker*: feed it a function that transforms the base types, and *map* gives you back a homomorphism amongst collection monoids.

Remark 1. *Generally, given a function of multiple arguments, the idea of giving it its arguments one at a time and interpreting the results as functions of the remaining arguments is* Currying. *A Curried function is a function of one argument that returns, recursively, a function of any remaining arguments. Any function can be Curried. To Curry a function of no arguments, just give it an argument to ignore. The similarity to the construction of collection monoids from singletons is intentional: Currying builds up multi-ary functions from unary functions, that is, from functions of single arguments.*

The *map* idiom, as understood so far, produces homomorphisms for collection monoids. Can it be given any meaning for non-collection monoids? Let $\mathcal{L}_{\mathbb{N}}$ be $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$ and $\mathcal{L}_{\mathbb{Z}}$ be $\mathcal{M}\langle\mathbb{Z}, +, 0\rangle$, and let f be a function from $\mathbb{N}$ to $\mathbb{Z}$ that, say, takes every input and returns its negative. So, $f(3) = -3$ and so on. f is properly formed to be an argument of *map* because f is a function from the base-type set of a monoid to another base-type set of the 'same kind of' monoid. What could be the meaning of `map f` in this case? How about a mapping from $\mathcal{L}_{\mathbb{N}}$ to $\mathcal{L}_{\mathbb{Z}}$ that preserves the operation $+$? In other words, as a monoid homomorphism? Viewed this way, map fulfills exactly definition 16. Specifically, it takes a function between two bare sets, namely f, and converts it into a homomorphism, namely `map f`, between two instances of the same kind of monoid. The 'kind' of monoid is specified by the monoid operation and the identity element, precisely what $\mathbb{N}$ and $\mathbb{Z}$ have in common in our example. In this case, the homomorphism is not 'onto' because it never produces a positive element of $\mathbb{Z}$. If we want an 'onto' homomorphism, then we could view the target as the monoid of negative integers, $\mathcal{L}_{-\mathbb{N}}$, which is certainly closed under addition and therefore a properly defined monoid.

Now, recall that *map* does some subtle juggling of the unit functions and their inverses. Let's look at a non-collection monoid like $\mathcal{M}\langle\mathbb{N}, +_{12}, 12\rangle$ whose unit function has a multi-valued, non-functional inverse. That unit function is *residual modulo 12*, and its inverse is multi-valued because, for instance, 1, 13, 25, ...are all preimages of 1. Therefore the 'inverse' of 1 does not have a single answer, so it's not a function. Now, consider an f like $\lambda x.x^2$, the function that just computes the square of its input. How could we compute, say, `map` $(\lambda x.x^2)$ $(7)_{12}$, where the monoid is the clock-arithmetic monoid? Well, since we immediately un-invert the inverse (see definitions 14 and 15), we can pick *any* of its values, because, in all cases, the answer is just $49 \bmod 12 = 1$.

All This means that we can promote the *map* idiom from its origin over lists in functional programming up to general collection monoids and further up to non-collection monoids.

4.1.2. *The* filter *idiom.* *Filter* is a function of two arguments: (1) a predicate that, for any element of the type set $\mathcal{T}$, returns a boolean $\in \{\text{true}, \text{false}\}$; and (2), an element of a collection monoid $\mathcal{A} = \mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$ of type $(\mathcal{T})$. It is defined as follows:

$$\texttt{filter}\ p\ \mathcal{A} = \bigoplus_{i=1}^{n} \begin{cases} \mathcal{U}(a_i) & \text{if } p(a_i) \\ \mathcal{Z} & \text{otherwise} \end{cases}$$

This just means "use $\oplus$ to combine either the singleton $\mathcal{U}(a_i)$ or the identity $\mathcal{Z}$ of the monoid into the overall result for every a_i in the input, depending on whether the predicate $p(a_i)$ evaluates to true." An alternative definition, in recursive style, equivalent in every way for collection monoids, is the following:

$$\texttt{filter}\ p\ (a_1, a_2, \ldots, a_n) = \begin{cases} \mathcal{U}(a_1) \oplus \texttt{filter}\ p\ (a_2, \ldots, a_n) & \text{if } p(a_i) \\ \texttt{filter}\ p\ (a_2, \ldots, a_n) & \text{otherwise} \end{cases}$$

