

Word count — the derivation

Maarten Fokkinga

Version of February 19, 2010, 10:42

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, $f = f_1 \cdot (f_2 \hat{+} f_3) \cdot f_4$ over $f(x) = f_1(f_2(f_4(x)) + f_3(f_4(x)))$, so that the replacement of $f_2 \hat{+} f_3$ by $f_3 \hat{+} f_2$ or by f' is easier to perform.

The list of items a, b, c, \dots , in that order, is denoted $[a, b, c, \dots]$. 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: $\text{tip } x = [x]$.

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:

$$\begin{aligned} id^* &= id && (\textit{id} \text{ is the identity function, } id\ x = x) \\ (f \cdot g)^* &= f^* \cdot g^* \\ f^* \cdot \# / &= \# / \cdot (f^*)^* \\ \oplus / \cdot \# / &= \oplus / \cdot (\oplus /)^* \\ \# / \cdot \# / &= \# / \cdot (\# /)^* \\ g \cdot \oplus / &= \otimes / \cdot g^* && \text{if } g \cdot (\oplus) = (\otimes) \cdot g \times g && \text{“promotion”} \end{aligned}$$

The last law has the 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.

The word count algorithm Lämmel [?] gives an elaborate discussion of the word count algorithm, resulting in an elaborate Haskell program in his Figure 1. Lämmel uses *types* to guide the construction of the word count algorithm as intended by Dean and Ghemawat [?]; in contrast we use the word count *specification* to derive the intended word count algorithm. Another difference is that all our programs are “one-liners”, which makes it easy to manipulate

them and do the derivation formally, whereas Lämmel’s formulation in Haskell takes 27 (?) lines plus some libraries. We think that the short notations like f^* and $\oplus/$ are of great help to understand what’s going on.

These entities are given:

Doc a set
 Key a set
 $Word$ a set
 $words$: $Doc \rightarrow [Word]$

We’ll pronounce ‘ $words\ d$ ’ as “the list of words in d ”. The problem is to produce from a given dictionary (f , say, of type $Key \rightarrow Doc$) a function (of type $Word \rightarrow \mathbb{N}$) which assigns to each word w the total number of occurrences of w in the dictionary’s documents:

$$\begin{aligned}
 wc & : (Key \rightarrow Doc) \rightarrow (Word \rightarrow \mathbb{N}) \\
 (1) \quad wc\ f\ w & = \text{the total number of occurrences of } w \text{ in the documents yielded by } f \\
 & = \sum_{k: \text{dom } f} \text{the number of occurrences of } w \text{ in } f\ k \\
 & = \sum_{k: \text{dom } f} size \cdot (=w) \triangleleft \cdot words \cdot f\ k
 \end{aligned}$$

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

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

- Dictionaries are represented by lists of argument-result pairs. Thus the type of the required wordcount function, now called wc' , is:

$$wc' : [Key \times Doc] \rightarrow [Word \times \mathbb{N}]$$

- The defining expression for wc' 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 show now right-away two versions of wc' , and we will later show how these definitions can be *derived formally* from the original specification. In order to get at a concise expression, we need a few auxiliary notations and concepts:

$(-, 1)$ = the function that maps x to $(x, 1)$
 $f \times g$ = the function that maps (x, y) to $(f\ x, g\ y)$
 exr = the function that maps (x, y) to y (“EXtract Right component”)
 $grp\ xs$ = a function that maps $[..., (w, n_1), ..., (w, n_2), ..., (w, n_k), ...]$ to $[..., (w, ns), ...]$
 where
 ns is such that $+/\ ns = +/[n_1, n_2, ..., n_k]$
 (for example, one may take $ns = [n_1, n_2, ..., n_k]$)

Function wc' can now be expressed as follows:

$$wc' = (id \times +/)^* \cdot grp \cdot \#/\ \cdot ((-, 1))^* \cdot words \cdot exr^*$$

Reading 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 $(w, [\dots, 1, \dots])$ -list;
- $(id \times +/\)^*$: replaces, in the list, each $(w, [1, \dots])$ item by $(w, +/[1, \dots])$.

