中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-19-002

Equational reasoning for non-determinism monad: the case of Spark aggregation

Shin-Cheng Mu

June 11, 2019 || Technical Report No. TR-IIS-19-002
http://www.iis.sinica.edu.tw/page/library/TechReport/tr2019/tr19.html

Equational Reasoning for Non-determinism Monad:

The Case of Spark Aggregation

SHIN-CHENG MU, Academia Sinica, Taiwan

As part of the author's studies on equational reasoning for monadic programs, this report focus on non-determinism monad. We discuss what properties this monad should satisfy, what additional operators and notations can be introduced to facilitate equational reasoning about non-determinism, and put them to the test by proving a number of properties in our example problem inspired by the author's previous work on proving properties of Spark aggregation.

1 INTRODUCTION

In functional programming, *pure* programs are those that can be understood as static mappings from inputs to outputs. The main advantage of staying in the pure realm is that properties of pure entities can be proved by equational reasoning. Side effects, in contrast, used to be considered the "awkward squad" that are difficult to be reasoned about. Gibbons and Hinze [2011], however, showed that effectful, monadic programs may also be reasoned about in a mathematical manner, using monad laws and properties of effect operators.

This report is part of a series of the author's studies on equational reasoning for monadic programs. In this report we focus on non-determinism monad — in our definition that is a monad having two effect operators, one allowing a program to fail, another allowing a non-deterministic choice between two results. We discuss what properties these operators should satisfy, what additional operators and notations can be introduced to facilitate equational reasoning of this monad, and put them to the test by proving a number of properties in our example problem: Spark aggregation.

Much of this report is inspired by the author's joint work with Chen et al. [2017], in which we formalised Spark, a platform for distributed computation, and derived properties under which a distributed Spark aggregation represents a *deterministic* computation. Therefore, many examples in this report are about finding out when processing a non-deterministic permutation (simulating arbitrary distribution of data) produces a deterministic result.

2 MONAD AND NON-DETERMINISM

A monad consists of a type constructor $\mathsf{M} :: * \to *$ and two operators $\mathit{return} :: a \to \mathsf{M}\ a$ and "bind" $(\mathrel{=\!\!<\!\!<}) :: (a \to \mathsf{M}\ b) \to \mathsf{M}\ a \to \mathsf{M}\ b$ that satisfy the following *monad laws*:

$$f \mathrel{=\!\!<\!\!<} \mathit{return}\ x = f\ x\ , \tag{1}$$

$$\mathit{return} \mathrel{=\!\!<\!\!<} m = m\ , \tag{2}$$

$$f \mathrel{=\!\!<\!\!<} (g \mathrel{=\!\!<\!\!<} m) = (\lambda x \to f \mathrel{=\!\!<\!\!<} g\ x) \mathrel{=\!\!<\!\!<} m\ . \tag{3}$$

Rather than the usual $(\mathrel{>\!\!>\!\!=}) :: \mathsf{M}\ a \to (a \to \mathsf{M}\ b) \to \mathsf{M}\ b$, in the laws above we use the reversed bind $(\mathrel{=\!\!<\!\!<})$, which is consistent with the direction of function composition and more readable when we program in a style that uses composition. When we use bind with λ-abstractions, it is more natural to write $m \mathrel{>\!\!>\!\!=} \lambda x \to f\ x$. In this report we use the former more than the latter, thus the choice of notation. We also define $m_1 \ll m_2 = \mathit{const}\ m_1 \mathrel{=\!\!<\!\!<} m_2$. Note that $(\gg)$ has type $\mathsf{M}\ a \to \mathsf{M}\ b \to \mathsf{M}\ b$.

More operators we find useful are given in Figure 1. Right-to-left Kleisli composition, denoted by $(\mathrel{<\!\!=\!\!<})$, composes two monadic operations $a \to \mathsf{M}\ b$ and $b \to \mathsf{M}\ c$ into an operation $a \to \mathsf{M}\ c$. Operators $(\langle\$\rangle)$ and $(\langle\bullet\rangle)$ are monadic counterparts of function application and composition: $(\langle\$\rangle)$ applies a pure function to a monad, while $(\langle\bullet\rangle)$ composes a pure function after a monadic function.

Author's address: Shin-Cheng Mu, Institute of Information Science, Academia Sinica, Taiwan, scm@iis.sinica.edu.tw.

$$
\begin{aligned}
&(\mathbin{<\!\!=\!\!<}) :: (b \to \mathsf{M}\ c) \to (a \to \mathsf{M}\ b) \to a \to \mathsf{M}\ c\\
&(f \mathbin{<\!\!=\!\!<} g)\ x = f \mathbin{=\!\!<\!\!<} g\ x\\
&(\langle\$\rangle) :: (a \to b) \to \mathsf{M}\ a \to \mathsf{M}\ b\\
&f \mathbin{\langle\$\rangle} m = (\mathit{return} \cdot f) \mathbin{=\!\!<\!\!<} m\\
&(\langle\bullet\rangle) :: (b \to c) \to (a \to \mathsf{M}\ b) \to (a \to \mathsf{M}\ c)\\
&f \mathbin{\langle\bullet\rangle} g = (\mathit{return} \cdot f) \mathbin{<\!\!=\!\!<} g
\end{aligned}
$$

Fig. 1. Some monadic operators we find handy for this paper.

We now introduce a collections of properties that allows us to rotate an expression that involves two operators and three operands. These properties will be handy when we need to move parenthesis around in expressions. To begin with, the following properties show that $(\langle\$\rangle)$ and $(\langle\bullet\rangle)$ share properties similar to pure function application and composition:

$$(f \mathbin{\langle\bullet\rangle} g)\ x = f \mathbin{\langle\$\rangle} g\ x\ , \tag{4}$$

$$f \mathbin{\langle\$\rangle} (g \mathbin{\langle\$\rangle} m) = (f \cdot g) \mathbin{\langle\$\rangle} m\ , \tag{5}$$

$$f \mathbin{\langle\bullet\rangle} (g \mathbin{\langle\bullet\rangle} h) = (f \cdot g) \mathbin{\langle\bullet\rangle} h\ . \tag{6}$$

We also have the following law that allows us to rotate an expression that uses $(\langle\bullet\rangle)$ and $(\cdot)$:

$$f \mathbin{\langle\bullet\rangle} (g \cdot h) = (f \mathbin{\langle\bullet\rangle} g) \cdot h\ . \tag{7}$$

