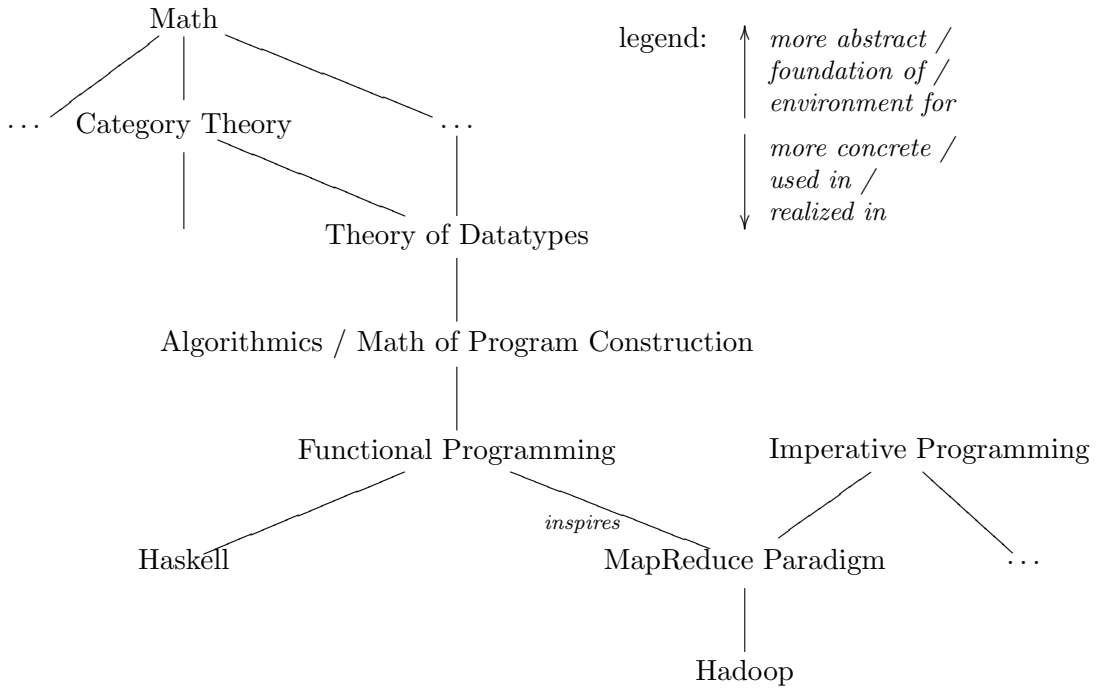
Background info for Map and Reduce

Maarten Fokkinga

Version of December 14, 2011, 16:19

1 Intro. What I will discuss falls in the range of Functional Programming up to the Theory of Datatypes, and is meant to be *background* info for the MapReduce paradigm:



2 Notation. Function composition is written as $f \cdot g$, and application of f to a is written $f\ a$ or $f\ .\ a$. Application written as a space has the highest priority and application written as a "low dot" has the lowest priority (as suggested by the wide space surrounding the dot). So, $f \cdot g\ .\ x + y = f(g\,(x + y))$. This convention saves parentheses, thus improving readability.

In order to facilitate reasoning in the form of algebraic manipulation (that is, repeatedly replacing a part of an expression by a different but semantically equal part) we generally prefer to work on the function level (expressing a function as combination of other functions) instead of the point level (where the outcome of a function application is expressed as a combination of the outcomes of other function applications). Thus we prefer to say, for example, $f = f_1 \cdot (+) \cdot (f_2 \vartriangle f_3) \cdot f_4$ over $f(x) = f_1(f_2(f_4(x)) + f_3(f_4(x)))$, so that the replacement of $(f_2 \vartriangle f_3)$ by $(f_3 \vartriangle f_2)$ or by f' is easier to perform.

The list of items $a, b, c, \ldots$, in that order, is denoted $[a, b, c, \ldots]$. Operation $+\!\!+$ is list concatenation (also called *join*), so that $[a, b, c] +\!\!+ [d, e] = [a, b, c, d, e]$. Function *tip* is the singleton list former: $tip\ x = [x]$.

3 Map and reduce for lists. Here is a definition of 'map f' (denoted $f*$) and '$\oplus$ reduce' (denoted $\oplus/$):

- For arbitrary function f define function $f*$ by: $f* [a, b, c, \ldots, z] = [f\ a,\ f\ b,\ f\ c,\ \ldots,\ f\ z]$.
- For arbitrary operator $\oplus$ define function $\oplus/$ by: $\oplus/ [a, b, c, \ldots, z] = a \oplus b \oplus c \oplus \ldots \oplus z$.

These notions can be generalized to sets (instead of just lists):

- $f* \{a, b, \ldots, z\} = \{f\ a,\ f\ b,\ \ldots,\ f\ z\}$ and $\oplus/ \{a, b, \ldots, z\} = a \oplus b \oplus \ldots \oplus z$.

Here are some examples of map and reduce, showing the ubiquity of their appearances:

- $\mathit{sum}\, [a, b, c, \ldots, z] = a + b + c + \ldots + z = +/ [a, b, c, \ldots, z]$.
  So, sum is $+/$.
- $\mathit{flatten}\, [a, b, c, \ldots, z] = a +\!\!+ b +\!\!+ c +\!\!+ \ldots +\!\!+ z = +\!\!+/ [a, b, c, \ldots, z]$.
  So, $\mathit{flatten}$ is $+\!\!+/$.
- Let $\downarrow$ be "the selection of the minimum of its two operands".
  Then $\downarrow/$ yields the minimum of a list.
  Similarly for max: $\uparrow/$.
- Let $\ll$ be "the selection of the left one of its two operands".
  Then $\ll/$ yields the head of a non-empty list.
  Similarly for the last one of a non-empty list: $\gg/$.
- Let K_1 be the function that maps each argument to 1.
  Then $\mathit{size}[a, b, c, \ldots, z] = 1 + 1 + 1 + \cdots + 1 = +/ \cdot K_1* \ .\ [a, b, c, \ldots, z]$.
  So, size is $+/ \cdot K_1*$.
- Let $x \;\tilde{+\!\!+}\; y = y +\!\!+ x$. Specify $\mathit{reverse}$ by: $\mathit{reverse}\, [a, b, c, \ldots, z] = [z, \ldots, c, b, a]$.
  Then $\mathit{reverse} = \tilde{+\!\!+}/ \cdot \mathit{tip}*$.
- Let $p?$ be the function that maps x to $[x]$ if $p\, x$ holds, otherwise $[]$.
  Then "filter p", denoted $p\triangleleft$, is defined by: $p\triangleleft = +\!\!+/ \cdot (p?)*$.
  Example: $\mathit{odd}\triangleleft [1\,..\,10] = +\!\!+/\ .\ [[1], [\,], [3], [\,], [5], [\,], [7], [\,], [9], [\,]] = [1, 3, 5, 7, 9]$.
- Map and reduce in logic:
  $\exists\, x : [a, b, c, \ldots, z] \bullet Q(x) = \vee/ \cdot Q* \ .\ [a, b, c, \ldots, z]$
  $\forall\, x : [a, b, c, \ldots, z] \bullet Q(x) = \wedge/ \cdot Q* \ .\ [a, b, c, \ldots, z]$
  $\forall\, x : [a, b, c, \ldots, z] \mid P(x) \bullet Q(x) = \wedge/ \cdot Q* \cdot P\triangleleft \ .\ [a, b, c, \ldots, z]$
- Map and reduce in object-relational databases.
  Consider schema $\mathit{Person}\,(\mathit{Name} : \mathit{Txt},\ \mathit{Phone} : \mathit{Num},\ \mathit{Child} : \text{set of } \mathit{Person})$.
  To select the phone number of each grandchild of Joe one may write in SQL:1999:
  select $P.\mathit{Child}.\mathit{Child}.\mathit{Phone}$ from $\mathit{Person}\ P$ where $P.\mathit{Name} =$ 'Joe'.
  Viewing an attribute as a function that selects the required component, the outcome can be written as: $\mathit{Phone}* \cdot \cup/ \cdot \mathit{Child}* \cdot \cup/ \cdot \mathit{Child}* \cdot ((\text{'Joe'}=) \cdot \mathit{Name})\triangleleft \ .\ \mathit{Person}$.