Another way to understand the right-hand side, is to look at the types of the intermediate results. Abbreviating *Doc*, *Key*, *Word*, $[X]$ to D , K , W , L_X , the types read:

$$\begin{array}{cccccccccccccccc}
 & & (& id \times (+/ &) &)^* & \cdot & grp & \cdot & \#/\ & \cdot & (& (-, 1)^* & \cdot & words & \cdot & exr &)^* & \\
 \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\
 L_{W \times N} & & W \times N & & W \times L_{\mathbb{N}} & & L_{W \times L_{\mathbb{N}}} & & L_{W \times N} & & L_{L_{W \times N}} & & L_{W \times N} & & L_W & & D & & K \times D & & L_{K \times D}
 \end{array}$$

Showing also the labels that Lämmel uses, and also the second version where the reduce is partly performed as part of the initial map, we get:

$$\begin{aligned}
 (2) \quad wc' &= \overbrace{(id \times \boxed{+/\})^*}^{reducePerKey} \cdot \overbrace{grp}^{grpPerKey} \cdot \#/\cdot \overbrace{((-, 1)^* \cdot words \cdot exr)^*}^{mapPerKey} \\
 (3) \quad &= \underbrace{(id \times \boxed{+/\})^*}_{rEDUCE} \cdot \overbrace{grp}^{grpPerKey} \cdot \#/\cdot \overbrace{(\boxed{id \times +/\})^* \cdot grp}_{cOMPOSE} \cdot \overbrace{((-, 1)^* \cdot words \cdot exr)^*}_{mAP}
 \end{aligned}$$

Dean and Ghemawat designate *mAP* as ‘the map’ and *rEDUCE* as ‘the reduce’.

In the next paragraph we show how these expressions for wc' can be derived from the definition of wc .

The derivation Here we present a fully formal derivation of the MapReduce-style word-count 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 $W \rightarrow \mathbb{N}$ 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:

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

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:

$$\begin{aligned} toLf &= [(x, f x) \mid x \leftarrow \text{“domain of } f\text{”}; f x \neq 0] \\ toF &= \hat{+}/ \cdot mkFct* \end{aligned}$$

We have the following useful consequences:

- $toL \cdot toF$ is a kind of list normalization (it will occur frequently in the sequel)
- (4) $toF \cdot toL =$ the identity function id of type $W \rightarrow N$
- (5) $toF \cdot \#/= toF \cdot \#/= (toL \cdot toF)*$

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

$$\begin{aligned} & toF \cdot \#/= \\ = & \text{promotion } (\S, \text{ page 1}), \text{ since: } toF \cdot (\#)= (\hat{+}) \cdot (toF \times toF) \\ & \hat{+}/ \cdot toF* \\ = & \text{eqn (4): } id = toF \cdot toL \\ & \hat{+}/ \cdot (toF \cdot toL \cdot toF)* \\ = & \hat{+}/ \cdot toF* \cdot (toL \cdot toF)* \\ = & \text{promotion, as in the first line but now in the reverse direction} \\ & toF \cdot \#/= (toL \cdot toF)* \end{aligned}$$

This completes the preparation.

In order to *derive* claim (2) and (3) for wc' from definition (1) of wc , let us first play a bit with the defining expression (1) for $wc f w$:

$$\begin{aligned} & wc f w \\ = & \sum_{k: \text{dom } f} \text{size} \cdot (=w)\triangleleft \cdot \text{words} \cdot fk \\ = & \text{summation is a reduce: } \sum_{x: \text{dom } f} g(f x) = +/ \cdot (g \cdot \text{exr})* \cdot toLf \\ & +/ \cdot (\text{size} \cdot (=w)\triangleleft \cdot \text{words} \cdot \text{exr})* \cdot toLf \\ = & +/ \cdot (\text{size} \cdot (=w)\triangleleft)* \cdot \text{words}* \cdot \text{exr}* \cdot toLf \\ = & \text{promotion (see } \S, \text{ page 1): with } g, \oplus, \otimes = \text{size} \cdot (w=)\triangleleft, (\#), (+) \\ & \text{size} \cdot (=w)\triangleleft \cdot \#/= \cdot \text{words}* \cdot \text{exr}* \cdot toLf \\ (\dagger) = & \text{cata-UNIQ, elaborated below (this is an “eureka” step)} \\ & (\cdot w) \cdot toF \cdot (-, 1)* \cdot \#/= \cdot \text{words}* \cdot \text{exr}* \cdot toLf \\ = & (toF \cdot (-, 1)* \cdot \#/= \cdot \text{words}* \cdot \text{exr}* \cdot toLf) \cdot w \end{aligned}$$