Note that g in (7) must be a function returning a monad. Furthermore, (8) and (9) relate $(\mathbin{=\!\!<\!\!<})$ and $(\langle\$\rangle)$, both operators applying functions to monads, while (10) and (11) relate $(\mathbin{<\!\!=\!\!<})$ and $(\langle\bullet\rangle)$, both operators composing functions on monads:

$$f \mathbin{=\!\!<\!\!<} (g \mathbin{\langle\$\rangle} m) = (f \cdot g) \mathbin{=\!\!<\!\!<} m\ , \tag{8}$$

$$f \mathbin{\langle\$\rangle} (g \mathbin{=\!\!<\!\!<} m) = (f \mathbin{\langle\bullet\rangle} g) \mathbin{=\!\!<\!\!<} m\ , \tag{9}$$

$$f \mathbin{<\!\!=\!\!<} (g \mathbin{\langle\bullet\rangle} h) = (f \cdot g) \mathbin{<\!\!=\!\!<} h\ , \tag{10}$$

$$f \mathbin{\langle\bullet\rangle} (g \mathbin{<\!\!=\!\!<} h) = (f \mathbin{\langle\bullet\rangle} g) \mathbin{<\!\!=\!\!<} h\ . \tag{11}$$

Having these properties is one of the advantages of writing $(\mathbin{=\!\!<\!\!<})$ and $(\mathbin{<\!\!=\!\!<})$ backwards. All the properties above can be proved by expanding definitions, and it is a good warming-up exercise proving some of them. Some of them are proved in Appendix A.

None of these operators and properties are strictly necessary: they can all be reduced to return, $(\mathbin{=\!\!<\!\!<})$, and λ-abstractions. As is often the case when designing notations, having more operators allows ideas to be expressed concisely in a higher level of abstraction, at the expense of having more properties to memorise. It is personal preference where the balance should be. Properties (4) through (11) may look like a lot of properties to remember. In practice, we find it usually sufficient to let us be guided by types. For example, when we have $f \mathbin{\langle\$\rangle} g\ x$ and want to bring f and g together, by their types we can figure out the resulting expression should be $(f \mathbin{\langle\bullet\rangle} g)\ x$.

Non-determinism Monad. Non-determinism is the only effect we use in this report. We assume two operators $\emptyset$ and $([\!])$: the former denotes failure, while $m \mathbin{[\!]} n$ denotes that the computation may yield either m or n. As pointed out by Gibbons and Hinze [2011], for proofs and derivations, what matters is not how a monad is implemented but what properties its operators satisfy. What laws $\emptyset$ and $([\!])$ should satisfy, however, can be a tricky issue. As discussed by Kiselyov [2015], it eventually comes down to what we use the monad for. It is usually expected that $(a, ([\!]), \emptyset)$ be a monoid. That is, $(\mathbin{[\!]})$ is associative, with $\emptyset$ as its zero:

$$\begin{aligned}(m \mathbin{[\!]} n) \mathbin{[\!]} k &= m \mathbin{[\!]} (n \mathbin{[\!]} k)\,,\\ \emptyset \mathbin{[\!]} m = m &= m \mathbin{[\!]} \emptyset\,.\end{aligned}$$

It is also assumed that monadic bind distributes into $(\mathbin{[\!]})$ from the end, while $\emptyset$ is a right zero for $(=\!\!\ll)$:

$$f =\!\!\ll (m_1 \mathbin{[\!]} m_2) = (f =\!\!\ll m_1) \mathbin{[\!]} (f =\!\!\ll m_2)\,, \tag{12}$$

$$f =\!\!\ll \emptyset = \emptyset\,. \tag{13}$$

For our purpose in this section, we also assume that $(\mathbin{[\!]})$ is commutative ($m \mathbin{[\!]} n = n \mathbin{[\!]} m$) and idempotent ($m \mathbin{[\!]} m = m$). Implementation of such non-determinism monads have been studied by Fischer et al. [2011].

Here are some induced laws about how $(\langle\$\rangle)$ interacts with return and non-determinism operators:

$$f \mathbin{\langle\$\rangle} \mathit{return}\ x = \mathit{return}\ (f\ x)\,, \tag{14}$$

$$f \mathbin{\langle\$\rangle} \emptyset = \emptyset\,, \tag{15}$$

$$f \mathbin{\langle\$\rangle} (m_1 \mathbin{[\!]} m_2) = (f \mathbin{\langle\$\rangle} m_1) \mathbin{[\!]} (f \mathbin{\langle\$\rangle} m_2)\,. \tag{16}$$

3 PERMUTATION AND INSERTION

As a warm-up example, the function perm non-deterministically computes a permutation of its input, using an auxiliary function insert that inserts an element to an arbitrary position in a list:

```
perm             :: [a] → M [a]
perm []          = return []
perm (x : xs)    = insert x =≪ perm xs ,

insert           :: a → [a] → M [a]
insert x []      = return [x]
insert x (y : xs) = return (x : y : xs) [] ((y:) ⟨$⟩ insert x xs) .
```

For example, possible results of $\mathit{perm}\ [0,1,2]$ include $[0,1,2]$, $[0,2,1]$, $[1,0,2]$, $[1,2,0]$, $[2,0,1]$, and $[2,1,0]$.

Determinism. The following lemma presents properties under which permuting the input list does not change the result of a foldr:

Lemma 3.1. *Given* $(\odot) :: a \to b \to b$. *If* $x \odot (y \odot z) = y \odot (x \odot z)$ *for all* $x, y :: a$ *and* $z :: b$, *we have*

$$\mathit{foldr}\ (\odot)\ z \mathbin{\langle\bullet\rangle} \mathit{perm} = \mathit{return} \cdot \mathit{foldr}\ (\odot)\ z\,.$$

Since perm is defined in terms of insert, proof of Lemma 3.1 naturally depends on a lemma about a related property of insert:

Lemma 3.2. *Given* $(\odot) :: a \to b \to b$, *we have*

$$\mathit{foldr}\ (\odot)\ z \mathbin{\langle\bullet\rangle} \mathit{insert}\ x = \mathit{return} \cdot \mathit{foldr}\ (\odot)\ z \cdot (x{:})\,,$$

provided that $x \odot (y \odot z) = y \odot (x \odot z)$ *for all* $x, y :: a$ *and* $z :: b$.

Proof. Prove $\mathit{foldr}\ (\odot)\ z \mathbin{\langle\$\rangle} \mathit{insert}\ x\ xs = \mathit{return}\ (\mathit{foldr}\ (\odot)\ z\ (x : xs))$. Induction on xs.