Following in the footsteps of *map*, above, consider these type expressions:

$p : \mathcal{T} \to \{\text{true}, \text{false}\}$	*The type of* p *is 'function from* $\mathcal{T}$ *to the boolean constants'*
$\mathcal{A} : (\mathcal{T})$	*The type of* $\mathcal{A}$ *is* $(\mathcal{T})$ *or 'collection monoid with base type* $\mathcal{T}$*'*
$\texttt{filter}\ p\ \mathcal{A} : (\mathcal{T})$	*The type of* `filter` p $\mathcal{A}$ *is also* $\mathcal{T}$
$\texttt{filter}\ p : (\mathcal{T}) \to (\mathcal{T})$	*The type of* `filter` p *is* $(\mathcal{T}) \to (\mathcal{T})$*, or 'function from* $(\mathcal{T})$ *to* $(\mathcal{T})$*,' that is, a monoid homomorphism*
$\texttt{filter} : (\mathcal{T} \to \{\text{true}, \text{false}\}) \to (\mathcal{T}) \to (\mathcal{T})$	*The type of* `filter` *is 'function of a predicate that returns a monoid homomorphism'*

So `filter` p is a homomorphism from $(\mathcal{T})$ to $(\mathcal{T})$, and *filter* is another homomorphism-maker, this time, a function that converts *predicates* into monoid homomorphisms. Each resulting homomorphism changes none of the the base type, the operation, the unit, or the identity of the monoid; it just removes elements from elements of the source collection monoid that happen not to satisfy the predicate p.

So, how do the source and destination monoids differ? They may or may not. The most straightforward interpretation of homomorphism `filter` p is that it just takes an element of a monoid and produces another element of the same monoid. But, depending on one's point of view and details of the predicate, it also produces an element of another monoid, one in which elements of the base type — constituents elements of the elements of the monoid — are only those that satisfy p! Suppose p is *even?*, the predicate that returns *true* if its argument is an even element of $\mathbb{N}$ and *false* otherwise, and the source monoid is lists of natural numbers, $[\mathbb{N}]$. Then, `filter` p could be viewed either as a homomorphism to the distinct monoid of lists of even integers or as a homomorphism back to $[\mathbb{N}]$, just one that happens always to produce lists of even numbers.

This will work for predicate *odd?*, too, since a set of *collections* of odd numbers is a monoid. But, a set of odd numbers, under $+$, is not a monoid because it's not closed: the sum of two odd numbers is nod odd. So what sense can we make of *filter* for non-collection monoids? Once again, let $\mathcal{L}_{\mathbb{N}}$ be $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$. Recall that the unit function for this monoid is $\lambda x.x$, the identity function. Now, let p be *even?*. Looking at the development above, `filter` p could be a monoid homomorphism to the monoid of even natural numbers

under $+$ with 0 as identity element. That is a proper monoid, since the sum of any two even integers is an even integer. It would not work if the predicates were *odd?* or *prime?*, since the destinations would not be monoids (adding two primes does not necessarily produce a prime, and adding two odds never produces an odd). The lesson, here, is to be careful:

Observation 6. `filter` p *over non-collection monoids may produce outputs in a set that is not a monoid.*

The predicate can depend upon any attributes of its input. *Filter* creates a different homomorphism for each predicate. Imagine a predicate that can test an employee record for, say, green-card status:

$$\texttt{hasGreenCard} : \text{employeeRecord} \to \text{bool}$$

Then,

```
filter hasGreenCard
```

is a homomorphism that will take an element of any collection monoid of employee records and return an element of another collection monoid — of the same kind of collection — containing only those employees with green cards. This is just what the following SQL would do, presuming an attribute to store or method to compute the predicate:

```
SELECT e
FROM employees e
WHERE e.hasGreenCard
```