- A property of natural join $\bowtie$.
  Suppose $r = r' \cup r''$ with $r' \cap r'' = \varnothing$ (i.e., r', r'' is a partition of r), and similarly for s.
  Then $r \bowtie s \;=\; r' \bowtie s' \cup r' \bowtie s'' \bowtie r'' \bowtie s' \cup r'' \bowtie s'' \;=\; \cup/ \cdot \bowtie* \;.\; \{r', r''\} \times \{s', s''\}$.
  This generalizes to arbitrary partitions pr, ps of r, s, respectively:
  $r \bowtie s \;=\; \cup/ \cdot \bowtie* \;.\; pr \times ps \;=\; \bigcup\{r' : pr \bullet \bigcup\{s' : ps \bullet r' \bowtie s'\}\}$.
  This property is exploited in the block-nested loop (and other access path for *join*).

- Text processing (taken from Bird [1]). Let *NL* denote the newline character; a *line* of a text is a non-empty maximal segment of the text not containing *NL*. Define $\underline{\text{cat}}$ by $x \;\underline{\text{cat}}\; y = x +\!\!+ [NL] +\!\!+ y$. Then $\underline{\text{cat}}/$ concatenates a list of lines with the *NL* character in between adjacent lines, to form a text.
  The problem is to decompose a text into its constituent lines; more precisely, we want to define a function *lines* that satisfies: $lines(\underline{\text{cat}}/\; xs) = xs$, for each nonempty list of proper lines.
  With some equational reasoning (not done here), one can derive: $lines = \oplus/ \cdot f*$ where $f\,NL = [[], []]$ and $f\;a = [[a]]$ for $a \neq NL$, and $(xs +\!\!+ [x]) \oplus ([y] +\!\!+ ys) = xs +\!\!+ [x +\!\!+ y] +\!\!+ ys$.

4 Caveat. When using the form $\oplus/$ on lists, operator $\oplus$ *must* be associative. On the one hand, this is implicit in the definition of $\oplus/$ that we gave: we did not use parentheses. On the other hand, we can easily show that $\oplus$ inherits the associativity of $+\!\!+$ (so that a contradiction arises if it were not associative):

First observe that $\oplus/[x] = x$ and $\oplus/(xs +\!\!+ ys) = (\oplus/xs) \;\oplus\; (\oplus/ys)$.
(This can be proved, or is the *formal definition*, by induction on the length of the list.)

Then derive :

$$
\begin{array}{ll}
 & (x \oplus y) \oplus z \\
= & \\
 & (\oplus/[x] \;\oplus\; \oplus/[y]) \;\oplus\; (\oplus/[z]) \\
= & \\
 & \oplus/\,((([x] +\!\!+ [y]) +\!\!+ [z]) \\
= & \\
 & \oplus/\,[x, y, z] \\
= & \\
 & \oplus/\,([x] +\!\!+ ([y] +\!\!+ [z]))
\end{array}
\Bigg\} \text{ associativity of } +\!\!+
$$

$$
\begin{array}{ll}
= & \\
 & (\oplus/[x]) \;\oplus\; (\oplus/[y] \;\oplus\; \oplus/[z]) \\
= & \\
 & x \oplus (y \oplus z)
\end{array}
$$

Similarly, when using $\oplus/$ on sets, operation $\oplus$ must not only be associative, but also be commutative ($x \oplus y = y \oplus x$) and absorptive ($x \oplus x = x$).

5 Empty list, neutral elements.
Zero is the neutral element for addition, because $0 + x = x = x + 0$. Similarly:

The neutral element for $\times$ is 1.
The neutral element for $\wedge$ is *true*.
The neutral element for $\vee$ is *false*.
The neutral element for $+\!\!+$ is [].
Operations $\ll, \downarrow, \uparrow$ have no neutral element.

When there exists a neutral element ν for $\oplus$, then we define $\oplus/[] = \nu$. In fact, the value $\oplus/\,[]$ *must* be a neutral element of $\oplus$, for otherwise there is a contradiction similar to the one above for associativity of $\oplus$.

> **Footnote (skip upon first reading).** We can always adjoin a new fictitious element to a domain, and declare it to be the neutral element of a specific operation. For example, we may adjoin a "fictitious number ∞" to the domain of numbers, define this to be the neutral element of $\downarrow$, so that $\infty \downarrow x = x = x \downarrow \infty$ and $\downarrow/\,[] = \infty$. However, no other properties of such a fictitious element may be assumed. For example, $\downarrow/ \cdot (a+)* \;.\; xs = (a+) \cdot \downarrow/ \;.\; xs$ is only valid when restricted to the case $xs \neq []$.

6 Laws for map and reduce. There are various laws for map and reduce, showing that it is worthwhile to have a separate, short, notation for them so that we can easily do algebraic manipulations:

$$
\begin{array}{lcll}
id* & = & id & (id \text{ is the identity function, } id\,x = x) \\
(f \cdot g)* & = & f* \cdot g* & \text{"map distribution"} \\
p\triangleleft \cdot q\triangleleft & = & q\triangleleft \cdot p\triangleleft & \\
p\triangleleft \cdot p\triangleleft & = & p\triangleleft & \\
p\triangleleft \cdot f* & = & f* \cdot (p \cdot f)\triangleleft & \\
p\triangleleft \cdot +\!\!+/ & = & +\!\!+/ \cdot (p\triangleleft)* & \\
f* \cdot +\!\!+/ & = & +\!\!+/ \cdot (f*)* & \\
+/ \cdot +\!\!+/ & = & +/ \cdot (+/)* & \\
\oplus/ \cdot +\!\!+/ & = & \oplus/ \cdot (\oplus/)* & \text{for arbitrary } \oplus \\
+\!\!+/ \cdot +\!\!+/ & = & +\!\!+/ \cdot (+\!\!+/)* & \\
g \;\cdot \oplus/ & = & \otimes/ \cdot g* \qquad \text{if } \; g \cdot (\oplus) = (\otimes) \cdot g \times g & \text{"promotion"}
\end{array}
$$

The last law has the five preceding reduce laws as consequences (by suitable choices for $g, \oplus, \otimes$). The laws can be proved for all finite lists by induction on the length of the list, but the category theoretic approach provides (after the burden and overhead to represent the relevant concepts in categories) an elegant and simple (inductionless!) proof. See the appendix §16; there the "promotion" law is called reduce-Promo (page 12) and is formally proved.