Case $xs := [\,]$:

$$\begin{aligned}&\mathit{foldr}\ (\odot)\ z \mathbin{\langle\$\rangle} \mathit{insert}\ x\ [\,]\\ =\ &\quad \{\text{definition of } \mathit{insert}\}\end{aligned}$$

$$\begin{aligned}
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{return}~[x] \\
={}& \quad \{\, \text{by (14)} \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~[x])~.
\end{aligned}$$

Case $xs := y : xs$:

$$\begin{aligned}
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{insert}~x~(y : xs) \\
={}& \quad \{\, \text{definition of } \mathit{insert} \,\} \\
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} (\mathit{return}~(x : y : xs) \mathbin{[\!]} ((y{:}) \mathbin{\langle\$\rangle} \mathit{insert}~x~xs)) \\
={}& \quad \{\, \text{by (16), (14), and (5)} \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : y : xs)) \mathbin{[\!]} ((\mathit{foldr}~(\odot)~z \cdot (y{:})) \mathbin{\langle\$\rangle} \mathit{insert}~x~xs)~.
\end{aligned}$$

Focus on the second branch of $(\mathbin{[\!]})$:

$$\begin{aligned}
& (\mathit{foldr}~(\odot)~z \cdot (y{:})) \mathbin{\langle\$\rangle} \mathit{insert}~x~xs \\
={}& \quad \{\, \text{definition of } \mathit{foldr} \,\} \\
& ((y\odot) \cdot \mathit{foldr}~(\odot)~z) \mathbin{\langle\$\rangle} \mathit{insert}~x~xs \\
={}& \quad \{\, \text{by (5)} \,\} \\
& (y\odot) \mathbin{\langle\$\rangle} (\mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{insert}~x~xs) \\
={}& \quad \{\, \text{induction} \,\} \\
& (y\odot) \mathbin{\langle\$\rangle} \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : xs)) \\
={}& \quad \{\, \text{by (14)} \,\} \\
& \mathit{return}~(y \odot \mathit{foldr}~(\odot)~z~(x : xs)) \\
={}& \quad \{\, \text{definition of } \mathit{foldr} \,\} \\
& \mathit{return}~(y \odot (x \odot \mathit{foldr}~(\odot)~z~xs)) \\
={}& \quad \{\, \text{since } x \odot (y \odot z) = y \odot (x \odot z) \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : y : xs))~.
\end{aligned}$$

Thus we have

$$\begin{aligned}
& (\mathit{foldr}~(\odot)~z \mathbin{\langle\bullet\rangle} \mathit{insert}~x)~(y : xs) \\
={}& \quad \{\, \text{calculation above} \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : y : xs)) \mathbin{[\!]} \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : y : xs)) \\
={}& \quad \{\, \text{idempotence of } (\mathbin{[\!]}) \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~(x : y : xs))~.
\end{aligned}$$

□

Proof of Lemma 3.1 then follows:

Proof. Prove that $\mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{perm}~xs = \mathit{return}~(\mathit{foldr}~(\odot)~z~xs)$. Induction on xs.
Case $xs := [\,]$:

$$\begin{aligned}
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{perm}~[\,] \\
={}& \quad \{\, \text{definitions of } \mathit{perm} \,\} \\
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{return}~[\,] \\
={}& \quad \{\, \text{by (14)} \,\} \\
& \mathit{return}~(\mathit{foldr}~(\odot)~z~[\,])~.
\end{aligned}$$

Case $xs := x : xs$:

$$\begin{aligned}
& \mathit{foldr}~(\odot)~z \mathbin{\langle\$\rangle} \mathit{perm}~(x : xs) \\
={}& \quad \{\, \text{definition of } \mathit{perm} \,\}
\end{aligned}$$

$$
\begin{aligned}
&\mathit{foldr}\;(\odot)\;z \mathbin{\langle\$\rangle} (\mathit{insert}\;x \mathbin{=\!\ll} \mathit{perm}\;\mathit{xs})\\
=&\quad \{\text{ by (9) }\}\\
&(\mathit{foldr}\;(\odot)\;z \mathbin{\langle\bullet\rangle} \mathit{insert}\;x) \mathbin{=\!\ll} \mathit{perm}\;\mathit{xs}\\
=&\quad \{\text{ Lemma 3.2 }\}\\
&(\mathit{return} \cdot \mathit{foldr}\;(\odot)\;z \cdot (x{:})) \mathbin{=\!\ll} \mathit{perm}\;\mathit{xs}\\
=&\quad \{\text{ definitions of } \mathit{foldr} \text{ and } (\langle\$\rangle) \}\\
&((x\odot) \cdot \mathit{foldr}\;(\odot)\;z) \mathbin{\langle\$\rangle} \mathit{perm}\;\mathit{xs}\\
=&\quad \{\text{ by (5) }\}\\
&(x\odot) \mathbin{\langle\$\rangle} (\mathit{foldr}\;(\odot)\;z \mathbin{\langle\$\rangle} \mathit{perm}\;\mathit{xs})\\
=&\quad \{\text{ induction }\}\\
&(x\odot) \mathbin{\langle\$\rangle} (\mathit{return}\;(\mathit{foldr}\;(\odot)\;z\;\mathit{xs}))\\
=&\quad \{\text{ by (14) }\}\\
&\mathit{return}\;(x \odot \mathit{foldr}\;(\odot)\;z\;\mathit{xs})\\
=&\quad \{\text{ definition of } \mathit{foldr} \}\\
&\mathit{return}\;(\mathit{foldr}\;(\odot)\;z\;(x:\mathit{xs}))\;.
\end{aligned}
$$

□

Map, Filter, and Permutation. It is not hard for one to formulate the following relationship between map and perm, which is also based on a related property relating map and insert:[1]

LEMMA 3.3. $\mathit{perm} \cdot \mathit{map}\;f = \mathit{map}\;f \mathbin{\langle\bullet\rangle} \mathit{perm}$.

LEMMA 3.4. $\mathit{insert}\;(f\;x) \cdot \mathit{map}\;f = \mathit{map}\;f \mathbin{\langle\bullet\rangle} \mathit{insert}\;x$.

The lemma is true because $\mathit{map}\;f$ is a pure computation — in reasoning about monadic programs it is helpful, and sometimes essential, to identify its pure segments, because these are the parts more properties are applicable. Note that the composition $(\cdot)$ on the lefthand side is turned into $(\langle\bullet\rangle)$ once we move $\mathit{map}\;f$ leftwards.

We prove only Lemma 3.4.