4.1.3. *The* fold *idiom.* So far, we have a way of building monoid homomorphisms that change the base type (`map` p) and a way of building monoid homomorphisms that change the members of the monoid (`filter` p) without changing the base type. This next idiom changes the kind of monoid but not the base type. Recall that $\mathcal{A}$ is an element of an arbitrary collection monoid, $\mathcal{M}\langle\mathcal{T}, \oplus, \mathcal{Z}\rangle$. Abbreviate the monoid as $\mathcal{M}_{\mathcal{A}}$. Now, suppose another monoid, not necessarily a collection monoid

$$\mathcal{M}_{\mathcal{B}} \stackrel{\text{def}}{=} \mathcal{M}\langle\mathcal{T}, \otimes, \mathcal{Y}\rangle$$

with unit function $\mathcal{U}_{\mathcal{B}} : \mathcal{T} \to \mathcal{M}_{\mathcal{B}}$. To make things easier to see, let

$$b_i = \mathcal{U}_{\mathcal{B}}(a_i) \tag{17}$$

The b_i are singletons in the new monoid, $\mathcal{M}_{\mathcal{B}}$, brought in one at a time from the constituent elements $a_i \in \mathcal{T}$ of $\mathcal{A} \in \mathcal{M}_{\mathcal{A}}$.

Fold is a function of three arguments: (1) the monoid operation, $\otimes$, of $\mathcal{M}_{\mathcal{B}}$; (2) its corresponding monoid identity element, $\mathcal{Y}$; and (3) an element $\mathcal{A}$ of a source collection monoid, $\mathcal{M}_{\mathcal{A}}$, potentially of a different kind from $\mathcal{M}_{\mathcal{B}}$. Given its first two arguments, *fold* converts

its last argument into an element of the kind of monoid implied by its first two arguments. Define

$$\texttt{fold} \otimes\ \mathcal{Y}\ \mathcal{A} \stackrel{\text{def}}{=} b_1 \otimes (b_2 \otimes (\ldots (b_n \otimes \mathcal{Y}) \ldots)) \tag{18}$$

or, more explicitly:

$$\texttt{fold} \otimes\ \mathcal{Y}\ \mathcal{A} \stackrel{\text{def}}{=} \mathcal{U}_{\mathcal{B}}(a_1) \otimes \left(\mathcal{U}_{\mathcal{B}}(a_2) \otimes \left(\ldots \left(\mathcal{U}_{\mathcal{B}}(a_n) \otimes \mathcal{Y}\right) \ldots\right)\right)$$

In recursive style, the base case is

$$\texttt{fold} \otimes\ \mathcal{Y}\ (\,) \stackrel{\text{def}}{=} \mathcal{Y} \tag{19}$$

and the rest is

$$\texttt{fold} \otimes\ \mathcal{Y}\ \mathcal{A} \stackrel{\text{def}}{=} \mathcal{U}_{\mathcal{B}}(a_1) \otimes \left(\texttt{fold} \otimes\ \mathcal{Y}\ \mathcal{A}'\right) \tag{20}$$

where $\mathcal{A} = (a_1) \oplus \mathcal{A}'$.

Notice that the unit function, $\mathcal{U}_{\mathcal{B}}$ is not one of the arguments of *fold*. As before, the unit is just implied by the other attributes of the destination monoid $\mathcal{M}\langle \mathcal{T}, \otimes, \mathcal{Y} \rangle$, in this case, by $\otimes$ and $\mathcal{Y}$.