Paragraph 13 show the laws in action, in a formal derivation of the wordcount function in MapReduce style. There we also need a few other operations that facilitate to deal with pairs at the function level:

$$
\begin{array}{lcll}
f \times g & = & \text{the function that maps } (x, y) \text{ to } (f\,x,\; g\,y) & \\
f \,\vartriangle\, g & = & \text{the function that maps } x \text{ to } (f\,x,\; g\,x) & \vartriangle \text{ is pronounced "con"} \\
exl & = & \text{the function that maps } (x, y) \text{ to } x & \text{"left extraction"} \\
exr & = & \text{the function that maps } (x, y) \text{ to } y & \text{"right extraction"}
\end{array}
$$

For these operations the following laws are valid, and many more:

$$
\begin{array}{lcl}
exl \;\cdot\; f \vartriangle g & = & f \\
exr \;\cdot\; f \vartriangle g & = & g
\end{array}
$$

7 Algorithmics. A wide variety of list problems and solutions can be expressed with the operations discussed above. Even more, the *derivation* of a solution from a problem specification can be done by using the above laws (in the same way as mathematicians derive solutions to, say, differential equations or other formally specified problems). Algorithmics [8], or the Mathematics of Program Construction [google on this term], is the field where one tries to put this into practice.

Immediately the question comes to mind whether all data types have such nice and useful laws as for list, as illustrated above. If this not the case, then Algorithmics is doomed to fail: without easily to remember laws, one cannot hope to do algebraic derivations for the widely different problems (and datatypes) that exist in practice. Fortunately, for a large part, there is a pattern in the laws for arbitrary datatypes [5, 9]:

> For *each* inductively defined datatype the notion of *catamorphism* (= "*function defined by induction on the structure of its argument*") exists with similar laws as above. For lists, a catamorphism happens to be equivalent to the form $\oplus/ \cdot f*$.

In full generality, this can be shown in category theory (a particular branch of mathematics). It is beyond the scope of this lecture to introduce, explain, and apply category theory.

> Category theory is a relatively young branch of mathematics, stemming from algebraic topology, and designed to describe various *structural* concepts from different mathematical fields in a *uniform* way. Category theory provides a bag of concepts (and theorems about those concepts) that form an abstraction of many concrete concepts in diverse branches of mathematics, including computing science. The language of category theory facilitates an elegant style of expression and proof (equational reasoning); for the use in algorithmics this happens to be reasoning at the function level, without the need (and the possibility) to introduce arguments explicitly. Also, the formulas often suggest and ease a far-reaching generalization, much more so than the usual set-theoretic formulations.
>
> Honesty requires us to say that category theory is often called "general abstract nonsense", indicating that it is of a high degree of generality, very abstract, and hard to understand upon first reading.

Not aiming at full generality, our MapReduce note [6] shows the claim for a typical example. Appendix §16 formalizes the same example, and proves the laws for maps and reduces listed earlier, in a setting that is close to category theory.

8 The word count algorithm. The notions of "map" and "reduce" defined above are standard in the literature, but differ somewhat from the notions of "the map" and "the reduce" by Dean and Ghemawat [2]. To get a feeling for the correspondence and discrepancy, we discuss the word count algorithm of Dean in our notation. Lämmel [7] gives an elaborate discussion, resulting in a more elaborate Haskell program in his Figure 1. (In my opinion, Lämmel's formulation *hides* the essence rather than illuminating it, due to the lengthy Haskell notation and the absence of short notations like $f*$ and $\oplus/$).

These entities are given:

$$\begin{array}{lcl} Doc & & \text{a set} \\ Key & & \text{a set} \\ Word & & \text{a set} \\ words & : & Doc \to [Word] \end{array}$$

We'll interpret *words d* as "the list of words in *d*". The problem is to produce from a given dictionary *ds* (of type $Key \to Doc$) a function (of type $Word \to \mathbb{N}$) which assigns to each

word w the total number of occurrences of w in the documents in ds:

$$
\begin{array}{lll}
\mathit{wordcount} & : & (\mathit{Key} \to \mathit{Doc}) \to (\mathit{Word} \to \mathbb{N}) \\
(9)\quad \mathit{wordcount}\ ds\ w & = & \text{the total number of occurrences of } w \text{ in the docs yielded by } ds \\
& = & \textstyle\sum_{k:\,\mathrm{dom}\ ds} \text{ the number of occurrences of } w \text{ in } ds\ k \\
& = & \textstyle\sum_{k:\,\mathrm{dom}\ ds}\ \mathit{size} \cdot (=w)\triangleleft \cdot \mathit{words}\ .\ ds\ k
\end{array}
$$

Notice that, for distinct k and k', even though $ds\,k$ and $ds\,k'$ might be the same document, the word occurrences in $ds\,k$ and $ds\,k'$ are counted separately.

A MapReduce expression for the word count function has two requirements:

- The input and output function of *wordcount* are represented by lists of argument-result pairs. Thus the type of the required wordcount function, now called $\mathit{wordcount}'$, is:

$$\mathit{wordcount}' \quad : \quad [\mathit{Key} \times \mathit{Doc}] \to [\mathit{Word} \times \mathbb{N}]$$

- The defining expression for $\mathit{wordcount}'$ should have a "reduce per key" part (containing the reduce), a "group per key" part, and a "map per key" part (containing the map).