PROOF. Prove by induction on xs that $\mathit{map}\;f \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs} = \mathit{insert}\;(f\;x)\;(\mathit{map}\;f\;\mathit{xs})$ for all xs. We present only the inductive case $\mathit{xs} := y : \mathit{xs}$:

$$
\begin{aligned}
&\mathit{map}\;f \mathbin{\langle\$\rangle} \mathit{insert}\;x\;(y:\mathit{xs})\\
=&\quad \{\text{ definition of } \mathit{insert} \}\\
&\mathit{map}\;f \mathbin{\langle\$\rangle} (\mathit{return}\;(x:y:\mathit{xs}) \mathbin{[\!]} ((y{:}) \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs}))\\
=&\quad \{\text{ by (16) and (14) }\}\\
&\mathit{return}\;(\mathit{map}\;f\;(x:y:\mathit{xs})) \mathbin{[\!]} (\mathit{map}\;f \mathbin{\langle\$\rangle} ((y{:}) \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs}))\;.
\end{aligned}
$$

For the second branch we reason:

$$
\begin{aligned}
&\mathit{map}\;f \mathbin{\langle\$\rangle} ((y{:}) \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs})\\
=&\quad \{\text{ by (5) }\}\\
&(\mathit{map}\;f \cdot (y{:})) \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs}\\
=&\quad \{\text{ definition of } \mathit{map} \}\\
&((f\;y{:}) \cdot \mathit{map}\;f) \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs}\\
=&\quad \{\text{ by (5) }\}\\
&(f\;y{:}) \mathbin{\langle\$\rangle} (\mathit{map}\;f \mathbin{\langle\$\rangle} \mathit{insert}\;x\;\mathit{xs})
\end{aligned}
$$

[1] Lemma 3.3 and 3.4 are in fact free theorems of perm and insert [Voigtländer 2009]. They serve as good exercises, nevertheless.

$$\begin{aligned} &= \quad \{\, \text{induction} \,\} \\ &\quad (f\ y{:}) \mathbin{\langle\$\rangle} \mathit{insert}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs})\ . \end{aligned}$$

Thus we have

$$\begin{aligned} &\quad \mathit{map}\ f \mathbin{\langle\$\rangle} \mathit{insert}\ x\ (y : \mathit{xs}) \\ &= \quad \{\, \text{calculation above} \,\} \\ &\quad \mathit{return}\ (f\ x : f\ y : \mathit{map}\ f\ \mathit{xs}) \mathbin{[\!]} ((f\ y{:}) \mathbin{\langle\$\rangle} (\mathit{insert}\ (f\ x)\ (\mathit{map}\ f\ \mathit{xs}))) \\ &= \quad \{\, \text{definitions of } \mathit{insert} \text{ and } \mathit{map} \,\} \\ &\quad \mathit{insert}\ (f\ x)\ (\mathit{map}\ f\ (y : \mathit{xs}))\ . \end{aligned}$$

□

One may have noticed that the style of proof is familiar: replace $\mathit{return}\ x$ by $[x]$ and $(\mathbin{[\!]})$ by $(+\!\!+)$, the proof is more-or-less what one would do for a list version of *insert*. This is exactly the point: the style of proofs we use to do for pure programs still works for monadic programs, as long as the monad satisfies the demanded laws, be it a list, a more advanced implementation of non-determinism, or a monad having other effects.

A similar property relating *perm* and *filter* can be formulated.

Lemma 3.5. *$\mathit{perm} \cdot \mathit{filter}\ p = \mathit{filter}\ p \mathbin{\langle\bullet\rangle} \mathit{perm}$.*

Its proof is routine and omitted. Finally, in a number of occasions it helps to know that *xs* is a result of *perm xs*. The proof is also routine and omitted.

Lemma 3.6. *For all xs we have that $\mathit{perm}\ \mathit{xs} = \mathit{return}\ \mathit{xs} \mathbin{[\!]} m$ for some m.*

4 SPARK AGGREGATION

Spark [Zaharia et al. 2012] is a popular platform for scalable distributed data-parallel computation based on a flexible programming environment with high-level APIs, considered by many as the successor of MapReduce. In a typical Spark program, data is partitioned and stored distributively on read-only *Resilient Distributed Datasets* (RDDs) — we can think of it as a list of lists, where each sub-list is potentially stored on a remote node. On an RDD one can apply operations, called *combinators*, such as *map*, *reduce*, and *aggregate*. The *aggregate* combinator, for example, takes user-defined functions $(\otimes)$ and $(\oplus)$: $(\otimes)$ accumulates a sub-result for each data partition while $(\oplus)$ merges sub-results across different partitions.

Programming in Spark, however, can be tricky. Since sub-results are computed across partitions concurrently, the order of their applications varies on different executions. Aggregation in Spark is therefore inherently non-deterministic. An example from Chen et al. [2017] showed that computing the integral of x^{73} from $x = -2$ to $x = 2$, which should be 0, using a function in the Spark machine learning library, yields results ranging from -8192.0 to 12288.0 in different runs. It is thus desirable to find out conditions, which Spark's documentation does not specify formally, under which a Spark computation yields deterministic outcomes.

4.1 List Homomorphism

Since a Spark aggregation is typically used to computes a *list homomorphism* [Bird 1987], we digress a little in this section to give a brief review and present some results that we will use. A function $h :: \mathsf{List}\ a \to b$ is called a list homomorphism if there exists $z :: b$, $k :: a \to b$, and $(\oplus) :: b \to b \to b$ such that:

$$\begin{aligned} h\,[\,] &= z \\ h\,[x] &= k\,x \\ h\,(xs \mathbin{+\!\!+} ys) &= h\,xs \oplus h\,ys\;. \end{aligned}$$

That h is such a list homomorphism is denoted by $h = hom\ (\oplus)\ k\ z$. Note that the properties above implicitly demand that $(\oplus)$ be associative with z as its identity element.

Lemma 4.1 and 4.2 below are about when a computation defined in terms of $foldr$ is actually a list homomorphism. In Lemma 4.2, $img\ f$ denotes the image of a function f.

Lemma 4.1. *$h = hom\ (\oplus)\ (h \cdot wrap)\ z$ if and only if $foldr\ (\oplus)\ z \cdot map\ h = h \cdot concat$, where $wrap\ x = [x]$.*

Lemma 4.2. *Let $(\oplus) :: b \rightarrow b \rightarrow b$ be associative on $img\ (foldr\ (\otimes)\ z)$ with z as its identity, where $(\otimes) :: a \rightarrow b \rightarrow b$. We have $foldr\ (\otimes)\ z = hom\ (\oplus)\ (\otimes z)\ z$ if and only if $x \otimes (y \oplus w) = (x \otimes y) \oplus w$ for all $x :: a$ and $y, w \in img\ (foldr\ (\otimes)\ z)$.*

Notice, in Lemma 4.2, that $(\otimes z) = foldr\ (\otimes)\ z \cdot wrap$. Proofs of both lemmas are interesting exercises, albeit being a bit off-topic. They are recorded in Appendix A.