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

$$(6) \quad wc = toF \cdot (-, 1)* \cdot \#/= \cdot \text{words}* \cdot \text{exr}* \cdot toL$$

Proof of step (\dagger) . 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.) We show this for $f = \text{size} \cdot (=w) \triangleleft$ and $g = (.w) \cdot \text{toF} \cdot (-, 1)^*$. First, we prove $f[x] = g[x]$:

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

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

$$\begin{aligned}
& f \cdot x \# y \\
&= \text{size} \cdot (=w) \triangleleft \cdot x \# y \\
&= (\text{size} \cdot (=w) \triangleleft \cdot x) + (\text{size} \cdot (=w) \triangleleft \cdot y) \\
&= f x + f y
\end{aligned}$$

and

$$\begin{aligned}
& g \cdot x \# y \\
&= (.w) \cdot \text{toF} \cdot (-, 1)^* \cdot x \# y \\
&= (.w) \cdot ((\text{toF} \cdot (-, 1)^* \cdot x) \hat{+} (\text{toF} \cdot (-, 1)^* \cdot y)) \\
&= ((.w) \cdot \text{toF} \cdot (-, 1)^* \cdot x) + ((.w) \cdot \text{toF} \cdot (-, 1)^* \cdot y) \\
&= g x + g y
\end{aligned}$$

End of proof of step (†)

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

$$(7) \quad wc' \cdot \text{toL} = \text{toL} \cdot wc$$

Consequently, for arguments of the form $f' = \text{toL} f$ we have:

$$\begin{aligned}
& wc' f' \\
&= wc' \cdot \text{toL} \cdot f \\
&= \text{specification (7) of } wc' \\
& \quad \text{toL} \cdot wc \cdot f \\
&= \text{just derived for } wc: \text{ eqn (6)} \\
& \quad \text{toL} \cdot \text{toF} \cdot (-, 1)^* \cdot \# / \cdot \text{words}^* \cdot \text{err}^* \cdot \text{toL} \cdot f \\
&= \text{toL} \cdot \text{toF} \cdot (-, 1)^* \cdot \# / \cdot \text{words}^* \cdot \text{err}^* \cdot f' \\
(8) \quad &= \text{toL} \cdot \text{toF} \cdot \# / \cdot (-, 1)^{**} \cdot \text{words}^* \cdot \text{err}^* \cdot f' \\
&= \text{law (5)} \\
(9) \quad & \text{toL} \cdot \text{toF} \cdot \# / \cdot (\text{toL} \cdot \text{toF})^* \cdot (-, 1)^{**} \cdot \text{words}^* \cdot \text{err}^* \cdot f'
\end{aligned}$$

So, defining wc' by (8) or (9), its specification (7) is satisfied. Next we aim at a concrete realization. To this end we *specify*(!) function grp by:

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

(We leave it to the industrious reader to find a suitable defining expression for grp .) Then we get from (8) and (9), and map distribution, the equations (2) and (3) for wc' that we were aiming at:

$$\begin{aligned} wc' &= (id \times +/)* \cdot grp \cdot \# / \cdot (\quad \quad \quad (-, 1)* \cdot words \cdot err)* \\ &= (id \times +/)* \cdot grp \cdot \# / \cdot ((id \times +/)* \cdot grp \cdot (-, 1)* \cdot words \cdot err)* \end{aligned}$$

Notice that the semantic equality of these two defining expressions for wc' might be a *non-trivial puzzle*, if you try to prove it on account of a concrete definition of grp . However, we have found the defining expressions of wc' immediately from (8) and (9), which in turn are shown equal by a simple application of law (5). It is here where we get the pay-off for working as long as possible at the “functional” level.