We give and explain now right-away two definitions of $\mathit{wordcount}'$, namely (11) and (12), and in the next paragraph we *formally derive* these from specification (9). In order to get at a concise expression, we need a few auxiliary notations and concepts:

$$
\begin{array}{lll}
(\_, 1) & = & \mathit{id} \vartriangle K_1 \quad = \quad \text{the function that maps } x \text{ to } (x, 1) \\
(10)\quad \mathit{grp} & = & \text{a function that maps } [..., (w, n_1), ..., (w, n_2), ..., (w, n_k), ...] \text{ to } [..., (w, ns), ...] \\
& & \text{where } ns \text{ is a list containing precisely } n_1, n_2, ..., n_k
\end{array}
$$

Function $\mathit{wordcount}'$ now reads as follows, in two versions:

$$
\begin{array}{lll}
(11)\quad \mathit{wordcount}' & = & (\mathit{id} \times (+/))* \ \cdot\ \mathit{grp}\ \cdot\ +\!\!+/\ \cdot\ (\qquad\qquad\qquad\qquad (\_, 1)* \cdot \mathit{words} \cdot \mathit{exr})* \\
(12)\quad & = & (\mathit{id} \times (+/))* \ \cdot\ \mathit{grp}\ \cdot\ +\!\!+/\ \cdot\ ((\mathit{id} \times (+/)) \cdot \mathit{grp} \cdot (\_, 1)* \cdot \mathit{words} \cdot \mathit{exr})*
\end{array}
$$

In definition (12) the reduce $+/$ is already partly performed as part of the initial map. Reading equation (11) from right to left, the right-hand side constructs its result as follows, given a list of key-document pairs:

- Per key-document pair:
  - *exr*: discards the key, retaining only the document,
  - *words*: produces the list of all word occurrences in the document,
  - $(\_, 1)*$: replaces each word w in the list by $(w, 1)$;
- $+\!\!+/$: concatenates the "$(w, 1)$-lists" (one per document) to one long "$(w, 1)$-list";
- *grp*: groups items $(w, 1)$ per word, yielding one $(w, [..., 1, ...])$-list;
- $(\mathit{id} \times +/)*$: replaces, in the list, each $(w, [1, ...])$ item by $(w, +/[1, ...])$.

Another way to understand the right-hand side of (11), is to look at the types of the intermediate results. Abbreviating $\mathit{Doc}, \mathit{Key}, \mathit{Word}$ to D, K, W, these types read:

$$\underset{[W\times\mathbb{N}]}{\uparrow}\ (\ \underset{W\times\mathbb{N}}{\uparrow}\ id\times(+/)\ \underset{W\times[\mathbb{N}]}{\uparrow}\ )*\ \cdot\ \underset{[W\times[\mathbb{N}]]}{\uparrow}\ grp\ \cdot\ \underset{[W\times\mathbb{N}]}{\uparrow}\ +\!\!+/\ \cdot\ \underset{[[W\times\mathbb{N}]]}{\uparrow}\ (\ \underset{[W\times\mathbb{N}]}{\uparrow}\ (\_,1)*\ \cdot\ \underset{[W]}{\uparrow}\ words\ \cdot\ \underset{D}{\uparrow}\ exr\ \underset{K\times D}{\uparrow}\ )*\ \underset{[K\times D]}{\uparrow}$$

Dean and Ghemawat [2] and Lämmel [7] name some parts of the expressions as follows:

$$\overbrace{(id\times\boxed{+/})*}^{reducePerKey}\cdot\ \overbrace{grp}^{grpPerKey}\ \cdot\ \overbrace{+\!\!+/\cdot(\ \boxed{(\_,1)*\cdot words\ \cdot\ exr}\ )*}^{mapPerKey}$$

$$(id\times\boxed{+/})*\cdot\ grp\ \cdot\ +\!\!+/\cdot(\ (id\times\boxed{+/})*\cdot grp\ \cdot\boxed{(\_,1)*\cdot words\ \cdot\ exr}\ )*$$

rEDUCE	*cOMBINE*	*mAP*
the reduce	the combiner	the map

In the next paragraph we show how these expressions for *wordcount′* can be derived from the definition of *wordcount*.

13 Derivation of the MapReduce wordcount function. For simplicitly and readability we identify functions with their list representations, that is, we do not write explicitly the operation that maps a function to a list of argument-result pairs, or the other way around. Related to this identification we define $xs \bullet x$ to represent function application where the function is represented by a list xs of argument-result pairs:

$$[\ldots(x,y)\ldots]\bullet x \quad = \quad y$$

Based on the specification (10) of *grp*, we conclude the following law, for a function f such that $f\ xs = f\ ys$ if xs and ys represent the same bag or multi-set (and stipulating that $[\,]\bullet x = f\,[\,]$):

$$(14)\quad (\bullet x)\cdot(id\times f)*\cdot grp \quad = \quad f\cdot exr*\cdot((=x)\cdot exl)\triangleleft$$

Paragraph §15 gives a formal definition of *grp* and proves the law.

The derivation proceeds as follows (*ds* is mnemonic for documents, *w* for word):

$$\begin{array}{ll}
 & wordcount\ ds\ w \\
= & \quad \text{definition } wordcount \\
 & \sum_{k:\mathrm{dom}\ ds}\ (size\cdot(=w)\triangleleft\cdot words)\ .\ ds\ k \\
= & \quad \text{list representation of funtions: } \sum_{k:\mathrm{dom}\ f} F(f\ k) = +/\cdot F\cdot exr*\ .\ f \\
 & +/\cdot(size\cdot(=w)\triangleleft\cdot words)*\ \cdot exr*\ .\ ds \\
= & \quad \text{definition } size \\
 & +/\cdot(+/\cdot K_1*\cdot(=w)\triangleleft\cdot words)*\ \cdot exr*\ .\ ds \\
= & \quad \text{map distribution} \\
 & +/\cdot+/*\cdot K_1**\cdot(=w)\triangleleft**\cdot words**\cdot exr*\ .\ ds \\
= & \quad \text{promotion, noting that } +/(xs +\!\!+ ys) = (+/xs)+(+/ys) \\
 & +/\cdot+\!\!+/\cdot K_1**\cdot(=w)\triangleleft**\cdot words**\cdot exr*\ .\ ds \\
= & \quad \text{law: } exr\cdot f\vartriangle g = g, \text{ together with defn } (_,1) = id\ \vartriangle\ K_1 \\
 & +/\cdot+\!\!+/\cdot(exr\cdot(_,1))**\cdot(=w)\triangleleft**\cdot words**\cdot exr*\ .\ ds \\
= & \quad \text{map distribution}
\end{array}$$