4.2 Formalisation and Results

Distributed collections of data are represented by *Resilient Distributed Datasets* (RDDs) in Spark. Informally, an RDD is a collection of data entries; these data entries are further divided into partitions stored on different machines. Abstractly, an RDD can be seen as a list of lists:

$$\begin{aligned} &\textbf{type}\ \mathsf{Partition}\ a = [a]\,, \\ &\textbf{type}\ \mathsf{RDD}\ a \quad\;\; = [\mathsf{Partition}\ a]\,, \end{aligned}$$

where each Partition may be stored in a different machine.

While Spark provides a collection of *combinators* (functions on RDDs that are designed to be composed to form larger programs), in this report we focus on a particular one, $aggregate$. It can be seen as a parallel implementation $foldr$. The combinator processes an RDD in two levels: each partition is first processed locally on one machine by $foldr\ (\otimes)\ z$. The sub-results are then communicated and combined — this second step can be think of as another $foldr$ with $(\oplus)$.[2]

Spark programmers like to assume that their programs are deterministic. To exploit concurrency, however, the sub-results from each machine might be processed in arbitrary order and the result could be non-deterministic. The following is our characterisation of $aggregate$, where we use $perm$ to model the fact that sub-results from each machine are processed in unknown order:

$$\begin{aligned} &aggregate :: b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow (b \rightarrow b \rightarrow b) \rightarrow \mathsf{RDD}\ a \rightarrow \mathsf{M}\ b \\ &aggregate\ z\ (\otimes)\ (\oplus) = foldr\ (\oplus)\ z \mathbin{\langle\$\rangle} (perm \cdot map\ (foldr\ (\otimes)\ z))\;. \end{aligned}$$

It is clear from the types that $foldr\ (\otimes)\ z$ and $foldr\ (\oplus)\ z$ are pure computations, and non-determinism is introduced solely by $perm$.

Deterministic Aggregation. We are interested in finding out conditions under which $aggregate$ produces deterministic outcomes.

Theorem 4.3. *Given $(\otimes) :: a \rightarrow b \rightarrow b$ and $(\oplus) :: b \rightarrow b \rightarrow b$, where $(\oplus)$ is associative and commutative, we have:*

$$aggregate\ z\ (\otimes)\ (\oplus) = return \cdot foldr\ (\oplus)\ z \cdot map\ (foldr\ (\otimes)\ z)\;.$$

[2] In fact, the actual Spark aggregation (and that modelled in Chen et al. [2017]) are like $foldl$. For convenience in our proofs we see all list operations the other way round and use $foldr$. This is not a fundamental difference.

Proof. We reason:

$$\begin{aligned}
& \mathit{aggregate}\ z\ (\otimes)\ (\oplus) \\
= \quad & \{\ \text{definition of } \mathit{aggregate}\ \} \\
& \mathit{foldr}\ (\oplus)\ z \mathbin{\langle\bullet\rangle} (\mathit{perm} \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)) \\
= \quad & \{\ \text{by (7)}\ \} \\
& (\mathit{foldr}\ (\oplus)\ z \mathbin{\langle\bullet\rangle} \mathit{perm}) \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z) \\
= \quad & \{\ \text{Lemma 3.1, since } (\oplus) \text{ is associative and commutative}\ \} \\
& \mathit{return} \cdot \mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ .
\end{aligned}$$

□

The following corollary summaries the results and present conditions under which *aggregate* computes a homomorphism.

Corollary 4.4. *$\mathit{aggregate}\ z\ (\otimes)\ (\oplus) = \mathit{return} \cdot \mathit{hom}\ (\oplus)\ (\otimes z)\ z \cdot \mathit{concat}$, provided that $(\oplus)$ is associative, commutative, and has z as identity, and that $x \otimes (y \oplus w) = (x \otimes y) \oplus w$ for all $x :: a$ and $y, w \in \mathit{img}\ (\mathit{foldr}\ (\otimes)\ z)$.*

Proof. We reason:

$$\begin{aligned}
& \mathit{aggregate}\ z\ (\otimes)\ (\oplus) \\
= \quad & \{\ \text{Theorem 4.3}\ \} \\
& \mathit{return} \cdot \mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z) \\
= \quad & \{\ \mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z \text{ by Lemma 4.2; Lemma 4.1}\ \} \\
& \mathit{return} \cdot \mathit{hom}\ (\oplus)\ (\otimes z)\ z \cdot \mathit{concat}\ .
\end{aligned}$$

□

Determinism Implies Homomorphism. The final part of the report deals with an opposite question: what can we infer if we know that *aggregate* is deterministic? To answer that, however, we need to assume two more properties:

$$m_1 \mathbin{[\!]} m_2 = \mathit{return}\ x \implies m_1 = m_2 = \mathit{return}\ x. \tag{17}$$

$$\mathit{return}\ x_1 = \mathit{return}\ x_2 \implies x_1 = x_2. \tag{18}$$

Property (17) can be seen as the other direction of idempotency of $(\mathbin{[\!]})$, while (18) states that *return* is injective.

The following lemma can be understood this way: when $\mathit{aggregate}\ z\ (\otimes)\ (\oplus)$, which could be non-deterministic, can be performed by a deterministic function, the operator $(\oplus)$ should be insensitive to ordering:

Lemma 4.5. *If $\mathit{aggregate}\ z\ (\otimes)\ (\oplus) = \mathit{return} \cdot \mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}$, and $\mathit{perm}\ \mathit{xss} = \mathit{return}\ \mathit{yss} \mathbin{[\!]} m$ for some m, we have*

$$\begin{aligned}
& \mathit{foldr}\ (\otimes)\ z\ (\mathit{concat}\ \mathit{xss}) = \\
& \quad \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ \mathit{xss}) = \\
& \qquad \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ \mathit{yss})\ .
\end{aligned}$$

Proof. We reason:

$$\begin{aligned}
& \mathit{return} \cdot \mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}\ \$\ \mathit{xss} \\
= \quad & \{\ \text{assumption}\ \} \\
& \mathit{aggregate}\ z\ (\otimes)\ (\oplus)\ \$\ \mathit{xss}
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{\ \text{definition of } \mathit{aggregate}\text{, Lemma 3.3, and (6)}\ \} \\
&\quad (\mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z))\ \langle\$\rangle\ \mathit{perm}\ \mathit{xss} \\
&= \quad \{\ \text{assumption: } \mathit{perm}\ \mathit{xss} = \mathit{return}\ \mathit{yss} \mathbin{[\!]} m\text{, by (16) and (14)}\ \} \\
&\quad (\mathit{return} \cdot \mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ \$\ \mathit{yss}) \mathbin{[\!]} \\
&\qquad ((\mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z))\ \langle\$\rangle\ m)\ .
\end{aligned}
$$

Thus by (17) and (18), $\mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}\ \$\ \mathit{xss}$ equals $\mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ \$\ \mathit{yss}$. The former also equals $\mathit{foldr}\ (\oplus)\ z \cdot \mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ \$\ \mathit{xss}$ because, by Lemma 3.6, $\mathit{perm}\ \mathit{xss} = \mathit{return}\ \mathit{xss} \mathbin{[\!]} m$ for some m. □

Based on Lemma 4.5, the following theorem explicitly states that $(\oplus)$ should be associative, commutative, and has z as its identity in restricted domain.