So, $\texttt{fold} \otimes\ \mathcal{Y}$ — given just two of its three arguments, as we can always do because of Currying — is a function that converts any instance of an input *collection* monoid to an instance of another monoid, $\mathcal{M}\langle \mathcal{T}, \otimes, \mathcal{Y} \rangle$, not necessarily a collection monoid. In other words, $\texttt{fold} \otimes\ \mathcal{Y}$ is a monoid homomorphism. That means that *fold* must be another homomorphism-maker. Just as we did with *map* and *filter* above, let us write out *fold's* types:

$\texttt{fold} \otimes\ \mathcal{Y} : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$	*The type of* $\texttt{fold} \otimes\ \mathcal{Y}$ *is 'function from* $\mathcal{M}\langle \mathcal{T}, \oplus, \mathcal{Z} \rangle$ *to* $\mathcal{M}\langle \mathcal{T}, \otimes, \mathcal{Y} \rangle$*', or just* $\mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$ *for short; that is, a homomorphism from any collection monoid* $\mathcal{M}_{\mathcal{A}}$ *to a monoid* $\mathcal{M}_{\mathcal{B}}$
$\texttt{fold} \otimes : \mathcal{Y} \to \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$	*The type of* $\texttt{fold} \otimes$ *is 'function from an identity element,* $\mathcal{Y}$*, to a homorphism from any monoid* $\mathcal{M}_{\mathcal{A}}$ *to a monoid* $\mathcal{M}_{\mathcal{B}}$*'*
$\texttt{fold} : \otimes \to \mathcal{Y} \to \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$	*The type of* $\texttt{fold}$ *is 'function from an operation,* $\otimes$ *and a identity element,* $\mathcal{Y}$*, to a homorphism from any monoid* $\mathcal{M}_{\mathcal{A}}$ *to a monoid* $\mathcal{M}_{\mathcal{B}}$*'*

Fold lets us, quite prettily, specify the target monoid homomorphism in bits and pieces. Fully specify it by writing $\texttt{fold} \otimes\ \mathcal{Y}$; that expression is ready to take a monoid instance, $\mathcal{A} \in \mathcal{M}_{\mathcal{A}}$, and convert it into an instance $\mathcal{B}$ of $\mathcal{M}_{\mathcal{B}}$. Specify the target monoid's operation, but not its identity element, by writing $\texttt{fold} \otimes$. Give that expression an identity element to get a monoid homomorphism. Delay all commitments by writing just $\texttt{fold}$: to get a monoid homomorphism, supply both an operation and a identity element.

Fold is extremely versatile. Recall that this section started off with a homomorphism, c, from lists of elements, $\mathcal{M}\langle\mathcal{T}, +\!\!+, []\rangle$, to the monoid of natural numbers under addition, $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$. That homomorphism effects a count of the number of elements in the list. We defined c explicitly by its actions on the empty list and, recursively, its action on a list consisting of a singleton concatenated to another list. Can we get an equivalent homomorphism out of *fold*? Easily. First, map a function that converts anything to the constant 1 over the list, then apply `fold + 0` to the result. The result will be a member of $\mathcal{M}\langle\mathbb{N}, +, 0\rangle$.

Generalizing the above yields a prescription for *function composition*. In general, if g is a function from a set A to a set B, and f is a function from B to a set C, then the composition of f and g, written $f \circ g$, is the function from A to C defined by $f(g(a))$ for any a in A. Spelling it out:

$$\begin{aligned} g &: \mathcal{A} \to \mathcal{B} \\ f &: \mathcal{B} \to \mathcal{C} \\ f \circ g &: \mathcal{A} \to \mathcal{C} \end{aligned}$$

The function to map over the lists is $\lambda x.1$, and

$$g \stackrel{\text{def}}{=} \texttt{map}\ \lambda x.1 : [\mathcal{T}] \to [1]$$

is a homomorphism from a list monoid of elements of any type to the (unique) monoid of lists of copies of the natural number 1. This homomorphism preserves the number of elements in the list, as would any map over a list, but changes every input element into the number 1. Now, let f be `fold + 0`, the homomorphism that takes any collection of natural numbers and sums them. The composition

$$c = f \circ g = (\texttt{fold}\ +\ 0) \circ (\texttt{map}\ \lambda x.1)$$

will be the monoid homomorphism that counts the elements in any list.