$$
\begin{array}{ll}
 & +\!\!/ \cdot +\!\!\!+\!\!/ \cdot (exr* \cdot (\_,1)* \cdot (=w)\triangleleft)* \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{law: } f = f \cdot id \\
 & +\!\!/ \cdot +\!\!\!+\!\!/ \cdot (exr* \cdot (\_,1)* \cdot ((=w) \cdot id)\triangleleft)* \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{law: } f = exl \cdot f \vartriangle g \text{, together with } (\_,1) \ = \ (\lambda\, x.\ (x,1)) \ = \ id \vartriangle K_1 \\
 & +\!\!/ \cdot +\!\!\!+\!\!/ \cdot (exr* \cdot (\_,1)* \cdot ((=w) \cdot exl \cdot (\_,1))\triangleleft)* \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{law: } f* \cdot (p \cdot f)\triangleleft \ = \ p\triangleleft \cdot f* \\
 & +\!\!/ \cdot +\!\!\!+\!\!/ \cdot (exr* \cdot ((=w) \cdot exl)\triangleleft \cdot (\_,1)*)* \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{map distribution} \\
 & +\!\!/ \cdot +\!\!\!+\!\!/ \cdot exr** \cdot ((=w) \cdot exl)\triangleleft* \cdot (\_,1)** \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{promotion: } +\!\!\!+\!\!/ \cdot f** \ = \ f* \cdot +\!\!\!+\!\!/ \quad \text{and} \quad +\!\!\!+\!\!/ \cdot p\triangleleft* \ = \ p\triangleleft \cdot +\!\!\!+\!\!/ \\
 & +\!\!/ \cdot exr* \cdot ((=w) \cdot exl)\triangleleft \cdot +\!\!\!+\!\!/ \cdot (\_,1)** \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{property (14): for all } x, f \text{ we have } (\bullet x) \cdot (id \times f)* \cdot grp \ = \ f \cdot exr* \cdot ((=x) \cdot exl)\triangleleft \\
 & (\bullet w) \cdot (id \times +\!\!/)* \cdot grp \cdot +\!\!\!+\!\!/ \cdot (\_,1)** \cdot words* \cdot exr* \ .\ ds \\
= & \quad \text{law: } (\bullet x) \cdot f = f\ x \quad \text{(using our representation convention for functions)} \\
 & \Big((id \times +\!\!/)* \cdot grp \cdot +\!\!\!+\!\!/ \cdot (\_,1)** \cdot words* \cdot exr* \ .\ ds\Big)\ w \\
= & \quad \text{law: } (f \ .\ x)\, y = f\ x\ y \\
 & \Big((id \times +\!\!/)* \cdot grp \cdot +\!\!\!+\!\!/ \cdot (\_,1)** \cdot words* \cdot exr*\Big)\ ds\ w \\
= & \quad \text{map distribution} \\
 & \Big((id \times +\!\!/)* \cdot grp \cdot +\!\!\!+\!\!/ \cdot ((\_,1)* \cdot words \cdot exr)*\Big)\ ds\ w \qquad \text{Q.E.D}
\end{array}
$$

Unfortunately, we are currently unable to derive the optimization (12) in a satisfactory way.

15 Grouping. We present a possible definition of *grp* and then prove property (14). The remainder of the paper does not depend on the material in this section, so you may skip reading it without loss of continuity.

Inspired by its specification (10) we define *grp* as follows, using list comprehension:

$$grp\ xs \ = \ [\ (x,\ exr* \cdot ((=)x \cdot exl)\triangleleft \ .\ xs) \ \mid \ x \leftarrow exl*\ xs]$$

Now property (14) is easily proved:

$$
\begin{array}{ll}
 & (\bullet x) \cdot (id \times f)* \cdot grp \ .\ xs \\
= & \quad \text{definition } grp \\
 & (\bullet x) \cdot (id \times f)* \ .\ [(x,\ exr* \cdot ((=)x \cdot exl)\triangleleft \ .\ xs) \ \mid \ x \leftarrow exl*\ xs] \\
= & \quad \text{map on list comprehension} \\
 & (\bullet x) \ .\ [(id \times f)(x,\ exr* \cdot ((=)x \cdot exl)\triangleleft \ .\ xs) \ \mid \ x \leftarrow exl*\ xs] \\
= & \quad \text{definition } \times \text{ and } id \\
 & (\bullet x) \ .\ [(x,\ f \cdot exr* \cdot ((=)x \cdot exl)\triangleleft \ .\ xs) \ \mid \ x \leftarrow exl*\ xs] \\
= & \quad \text{definition } \bullet x, \text{ assuming } x \text{ occurs in } xs \\
 & f \cdot exr* \cdot ((=)x \cdot exl)\triangleleft \ .\ xs \qquad \text{q.e.d.}
\end{array}
$$

There is another property of *grp* that might prove useful, for example in a derivation of (12). First observe the type of *grp*:

for all α and β: $[\alpha \times \beta] \to [\alpha \times [\beta]]$

This typing implies by the "Theorems for Free" theorem [10] the following equality:

$$(f \times g*)* \cdot grp \quad = \quad grp \cdot (f \times g)*$$

16 Appendix: Formal treatment of map and reduce

This section contains a formal treatment of map and reduce. We have placed this material in an appendix and not in the main text, because it is not of immediate help for programming with map and reduce (which is the focus of the course MapReduce). Instead, this section is really background, and aims to show an elegant way of defining inductively defined datatypes and functions defined by induction over them, and to prove in an elegant way various laws for them. In particular, although everywhere it might be helpful to think of the datatype of lists or trees (and thus *tip* is the singleton former and *join* is list concatenation) for understanding what's going on, nowhere that particular interpretation is really used (so that the entire story readily generalizes to other datatypes as well). In order to achieve concise and readable expressions (presumably requiring some exercising to be understandable), we do not use function application in our formulas but use function composition instead. Moreover, we use $\amalg f = f \times f$ for the function that maps (x, x) to (fx, fx). Observe that $\amalg(f \cdot g) = \amalg f \cdot \amalg g$ and $\amalg id = id$. We also put $\amalg A = A \times A$ for arbitrary set A. (Pronounce $\amalg$ as "twin", and notice that a binary operation on A has a type of the form $\amalg A \to B$.)

Definition (Binary trees). Let A be a given set, fixed throughout the sequel. The assertion "T is the inductively defined datatype with constructors $tip : A \to T$ and $join : \amalg T \to T$ and catamorphism $(\!|\_, \_|\!)$" means *exactly* the following:

- T is a set.
- *tip* and *join* are functions of types $A \to T$ and $\amalg T \to T$, respectively.
- For arbitrary function $f : A \to B$ and binary operation $opn : \amalg B \to B$ there exists a function of type $T \to B$, denoted $(\!|f, opn|\!)$, which is characterized by:

$$h \cdot tip = f \quad \wedge \quad h \cdot join = opn \cdot \amalg h \qquad \Leftrightarrow \qquad h = (\!|f, opn|\!) \qquad \boxed{\text{cata-CHARN}}$$

Law cata-CHARN essentially says that function $(\!|f, opn|\!)$, call it h, is defined by induction on the *tip*, *join*-construction of its argument:

- clause $h \cdot tip = f$ says that the result of h on argument $tip\, x$ equals $f\, x$, and
- clause $h \cdot join = opn \cdot \amalg h$ says that the result of h on an argument constructed as $join(x, y)$ is defined as $opn(h\, x, h\, y)$.

Thus h systematically replaces each *tip* by f and each *join* by *opn*. The preposition *cata* comes from Greek: $\kappa\alpha\tau\alpha$ means downward; *morphism* is frequently used in mathematics for a kind of mapping that in some sense preserves the *shape* ($\mu o \rho \varphi$). Before further elaborating the consequences of cata-CHARN, let us first observe that this datatype definition plays the role of a typical example; other datatypes are obtained by the following generalization:

- Instead of just *tip* and *join*, take a collection of operations $cons_i : F_i(T_A) \to T_A$.