Theorem 4.6. *If $\mathit{aggregate}\ z\ (\otimes)\ (\oplus) = \mathit{return} \cdot \mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}$, we have that $(\oplus)$, when restricted to values in $\mathit{img}\ (\mathit{foldr}\ (\otimes)\ z)$, is associative, commutative, and has z as its identity.*

Proof. In the discussion below, let x,y, and w be in $\mathit{img}\ (\mathit{foldr}\ (\otimes)\ z)$. That is, there exists xs, ys, and ws such that $x = \mathit{foldr}\ (\otimes)\ z\ \mathit{xs}$, $y = \mathit{foldr}\ (\otimes)\ z\ \mathit{ys}$, and $w = \mathit{foldr}\ (\otimes)\ z\ \mathit{ws}$.

Identity. We reason:

$$
\begin{aligned}
&\quad y \\
&= \mathit{foldr}\ (\otimes)\ z\ (\mathit{concat}\ [\mathit{xs}]) \\
&= \quad \{\ \mathit{perm}\ [\mathit{xs}] = \mathit{return}\ [\mathit{xs}] \mathbin{[\!]} \emptyset\text{, Lemma 4.5}\ \} \\
&\quad \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{xs}]) \\
&= y \oplus z\ .
\end{aligned}
$$

Thus z is a right identity of $(\oplus)$. Similarly,

$$
\begin{aligned}
&\quad y \\
&= \mathit{foldr}\ (\otimes)\ z\ (\mathit{concat}\ [[\,], \mathit{xs}]) \\
&= \quad \{\ \mathit{perm}\ [[\,], \mathit{xs}] = \mathit{return}\ [[\,], \mathit{xs}] \mathbin{[\!]} m\text{, Lemma 4.5}\ \} \\
&\quad \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [[\,], \mathit{xs}]) \\
&= z \oplus (y \oplus z) \\
&= \quad \{\ z \text{ is a right identity of } (\oplus)\ \} \\
&\quad z \oplus y\ .
\end{aligned}
$$

Thus z is also a left identity of $(\oplus)$.

Commutativity. We reason:

$$
\begin{aligned}
&\quad x \oplus y \\
&= \quad \{\ z \text{ is a right identity}\ \} \\
&\quad x \oplus (y \oplus z) \\
&= \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{xs}, \mathit{ys}]) \\
&= \quad \{\ \mathit{perm}\ [\mathit{xs}, \mathit{ys}] = \mathit{return}\ [\mathit{ys}, \mathit{xs}] \mathbin{[\!]} m\text{, Lemma 4.5}\ \} \\
&\quad \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{ys}, \mathit{xs}]) \\
&= y \oplus (x \oplus z) \\
&= \quad \{\ z \text{ is a right identity}\ \} \\
&\quad y \oplus x\ .
\end{aligned}
$$

Associativity. We reason:

$$
\begin{aligned}
&\quad x \oplus (y \oplus w) \\
&= \quad \{\ z \text{ is a right identity}\ \}
\end{aligned}
$$

$$\begin{aligned}
& x \oplus (y \oplus (w \oplus z)) \\
= \; & \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{xs}, \mathit{ys}, \mathit{ws}]) \\
= \; & \quad \{\ (\oplus) \text{ commutative}\ \} \\
& \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{ws}, \mathit{xs}, \mathit{ys}]) \\
= \; & w \oplus (x \oplus (y \oplus z)) \\
= \; & \quad \{\ z \text{ is a right identity}\ \} \\
& w \oplus (x \oplus y) \\
= \; & \quad \{\ (\oplus) \text{ commutative}\ \} \\
& (x \oplus y) \oplus w\ .
\end{aligned}$$

□

THEOREM 4.7. *If* $\mathit{aggregate}\ z\ (\otimes)\ (\oplus) = \mathit{return} \cdot \mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}$, *we have* $\mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z$.

PROOF. Apparently $\mathit{foldr}\ (\otimes)\ z\ [\,] = z$ and $\mathit{foldr}\ (\otimes)\ z\ [x] = x \otimes z$. We are left with proving the case for concatenation.

$$\begin{aligned}
& \mathit{foldr}\ (\otimes)\ z\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) \\
= \; & \mathit{foldr}\ (\otimes)\ z\ (\mathit{concat}\ [\mathit{xs}, \mathit{ys}]) \\
= \; & \quad \{\ \text{Lemma 4.5}\ \} \\
& \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ (\mathit{foldr}\ (\otimes)\ z)\ [\mathit{xs}, \mathit{ys}]) \\
= \; & \mathit{foldr}\ (\otimes)\ z\ \mathit{xs} \oplus (\mathit{foldr}\ (\otimes)\ z\ \mathit{ys} \oplus z) \\
= \; & \quad \{\ \text{Theorem 4.6, } z \text{ is identity}\ \} \\
& \mathit{foldr}\ (\otimes)\ z\ \mathit{xs} \oplus \mathit{foldr}\ (\otimes)\ z\ \mathit{ys}\ .
\end{aligned}$$

□

COROLLARY 4.8. *Given* $(\otimes) :: a \to b \to b$ *and* $(\oplus) :: b \to b \to b$. $\mathit{aggregate}\ z\ (\otimes)\ (\oplus) = \mathit{return} \cdot \mathit{foldr}\ (\otimes)\ z \cdot \mathit{concat}$ *if and only if* $(\mathit{img}\ (\mathit{foldr}\ (\otimes)\ z), (\oplus), z)$ *forms a commutative monoid, and that* $\mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z$.

PROOF. A conclusion following from Theorem 4.3, Theorem 4.6, and Theorem 4.7. □

Acknowledgements. In around late 2016, Yu-Fang Chen, Chih-Duo Hong, Ondřej Lengál, Nishant Sinha and Bow-Yaw Wang invited me into their project formalising Spark. It was what inspired my interests in reasoning about monads, which led to a number of subsequent work. The initial proofs of properties of *aggregate* and other combinators were done by Ondřej Lengál, without using monads. Affeldt et al. [2019] modelled a hierarchy of monadic effects in Coq. The formalisation was applied to verify a number of equational proofs of monadic programs, including some of the proofs in an earlier version of this report. I am solely responsible for any remaining errors, however.