What meaning can *fold* have if its third argument is not a collection monoid? Note that $\mathcal{A}$, an element of the monoid $\mathcal{M}_{\mathcal{A}}$, will have a constituent element a such that $\mathcal{A} = \mathcal{U}_{\mathcal{A}}(a)$. The definition of *fold* in equation 18 now leads us to write

$$\texttt{fold} \otimes\ \mathcal{Y}\ \mathcal{A} \stackrel{\text{def}}{=} \mathcal{U}_{\mathcal{B}}(a) = \mathcal{U}_{\mathcal{B}}\left(\mathcal{U}_{\mathcal{A}}^{-1}(\mathcal{A})\right) \tag{21}$$

So, *fold*ing over a non-collection monoid effectively just brings the input argument, the instance $\mathcal{A} = \mathcal{U}_{\mathcal{A}}(a)$ of monoid $\mathcal{M}_{\mathcal{A}}$, into monoid $\mathcal{M}_{\mathcal{B}}$ *via* the unit function $\mathcal{U}_{\mathcal{B}}$. To do that, it must first 'back out' a from $\mathcal{A}$ by inverting $\mathcal{A}$'s unit function. The operation $\otimes$ and identity element $\mathcal{Y}$ in $\mathcal{M}_{\mathcal{B}}$ do not play a role when the domain monoid, $\mathcal{M}_{\mathcal{A}}$ is not a collection.

It is somewhat disquieting that we cannot write $\texttt{fold} \otimes \mathcal{Y} = \mathcal{U}_{\mathcal{B}} \circ \mathcal{U}_{\mathcal{A}}^{-1}$ because $\texttt{fold} \otimes \mathcal{Y}$ does 'know' enough about $\mathcal{A}$ to reference $\mathcal{U}_{\mathcal{A}}^{-1}$.

This gives us a requirement on unit functions: the composition $\mathcal{U}_{\mathcal{B}} \circ \mathcal{U}_{\mathcal{A}}^{-1}$ must be a function. In some specific cases, the composition $\mathcal{U}_{\mathcal{B}} \circ \mathcal{U}_{\mathcal{A}}^{-1}$ might be a function even if $\mathcal{U}_{\mathcal{A}}$ is not invertible. But, for it to be generally useful, we must be able to delay all commitments with *fold*, which means

Observation 7. Fold *operates only over monoids with invertible unit functions.*

5. From Monoids to Monads

The unit functions first arose from the constructions of lists as a monoid. We needed to say "lists of *what*?" So, we needed to specify the base-type set, $\mathcal{T}$, then promote elements of that set into the monoid as singletons. The reason was to allow us a recursive definition of *list*, which we could extend to the other kinds of collection monoid. All was well until we started building homomorphisms from the *map*, *filter*, and *fold* idioms, wherefrom the unit function leapt up again with a vengeance to complicate matters.

The way out is the *monad*, a derivative construct to the *monoid*, that dramatically simplifies and unifies homomorphisms. Monads can be understood in a couple of different lights. A formal development requires some category theory, out of scope for this tutorial, but see, for instance, chapter 10 of [AM75]. In practical terms, though, the difference is small, more a case of taking an opportunity for conciseness than anything else. But it is delicate and opens up the whole vista of optimization in the forms of the *fusion law* and *cheap deforestation*, also known as *acid rain*. For benefit, we can reduce the three idioms to one, the monadic version of fold, by writing the others in its terms. Such is not possible in the monoidal forms due to the implacable implicitness of the unit, as we demonstrate in the next section.