  Each $cons_i$ is called "*constructor*" and $F_i(T_A)$ is an expression built from set A, Cartesian product $\times$, disjoint union $+$, other given datatypes, and T_A. For the binary tree example, we have:

$$\begin{array}{lcll} cons_1 = tip & : & A \to T_A & \text{here, } F_1(X) = A \\ cons_2 = join & : & T_A \times T_A \to T_A & \text{here, } F_2(X) = X \times X \end{array}$$

- Optionally, some laws may be postulated for the constructors.

  In the binary tree example, the laws of associativity, commutativity, and absorptivity lead to lists, bags, and sets. The laws should also hold for the parameter of a catamorphism. We do not elaborate the addition of laws here; elsewhere I've discussed the formalization [3].

- It turns out, below, that T_A is the set consisting of all expressions (modulo the laws) built from the constructors.

  For binary trees we have:

$$T_A = \{tip\ a, tip\ a', \ldots, join(tip(a), tip(a')), \ldots, join(join(tip\ a, tip\ a'), tip\ a''), \ldots\}$$

Now we further "explain" $(\!|f, opn|\!)$ by showing four immediate consequences of law cata-CHARN, and then specialize it to maps and reduces.

(1) Making cata-CHARN's right-hand side true by taking $h := (\!|f, opn|\!)$ gives the way how a catamorphism is defined "by induction on the structure of its argument":

$$(\!|f, opn|\!) \cdot tip = f \quad \wedge \quad (\!|f, opn|\!) \cdot join = opn \cdot I\!I\,(\!|f, opn|\!) \qquad \boxed{\text{cata-SELF}}$$

Indeed, this is what we did above: the left conjunct says what the result of the catamorphism is on arguments of the form $tip(x)$, and the right conjunct expresses what the result of the catamorphism is on arguments of the form $join(x, y)$ in terms of the outcome of the catamorphism on x and y.

(2) Two functions that both satisfy cata-CHARN's lhs must be equal according to the rhs:

$$h \cdot tip = h' \cdot tip \ \wedge\ h \cdot join = opn \cdot I\!I\,h \ \wedge\ h' \cdot join = opn \cdot I\!I\,h' \ \Rightarrow\ h = h' \qquad \boxed{\text{cata-UNIQ}}$$

This law captures the essence of "proof by induction". In this case the equality of two functions is asserted in case they behave the same in the *tip*-case and in the *join*-case; no further call for an induction principle or whatever is needed! As a consequence we conclude that T only contains elements that can be finitely generated by *tip* and *join* (for otherwise, if T would contain an element x not expressible via *tip* and *join*, then h and h' might differ on x whereas they would satisfy the premiss).

(3) Making cata-CHARN's left-hand side true by taking $f, opn, h := tip, join, id$ gives:

$$id = (\!|tip, join|\!) \qquad \boxed{\text{cata-ID}}$$

Indeed, inductively replacing *tip* by *tip* and *join* by *join* is effectively the identity function.

(4) Here is corollary of cata-CHARN that gives a sufficient condition under which a composition of g with a catamorphism is itself a catamorphism. The law is frequently applicable in theoretical reasoning (as demonstrated in the sequel) and also in practical reasoning, since the expression as a composition might be more *understandable* as a specification, whereas the expression as a single catamorphism is probably more *efficient* as an algorithm:

$$g \cdot f = f' \quad \wedge \quad g \cdot opn = opn' \cdot \amalg g \qquad \Rightarrow \qquad g \cdot (\!|f, opn|\!) = (\!|f', opn'|\!) \qquad \boxed{\text{cata-FUSION}}$$

The proof of this law is a nice example of reasoning "at the function level":

$$\begin{array}{ll}
& g \cdot (\!|f, opn|\!) = (\!|f', opn'|\!) \\
\Leftrightarrow & \quad \text{cata-CHARN with } h \text{ replaced by } g \cdot (\!|f, opn|\!) \\
& g \cdot (\!|f, opn|\!) \cdot tip = f' \quad \wedge \quad g \cdot (\!|f, opn|\!) \cdot join = opn' \cdot \amalg (g \cdot (\!|f, opn|\!)) \\
\Leftrightarrow & \quad \text{law cata-SELF for the left two occurrences of } (\!|f, opn|\!) \\
& g \cdot f = f' \quad \wedge \quad g \cdot opn \cdot \amalg (\!|f, opn|\!) = opn' \cdot \amalg (g \cdot (\!|f, opn|\!)) \\
\Leftrightarrow & \quad \text{at the extreme right: law } \amalg (x \cdot y) = \amalg x \cdot \amalg y \\
& g \cdot f = f' \quad \wedge \quad g \cdot opn \cdot \amalg (\!|f, opn|\!) = opn' \cdot \amalg g \cdot \amalg (\!|f, opn|\!) \\
\Leftarrow & \quad \text{law "Leibniz": } x \cdot z = y \cdot z \Leftarrow x = y \\
& g \cdot f = f' \quad \wedge \quad g \cdot opn = opn' \cdot \amalg g
\end{array}$$

There are many more laws derivable in this general setting [4, 5, 9].

Specialization to map and reduce. The formal definitions of *map* and *reduce* read:

$$f* \quad = \quad (\!|tip \cdot f,\ join|\!) \qquad \boxed{\text{map-DEF}}$$

$$opn/ \quad = \quad (\!|id, opn|\!) \qquad \boxed{\text{reduce-DEF}}$$

Since each is a catamorphism, cata-SELF immediately gives their "inductive definition":

$$f* \cdot tip = tip \cdot f \quad \wedge \quad f* \cdot join = join \cdot \amalg f* \qquad \boxed{\text{map-SELF}}$$

$$opn/ \cdot tip = id \quad \wedge \quad opn/ \cdot join = opn \cdot \amalg opn/ \qquad \boxed{\text{reduce-SELF}}$$

The well-known distributivity properties of map read:

$$id = id* \qquad \boxed{\text{map-ID}}$$

$$f_2* \cdot f_1* = (f_2 \cdot f_1)* \qquad \boxed{\text{map-FUSION}}$$

The former one is an immediate application of cata-ID, and the latter one is merely an application of cata-FUSION with the substitution $g, f', f := f_2*, (tip \cdot f_1), (tip \cdot f_1 \cdot f_2)$:

$$\begin{array}{ll}
& f_2* \cdot f_1* = (f_2 \cdot f_1)* \\
\Leftrightarrow & \quad \text{map-DEF} \\
& f_2* \cdot (\!|tip \cdot f_1, join|\!) = (\!|tip \cdot f_2 \cdot f_1, join|\!) \\
\Leftarrow & \quad \text{cata-FUSION} \\
& f_2* \cdot tip \cdot f_1 = tip \cdot f_2 \cdot f_1 \quad \wedge \quad f_2* \cdot join = join \cdot \amalg f_2* \\
\Leftrightarrow & \quad \text{map-SELF}
\end{array}$$