REFERENCES

Reynald Affeldt, David Nowak, and Takafumi Saikawa. 2019. A hierarchy of monadic effects for program verification using equational reasoning. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Springer.

Richard S. Bird. 1987. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, Manfred Broy (Ed.). Number 36 in NATO ASI Series F. Springer-Verlag, 3–42.

Yu-Fang Chen, Chih-Duo Hong, Ondřej Lengál, Shin-Cheng Mu, Nishant Sinha, and Bow-Yaw Wang. 2017. An executable sequential specification for Spark aggregation. In *International Conference on Networked Systems*. Springer-Verlag.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely functional lazy nondeterministic programming. *Journal of Functional Programming* 21, 4-5 (September 2011), 413–465.

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *International Conference on Functional Programming*, Olivier Danvy (Ed.). ACM Press, 2–14.

Oleg Kiselyov. 2015. Laws of MonadPlus. http://okmij.org/ftp/Computation/monads.html#monadplus.

Janis Voigtländer. 2009. Free theorems involving type constructor classes. In *International Conference on Functional Programming*, Andrew Tolmach (Ed.). ACM Press, 173–184.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Networked Systems Design and Implementation*, Steven Gribble and Dina Katabi (Eds.). USENIX.

A MISCELLANEOUS PROOFS

Proving (8). $f =\!\!\ll (g \mathbin{\langle\$\rangle} m) = (f \cdot g) =\!\!\ll m$.

Proof. We reason:

$$\begin{array}{ll}
& f =\!\!\ll (g \mathbin{\langle\$\rangle} m) \\
= & \quad \{\, \text{definition of } (\langle\$\rangle) \,\} \\
& f =\!\!\ll ((\mathit{return} \cdot g) =\!\!\ll m) \\
= & \quad \{\, \text{monad law (3)} \,\} \\
& (\lambda x \rightarrow f =\!\!\ll \mathit{return}\ (g\ x)) =\!\!\ll m \\
= & \quad \{\, \text{monad law (1)} \,\} \\
& (\lambda x \rightarrow f\ (g\ x)) =\!\!\ll m \\
= & (f \cdot g) =\!\!\ll m\ .
\end{array}$$

□

Proving (5). $f \mathbin{\langle\$\rangle} (g \mathbin{\langle\$\rangle} m) = (f \cdot g) \mathbin{\langle\$\rangle} m$.

Proof. We reason:

$$\begin{array}{ll}
& f \mathbin{\langle\$\rangle} (g \mathbin{\langle\$\rangle} m) \\
= & \quad \{\, \text{definition of } (\langle\$\rangle) \,\} \\
& (\mathit{return} \cdot f) =\!\!\ll (g \mathbin{\langle\$\rangle} m) \\
= & \quad \{\, \text{by (8)} \,\} \\
& (\mathit{return} \cdot f \cdot g) =\!\!\ll m \\
= & \quad \{\, \text{definition of } (\langle\$\rangle) \,\} \\
& (f \cdot g) \mathbin{\langle\$\rangle} m\ .
\end{array}$$

□

For the next results we prove a lemma:

$$(f{=\!\!\ll}) \cdot (g{=\!\!\ll}) = (((f{=\!\!\ll}) \cdot g){=\!\!\ll})\ . \tag{19}$$

$$\begin{array}{ll}
& (f{=\!\!\ll}) \cdot (g{=\!\!\ll}) \\
= & \quad \{\, \eta \text{ intro.} \,\} \\
& (\lambda m \rightarrow f =\!\!\ll (g =\!\!\ll m)) \\
= & \quad \{\, \text{monad law (3)} \,\} \\
& (\lambda m \rightarrow (\lambda y \rightarrow f =\!\!\ll g\ y) =\!\!\ll m) \\
= & \quad \{\, \eta \text{ reduction} \,\} \\
& (((f{=\!\!\ll}) \cdot g){=\!\!\ll})\ .
\end{array}$$

Proving (6). $f \mathbin{\langle\bullet\rangle} (g \mathbin{\langle\bullet\rangle} m) = (f \cdot g) \mathbin{\langle\bullet\rangle} m$.

Proof. We reason:

$$\begin{aligned}
& f \mathbin{\langle\bullet\rangle} (g \mathbin{\langle\bullet\rangle} m) \\
= \quad & \{\, \text{definition of } (\langle\bullet\rangle) \,\} \\
& ((\mathit{return} \cdot f)\mathbin{=\!\!\ll}) \cdot ((\mathit{return} \cdot g)\mathbin{=\!\!\ll}) \cdot m \\
= \quad & \{\, \text{by (19)} \,\} \\
& ((((\mathit{return} \cdot f)\mathbin{=\!\!\ll}) \cdot \mathit{return} \cdot g)\mathbin{=\!\!\ll}) \cdot m \\
= \quad & \{\, \text{monad law (1)} \,\} \\
& ((\mathit{return} \cdot f \cdot g)\mathbin{=\!\!\ll}) \cdot m \\
= \quad & \{\, \text{definition of } (\langle\bullet\rangle) \,\} \\
& (f \cdot g) \mathbin{\langle\bullet\rangle} m \;.
\end{aligned}$$

□

Proving (10). $f \mathbin{<\!\!=\!\!<} (g \mathbin{\langle\bullet\rangle} h) = (f \cdot g) \mathbin{<\!\!=\!\!<} h$.

Proof. We reason:

$$\begin{aligned}
& f \mathbin{<\!\!=\!\!<} (g \mathbin{\langle\bullet\rangle} h) \\
= \quad & \{\, \text{definitions of } (<\!\!=\!\!<) \,\} \\
& (f\mathbin{=\!\!\ll}) \cdot ((\mathit{return} \cdot g)\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{by (19)} \,\} \\
& (((f\mathbin{=\!\!\ll}) \cdot \mathit{return} \cdot g)\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{monad law (1)} \,\} \\
& ((f \cdot g)\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{definition of } (<\!\!=\!\!<) \,\} \\
& (f \cdot g) \mathbin{<\!\!=\!\!<} h \;.
\end{aligned}$$

□

Proving (11). $f \mathbin{\langle\bullet\rangle} (g \mathbin{<\!\!=\!\!<} h) = (f \mathbin{\langle\bullet\rangle} g) \mathbin{<\!\!=\!\!<} h$.