5.1. Writing *map* as a *fold*?

All three homomorphism-makers iterate, in a sense, over their input monoids. This leads us to the question whether they can be written in terms of one another. The most obvious first thing to try is to write *map* as a *fold*. To keep it concrete, assume `i2c = intToChar`, as we mapped it before over `[97,98,99]`:

$$\begin{aligned}\texttt{map i2c [97,98,99]} \to \texttt{['a','b','c']} &= \texttt{"abc"} \\ &= \texttt{['a']++['b']++['c']}\end{aligned}$$

How can we write this as a monoidal fold?

$$\texttt{fold} \otimes\ y\ \mathcal{A} = \mathcal{U}_{\mathcal{B}}(a_1) \otimes \left(\mathcal{U}_{\mathcal{B}}(a_2) \otimes \left(\ldots \left(\mathcal{U}_{\mathcal{B}}(a_n) \otimes y\right) \ldots\right)\right)$$

The first thing to do is rewrite this in prefix notation so that its Curried forms are visually apparent. Instead of $x \otimes y$, write $\otimes\ x\ y$, and

$$\texttt{fold} \otimes\ y\ \mathcal{A} = \otimes\ \mathcal{U}_{\mathcal{B}}(a_1) \left(\otimes\ \mathcal{U}_{\mathcal{B}}(a_2) \left(\ldots \left(\otimes\ \mathcal{U}_{\mathcal{B}}(a_n)\ y\right) \ldots\right)\right)$$

What we want is

$$+\!\!+\ [f(a_1)]\ \left(+\!\!+\ [f(a_2)]\ \left(\ldots \left(+\!\!+\ [f(a_n)]\ []\right) \ldots\right)\right)$$

Since the destination collection monoid is still a list, $\mathcal{U}_{\mathcal{B}}$ must be $\mathcal{U}_{[]}$, but we really want it to be

```
xyz
```

5.2. **Real-world programming.** To cross the stream without category theory, it's best to work in a real-world, time-tested programming language, Haskell.

Let's look at how real programming languages approach the *map*, *filter*, and *fold* idioms. Real programming languages do not (unfortunately) have greek letters and mathematical symbols like $\circ$. Fortunately, the designers of Haskell have addressed the lack very concisely [has], so we may write the homomorphism c as

```
(fold (+) 0) . (map \_ -> 1)
```

and only three points of translation need addressing:

(1) In mathematical notation, we write $+$ for the binary function of two natural numbers that adds them. Haskell treats an unadorned $+$ as an infix operator, but can convert any infix operator into the normal prefix form of a function by wrapping it in parentheses. So, `(+)` is a Haskell function of type

```
(+) :: (Num a) => a -> a -> a
```

which means, given that type `a` is any of the numerical types, then `(+)` is of type function from `a` to function of `a` to `a`, or, equivalently, because of Currying, function of two `a`'s to one `a`.

(2) Haskell uses backslash, `\`, to denote λ; underscore, `_`, to denote a variable whose value is not used; and `->` to mean `.` in the context of a lambda abstraction. So $\lambda x.1$ is `\x -> 1`, the same as `\_ -> 1` since `x` is not used.

(3) Haskell uses `.` for function composition.

We will find Haskell to be genuinely useful in crossing the bridge from concepts to action. It directly supports the monoid concept in the form of a *monad*. The difference can be understood in a couple of different lights. A formal development requires some category theory, out of scope for this tutorial, but see, for instance, chapter 10 of [AM75]. In practical terms, though, the difference is small, non-critical, and more a case of taking an opportunity for conciseness than anything else. But it is delicate.

Let's review the monoidal way of building lists. Let x be an element of the base-type set $\mathcal{T}$, and $[x]$ be a singleton element of the monoid of lists $\mathcal{M}\langle\mathcal{T}, +\!\!+, []\rangle$. Then, we have

$$[x] = \mathcal{U}_{[]}(x)$$

The easiest way to understand the difference is with a side-by-side development.

`[x]++xs  (x:xs)`

Bind takes an instance of a monad, a function that converts the constituent element of an instance of a monad of that type

`bind` $[1,2,3]\ \mathcal{U}_{\mathcal{B}} \circ \mathcal{U}_{\mathcal{A}}^{-1}$

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

6. Fold: the Mother of All

This is essentially a crib of [Hut99].