$true$

Not only is each map and each reduce a catamorphism, a composition of them is a single catamorphism as well:

$$opn/ \cdot f* = (\!|f, opn|\!) \qquad \boxed{\text{Map-Reduce-Cata}}$$

The proof is a simple application of cata-Fusion:

$$\begin{array}{ll} & opn/ \cdot f* = (\!|f, opn|\!) \\ \Leftrightarrow & \quad \text{map-Def} \\ & opn/ \cdot (\!|tip \cdot f, join|\!) = (\!|f, opn|\!) \\ \Leftarrow & \quad \text{cata-Fusion} \\ & opn/ \cdot tip \cdot f = f \quad \wedge \quad opn/ \cdot join = opn \cdot I\!I\, opn/ \\ \Leftrightarrow & \quad \text{reduce-Self} \\ & true \end{array}$$

As a consequence, law cata-Id translates to maps and reduces:

$$id = join/ \cdot tip* \qquad \boxed{\text{map-reduce-Id}}$$

Finally, the well-know and frequently applicable "promotion" law for reduces reads:

$$g \cdot opn = opn' \cdot I\!I\, g \quad \Rightarrow \quad g \cdot opn/ = opn'/ \cdot g* \qquad \boxed{\text{reduce-Promo}}$$

Its proof is, again, an application of cata-Fusion:

$$\begin{array}{ll} & g \cdot opn/ = opn'/ \cdot g* \\ \Leftrightarrow & \quad \text{in the lhs reduce-Def, in the rhs Map-Reduce-Cata} \\ & g \cdot (\!|id, opn|\!) = (\!|g, opn'|\!) \\ \Leftarrow & \quad \text{cata-Fusion} \\ & g \cdot id = g \quad \wedge \quad g \cdot opn = opn' \cdot I\!I\, g \end{array}$$

This concludes our discussion of laws for reduce and maps. Notice once more that the concrete notion of list or tree has not been used. The only evidence that we have been working with lists or trees is the fact that T has just two constructors, with the types that the concrete "tip/singleton former" and "binary join/concatenation" happen to have. Since many datatypes can be defined in the same way that we did for T above, but with a different number of constructors and different types for the constructors, the theory discussed above is quite general indeed.

References

[1] R.S. Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer Verlag, 1987. Also Technical Monograph PRG–56, Oxford University, Computing Laboratory, Programming Research Group, October 1986.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150. USENIX, December 2004. Google Research Paper, `http://labs.google.com/papers/mapreduce.html`. Also: Communications of the ACM, Jan 2008, Vol. 5, nr 1,pp. 107–113.

[3] Maarten M. Fokkinga. Datatype laws without signatures. *Mathematical Structures in Computer Science*, 6(1):1–32, 1996. Obtainable by `http://www.cs.utwente.nl/~fokkinga/mmf91h.pdf`.

[4] M.M. Fokkinga. Calculate categorically! *Formal Aspects of Computing*, 4(4):673–692, 1992. Obtainable by `http://www.cs.utwente.nl/~fokkinga/mmf91j.pdf`.

[5] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, Dept INF, Enschede, The Netherlands, 1992. Obtainable by `http://www.cs.utwente.nl/~fokkinga/mmfphd.pdf`.

[6] M.M. Fokkinga. Mapreduce — a two-page explanation for laymen. Unpublished Technical Report, obtainable from `http://www.cs.utwente.nl/~fokkinga/mmf2008j.pdf`, 2008.

[7] Ralf Lämmel. Google's MapReduce programming model - Revisited. *Sci. Comput. Program.*, 70(1):1–30, 2008.

[8] L. Meertens. Algorithmics — towards programming as a mathematical activity. In J.W. de Bakker and J.C. van Vliet, editors, *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pages 289–334. North-Holland, 1986.

[9] E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *Lect. Notes in Comp. Sc.*, pages 124–144. Springer Verlag, 1991. Obtainable by `http://www.cs.utwente.nl/~fokkinga/mmf91m.ps`.

[10] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, September 1989. FPCA '89, Imperial College, London.

Addendum: Another derivation the wordcount function

We have placed this section in an addendum, because the *interesting* and *beautiful* derivation below is –in retrospect– quite *unsatisfactory*: the specification of *grp* seems to depend on the specific operations of *wordcount*. We recommend the reader not to read this section except when you have the time and the desire to improve my attempt.

* * *

Here we present a fully formal derivation of the MapReduce-style wordcount expression from the initial definition. My personal ambition is to strive for expressions that are as "conceptual" as possible, thus using functions and manipulations of functions instead of lists and manipulations of lists. The height of the conceptual level might be daunting to newcomers in this field: without some experience with down-to-earth list manipulations at the point level, there is a chance that one might find our exposition at the function level hard to follow.

> **Note.** All of the lists and list-joins below must be replaced by (or interpreted as) bags and bag-unions (so that the order of the elements becomes insignificant), for otherwise some of the equations are invalid. We do not perform this replacement since it would involve extra and new notation. We trust the reader can do so tacitly.

To smoothen our exposition, we first discuss how we –eventually– represent $W \to \mathbb{N}$ functions with nonzero outcomes on only a finite number of arguments, by finite $(W, \mathbb{N})$-lists, and how such lists are interpreted as functions. We start with some auxiliary tools:

$$\begin{array}{lcl} mkFct(x,y) & = & \text{the function mapping } x \text{ to } y\text{, and other arguments to } 0 \\ f \mathbin{\hat{+}} g & = & \text{the function mapping arbitrary } x \text{ to } f\,x + g\,x \end{array}$$

Operation $\hat{+}$ is called "lifted +"; it obeys the same laws as + does (associativity, commutativity). Now, the representation *toL* and the interpretation *toF* are defined as follows:

$$\begin{array}{lcl} toL\,f & = & [(x,\ f\,x) \mid x \leftarrow \text{"domain of } f\text{"};\ f\,x \neq 0] \\ toF & = & \hat{+}/ \cdot mkFct* \end{array}$$

We have the following useful consequences:

$$\begin{array}{lcl} & toL \cdot toF & \text{is a kind of list normalization (it will occur frequently in the sequel)} \\ (17) & toF \cdot toL & = \text{the identity function } id \text{ on type } W \to N \\ (18) & toF \cdot +\!\!+/ & = toF \cdot +\!\!+/ \cdot (toL \cdot toF)* \end{array}$$

The 1st claim is informal, the 2nd claim is almost immediate; the 3rd is proved as follows:

$$\begin{array}{ll} & toF \cdot +\!\!+/ \\ = & \quad \text{promotion (§6, page 4), since: } \ toF \cdot (+\!\!+) = (\hat{+}) \cdot (toF \times toF) \\ & \hat{+}/ \cdot toF* \\ = & \quad \text{eqn (17): } id = toF \cdot toL \\ & \hat{+}/ \cdot (toF \cdot toL \cdot toF)* \\ = & \hat{+}/ \cdot toF* \cdot (toL \cdot toF)* \\ = & \quad \text{promotion, as in the first line but now in the reverse direction} \end{array}$$

$$toF \cdot +\!\!+\!\!/ \cdot (toL \cdot toF)*$$

This completes the preparation.