Proof. We reason:

$$\begin{aligned}
& f \mathbin{\langle\bullet\rangle} (g \mathbin{<\!\!=\!\!<} h) \\
= \quad & \{\, \text{definitions of } (<\!\!=\!\!<) \text{ and } (\langle\bullet\rangle) \,\} \\
& ((\mathit{return} \cdot f)\mathbin{=\!\!\ll}) \cdot (g\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{by (19)} \,\} \\
& ((((\mathit{return} \cdot f)\mathbin{=\!\!\ll}) \cdot g)\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{definition of } (\langle\bullet\rangle) \,\} \\
& ((f \mathbin{\langle\bullet\rangle} g)\mathbin{=\!\!\ll}) \cdot h \\
= \quad & \{\, \text{definition of } (<\!\!=\!\!<) \,\} \\
& (f \mathbin{\langle\bullet\rangle} g) \mathbin{<\!\!=\!\!<} h \;.
\end{aligned}$$

□

Proof of Lemma 4.1.

Proof. A Ping-pong proof.

Direction ($\Rightarrow$). Let $h = \mathit{hom}\ (\oplus)\ (h \cdot \mathit{wrap})\ z$, prove $\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ \mathit{xss}) = h\ (\mathit{concat}\ \mathit{xss})$ by induction on xss.

Case $\mathit{xss} := [\,]$:

$$\begin{aligned}
&\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ [\,])\\
=\ &\mathit{foldr}\ (\oplus)\ z\ [\,]\\
=\ &z\\
=\ &h\ (\mathit{concat}\ [\,])\ .
\end{aligned}$$

Case $\mathit{xss} := \mathit{xs} : \mathit{xss}$:

$$\begin{aligned}
&\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ (\mathit{xs} : \mathit{xss}))\\
=\ &h\ \mathit{xs} \oplus \mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ \mathit{xss})\\
=\ &\quad \{\ \text{induction}\ \}\\
&h\ \mathit{xs} \oplus h\ (\mathit{concat}\ \mathit{xss})\\
=\ &\quad \{\ h\ \text{homomorphism}\ \}\\
&h\ (\mathit{concat}\ (\mathit{xs} : \mathit{xss}))\ .
\end{aligned}$$

Direction ($\Leftarrow$). Assuming $\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ \mathit{xss}) = h\ (\mathit{concat}\ \mathit{xss})$, prove that $h = \mathit{hom}\ (\oplus)\ (h \cdot \mathit{wrap})\ z$.

Case empty list:

$$\begin{aligned}
&h\ [\,]\\
=\ &h\ (\mathit{concat}\ [\,])\\
=\ &\quad \{\ \text{assumption}\ \}\\
&\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ [\,])\\
=\ &z\ .
\end{aligned}$$

Case singleton list: certainly $h\ [x] = h\ [x]$.

Case concatentation:

$$\begin{aligned}
&h\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys})\\
=\ &h\ (\mathit{concat}\ [\mathit{xs}, \mathit{ys}])\\
=\ &\quad \{\ \text{assumption}\ \}\\
&\mathit{foldr}\ (\oplus)\ z\ (\mathit{map}\ h\ [\mathit{xs}, \mathit{ys}])\\
=\ &h\ \mathit{xs} \oplus (h\ \mathit{ys} \oplus z)\\
=\ &h\ \mathit{xs} \oplus h\ \mathit{ys}\ .
\end{aligned}$$

□

Proof of Lemma 4.2.

Proof. A Ping-pong proof.

Direction ($\Leftarrow$). We show that $\mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z$, provided that $x \otimes (y \oplus w) = (x \otimes y) \oplus w$.

It is immediate that $\mathit{foldr}\ (\otimes)\ z\ [\,] = z$ around $\mathit{foldr}\ (\otimes)\ z\ [x] = x \otimes z$. For $\mathit{xs} \mathbin{+\!\!+} \mathit{ys}$, note that

$$\mathit{foldr}\ (\otimes)\ z\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) = \mathit{foldr}\ (\otimes)\ (\mathit{foldr}\ (\otimes)\ z\ \mathit{ys})\ \mathit{xs}\ .$$

The aim is thus to prove that

$$\mathit{foldr}\ (\otimes)\ (\mathit{foldr}\ (\otimes)\ z\ \mathit{ys})\ \mathit{xs} = (\mathit{foldr}\ (\otimes)\ z\ \mathit{xs}) \oplus (\mathit{foldr}\ (\otimes)\ z\ \mathit{ys})\ .$$

We perform an induction on xs. The case when $xs := [\,]$ trivially holds. For $xs := x : xs$, we reason:

$$\begin{aligned}
&\quad \mathit{foldr}\ (\otimes)\ (\mathit{foldr}\ (\otimes)\ z\ ys)\ (x : xs)\\
&= x \otimes \mathit{foldr}\ (\otimes)\ (\mathit{foldr}\ (\otimes)\ z\ ys)\ xs\\
&= \quad \{\ \text{induction}\ \}\\
&\quad x \otimes ((\mathit{foldr}\ (\otimes)\ z\ xs) \oplus (\mathit{foldr}\ (\otimes)\ z\ ys))\\
&= \quad \{\ \text{assumption: } x \otimes (y \oplus w) = (x \otimes y) \oplus w\ \}\\
&\quad (x \otimes (\mathit{foldr}\ (\otimes)\ z\ xs)) \oplus (\mathit{foldr}\ (\otimes)\ z\ ys)\\
&= (\mathit{foldr}\ (\otimes)\ z\ (x : xs)) \oplus (\mathit{foldr}\ (\otimes)\ z\ ys)\ .
\end{aligned}$$

Direction ($\Rightarrow$). Given $\mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z$, prove that $x \otimes (y \oplus w) = (x \otimes y) \oplus w$ for y and w in the range of $\mathit{foldr}\ (\otimes)\ z$.

Let $y = \mathit{foldr}\ (\otimes)\ z\ xs$ and $w = \mathit{foldr}\ (\otimes)\ z\ ys$ for some xs and ys. We reason:

$$\begin{aligned}
&\quad x \otimes (y \oplus w)\\
&= x \otimes (\mathit{foldr}\ (\otimes)\ z\ xs \oplus \mathit{foldr}\ (\otimes)\ z\ ys)\\
&= \quad \{\ \text{since } \mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z\ \}\\
&\quad x \otimes (\mathit{foldr}\ (\otimes)\ z\ (xs \mathbin{+\!\!+} ys))\\
&= \mathit{foldr}\ (\otimes)\ z\ (x : xs \mathbin{+\!\!+} ys)\\
&= \quad \{\ \text{since } \mathit{foldr}\ (\otimes)\ z = \mathit{hom}\ (\oplus)\ (\otimes z)\ z\ \}\\
&\quad \mathit{foldr}\ (\otimes)\ z\ (x : xs) \oplus \mathit{foldr}\ (\otimes)\ z\ ys\\
&= (x \otimes \mathit{foldr}\ (\otimes)\ z\ xs) \oplus \mathit{foldr}\ (\otimes)\ z\ ys\\
&= (x \otimes y) \oplus w\ .
\end{aligned}$$

□