6.1. **The Universal Property.** Suppose a function, g, that satisfies the following two equations:

$$\begin{array}{lcl} g\ [\,] & = & z \\ g\ (x : xs) & = & f\ x\ (g\ xs) \end{array}$$

$$\begin{array}{lcl} g\ [\,] & = & v \\ g\ (x : xs) & = & f\ x\ (gxs) \end{array} \quad \Leftrightarrow \quad g = \mathsf{fold}\ f\ v$$

6.2. **The Fusion Property.**

6.3. **Maps and Filters as Folds.**

7. MONOID CALCULUS

Let A_{list} be a **list** of arbitrary items, that is, an ordered collection, with duplicates allowed:

$$A_{list} = [a_1, \ldots, a_n]$$

(22) $$\hom[+\!\!+, \cup](f)A_{list} \to \{f(a_1), \ldots, f(a_n)\}$$

result $\leftarrow\{\}$;
foreach x **in** A_{list} **do**
  result $\leftarrow$result $\cup f(x)$;
return result;

8. APPENDIX A

Notation	Description
$x \in C$	x is an **element** or **element** of collection C
$\forall x \in C \ldots$	For every element x of collection C ...
$\exists x \in C \ldots$	There exists an element x in collection C...
$x \oplus y$	Combining x and y with a generalized operator, $\oplus$, similar to the familiar $x + y$ notation
$f : X \to Y$	The function f takes or maps elements of the set X to elements of the set Y
$f(x) = E$	The function f, given the input value x, produces the output value E
$\lambda x.E$	This entire expression denotes the anonymous function that takes an x and 'returns' E
$x : \mathcal{T}$	x is of type $\mathcal{T}$, or *has* type $\mathcal{T}$ (**type assertion**)
$\Gamma \vdash x : \mathcal{T}$	In the context of Γ, which is a comma-delimited sequence of type assertions, x has type $\mathcal{T}$
$P \vee Q, P \wedge Q$	P or Q (inclusive or); P and Q
$\neg P$	Not P; P is false (logical negation; very tight precedence binding)
$\frac{P}{Q}$ or $P \Rightarrow Q$	If P is true, then Q is true; same as $\neg Q \vee P$ (**implication**)
$\{a, b, \ldots\}$	A **set** of elements: unordered; duplicates not allowed
$[a, b, \ldots]$	A **sequence** or **list** of elements: ordered; duplicates allowed
$[a, b]$	An **ordered pair**; special case of list
$\wr a, b, \ldots \int$	A **bag** or **multiset** of elements: unordered; duplicates allowed
$(\!\| a, b, \ldots \|\!)$	A **permutation** of elements: ordered; duplicates not allowed
$\{\mathcal{T}\}, [\mathcal{T}], \wr \mathcal{T} \int, (\!\| \mathcal{T} \|\!)$	The type of (sets, lists, bags, permutations) of instances of type $\mathcal{T}$

REFERENCES

[A22] Sloan sequence a000522. http://www.research.att.com/projects/OEIS?Anum=A000522.

[A26] Sloan sequence a007526. http://www.research.att.com/projects/OEIS?Anum=A007526.

[A40] Sloan sequence a033540. http://www.research.att.com/projects/OEIS?Anum=A033540.

[AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.

[bab] The library of babel. http://en.wikipedia.org/wiki/The_Library_of_Babel.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.

[Fok94] Maarten M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Technical Report 94-28, 1994.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[Gru03] T. Grust. Monad comprehensions: A versatile representation for queries. In *The Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data*, chapter 10. Springer Verlag, 2003. ISBN 3-540-00375-4.

[has] Haskell language web site. http://www.haskell.org.

[HM96] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, 1996.

[Hut99] Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[MJ95] Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *Tutorial Text 1st Int. Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, 24–30 May 1995*, volume 925, pages 228–266. Springer-Verlag, Berlin, 1995.

[oei] Online encyclopedia of integer sequences. http://www.research.att.com/ njas/sequences/.

[PJ02] Simon Peyton-Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction, Marktoberdorf Summer School*, 2002.

[Wad90] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice*, pages 61–78, New York, NY, 1990. ACM.

[Wad92] Philip Wadler. The essence of functional programming. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Albequerque, New Mexico, 1992.

[Wad93] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag, 1993.