In order to *derive* claim (11) and (12) for *wordcount′* from definition (9) of *wordcount*, let us first play a bit with the defining expression (9) for *wordcount ds w*:

$$\begin{array}{ll}
 & wordcount\ ds\ w \\
= & \\
 & \sum_{k:\,\mathrm{dom}\ ds}\ size \cdot (=w)\triangleleft \cdot words\ .\ ds\,k \\
= & \qquad \text{summation is a reduce: } \sum_{x:\mathrm{dom}\,f}\ g\,(f\,x)\ =\ +\!/ \cdot (g \cdot exr)*\ .\ toL\,f \\
 & +\!/ \cdot (size \cdot (=w)\triangleleft \cdot words \cdot exr)*\ .\ toL\,ds \\
= & \\
 & +\!/ \cdot (size \cdot (=w)\triangleleft)* \cdot words* \cdot exr*\ .\ toL\,ds \\
= & \qquad \text{promotion (see §6, page 4): with } g, \oplus, \otimes\ =\ size \cdot (w=)\triangleleft,\ (+\!\!+),\ (+) \\
 & size \cdot (=w)\triangleleft \cdot +\!\!+\!\!/ \cdot words* \cdot exr*\ .\ toL\,ds \\
(\dagger) \quad = & \qquad \text{cata-UNIQ, elaborated below (this is an "eureka" step)} \\
 & (\ .\ w) \cdot toF \cdot (_, 1)* \cdot +\!\!+\!\!/ \cdot words* \cdot exr*\ .\ toL\,ds \\
= & \\
 & \qquad (\ toF \cdot (_, 1)* \cdot +\!\!+\!\!/ \cdot words* \cdot exr*\ .\ toL\,ds\ )\ .\ w
\end{array}$$

Hence, abstracting from w and then from ds too, we get:

$$(19)\quad wordcount\ =\ toF \cdot (\_, 1)* \cdot +\!\!+\!\!/ \cdot words* \cdot exr* \cdot toL$$

Proof of step (†). Law cata-UNIQ says that two functions are equal whenever they "inductively behave the same" on arguments of the form $x +\!\!+ y$ and $[x]$. (The law is formally derived in the appendix §16.) We show this for $f = size \cdot (=w)\triangleleft$ and $g = (\ .\ w) \cdot toF \cdot (\_, 1)*$. First, we prove $f\,[x] = g\,[x]$:

$$\begin{array}{ll}
 & f\ .\ [x] \\
= & \\
 & size \cdot (=w)\triangleleft\ .\ [x] \\
= & \\
 & 1 \text{ if } x = w \text{ else } 0 \\
= & \\
 & (\lambda\, x'.\ 1 \text{ if } x = x' \text{ else } 0)\ w \\
= & \\
 & (\ .\ w) \cdot (\lambda\, x'.\ 1 \text{ if } x = x' \text{ else } 0) \\
= & \\
 & (\ .\ w) \cdot toF \cdot (_, 1)*\ .\ [x] \\
= & \\
 & g\ .\ [x]
\end{array}$$

Second, we prove $f\,(x +\!\!+ y) = f\,x\ +\ f\,y$ and also $g\,(x +\!\!+ y) = g\,x\ +\ g\,y$:

$$\begin{array}{ll}
 & f\ .\ x +\!\!+ y \\
= & \\
 & size \cdot (=w)\triangleleft\ .\ x +\!\!+ y \\
= & \\
 & (size \cdot (=w)\triangleleft\ .\ x)\ +\ (size \cdot (=w)\triangleleft\ .\ y) \\
= & \\
 & f\,x\ +\ f\,y
\end{array}$$

and

$$\begin{array}{ll}
 & g\ .\ x +\!\!+ y \\
= & \\
 & (\ .\ w) \cdot toF \cdot (_, 1)*\ .\ x +\!\!+ y \\
= & \\
 & (\ .\ w)\ .\ ((toF \cdot (_, 1)*\ .\ x)\ \hat{+}\ (toF \cdot (_, 1)*\ .\ y)) \\
= & \\
 & ((\ .\ w) \cdot toF \cdot (_, 1)*\ .\ x)\ +\ ((\ .\ w) \cdot toF \cdot (_, 1)*\ .\ y) \\
= & \\
 & g\,x\ +\ g\,y
\end{array}$$

End of proof of step (†)

Now we turn to the MapReduce expression for $wordcount'$. Remember, $wordcount'$ does the same as $wordcount$ except that it uses the list representation of functions. So, $wordcount'$ is formally specified by:

(20) $wordcount' \cdot toL \quad = \quad toL \cdot wordcount$

Consequently, for arguments of the form $ds' = toL\, ds$ we have:

$$
\begin{array}{ll}
 & wordcount'\ ds' \\
= & \\
 & wordcount' \cdot toL\ .\ ds \\
= & \qquad \text{specification (20) of } wordcount' \\
 & toL \cdot wordcount\ .\ ds \\
= & \qquad \text{just derived for } wordcount\text{: eqn (19)} \\
 & toL \cdot toF \cdot (\_, 1)* \cdot +\!\!/ \cdot words* \cdot exr* \cdot toL\ .\ ds \\
= & \\
 & toL \cdot toF \cdot (\_, 1)* \cdot +\!\!/ \cdot words* \cdot exr*\ .\ ds' \\
= & \\
(21) \quad & toL \cdot toF \cdot +\!\!/ \cdot (\_, 1)** \cdot words* \cdot exr*\ .\ ds' \\
= & \qquad \text{law (18)} \\
(22) \quad & toL \cdot toF \cdot +\!\!/ \cdot (toL \cdot toF)* \cdot (\_, 1)** \cdot words* \cdot exr*\ .\ ds'
\end{array}
$$

So, defining $wordcount'$ by (21) or (22), its specification (20) is satisfied. Next we aim at a concrete realization. To this end we *specify*(!) function grp by:

$$(id \times +\!\!/)* \cdot grp \quad = \quad toL \cdot toF$$

(We leave it to the industrious reader to find a suitable defining expression.) Then we get from (21) and (22), and map distribution, the equations (11) and (12) for $wordcount'$ that we were aiming at:

$$
\begin{array}{lll}
wordcount' & = & (id \times +\!\!/)* \cdot grp \cdot +\!\!/ \cdot (\qquad\qquad\qquad\quad (\_, 1)* \cdot words \cdot exr)* \\
 & = & (id \times +\!\!/)* \cdot grp \cdot +\!\!/ \cdot ((id \times +\!\!/)* \cdot grp \cdot (\_, 1)* \cdot words \cdot exr)*
\end{array}
$$

Notice that the semantic equality of these two defining expressions for $wordcount'$ might be a *nontrivial puzzle*, if you try to prove it on account of a concrete definition of grp. However, we have found the defining expressions of $wordcount'$ immediately from (21) and (22), which in turn are shown equal by a simple application of law (18). It is here where we get the pay-off for working as long as possible at the "functional" level.