Google's MapReduce Programming Model — Revisited

Ralf Lämmel
Microsoft Corp., Redmond, WA

22 January, 2006, Draft

Abstract

MapReduce is a programming model (and an associated implementation) used at Google for processing large amounts of input data (such as millions of documents) and generating keyed or indexed query results (such as the indexes used for Google's web search). The programming model is stunningly simple, and it allows the programmer to largely abstract from the parallel and distributed execution of the data-processing computations. Parallel execution is promoted by the assumed skeleton of MapReduce computations. Load balancing, network performance and fault tolerance are taken care of by the MapReduce implementation.

We revisit the MapReduce programming model in an attempt to provide a rigorous description of the model. We focus on the key abstraction for MapReduce computations; this abstraction is parameterized by the problem-specific ingredients for data extraction and reduction. We use Haskell as a lightweight specification language to capture the essence of MapReduce computations in a succinct, executable and strongly typed manner. Our study substantiates that clarity, generality and correctness of designs (or the presentations thereof) are easily improved, if modest functional programming skills are put to work.

Keywords: Data processing, MapReduce, parallel programming, distributed programming, typed functional programming, map, reduce, list homomorphism, fold, unfold, Haskell.

Contents

1 Introduction

The paper discusses methods for improving the quality of software designs (and the presentations thereof). The reader may immediately think of design patterns or modeling languages like UML. By contrast, we discuss here the use of typed functional programming for capturing certain parts of a software design. We are not going to argue in favor of advanced formal specifications, which are certainly appropriate in designated areas; neither should the reader expect any sorts of advanced functional programming or any sorts of new theoretical results. Instead, we demonstrate the value of down-to-earth typed functional programming, when it is added to the repertoire of design methods. Typed functional programming is particularly good at validating 'design prose' that is centered around abstraction, composition, typing and algebraic properties. Our running example demonstrates that significant problems may go unnoticed when design prose is not subjected to the scrutiny of a design method like ours.

The paper is dedicated to the MapReduce programming model used at Google, as published in the paper "*MapReduce: Simplified Data Processing on Large Clusters*" [6] by Jeffrey Dean and Sanjay Ghemawat. MapReduce is a programming model (and an associated implementation) for processing large amounts of input data and generating keyed or indexed outputs in a parallel and distributed manner. For instance, Google's web search requires processing millions of documents and generating indexes from them. Thousands of MapReduce jobs are executed at Google every day on large clusters of commodity machines. The MapReduce programming model provides a stunningly simple but powerful enough abstraction for expressing data-processing computations, without requiring that the programmer engages in the complicated details of fault-tolerance, network performance, load balancing and other aspects of parallel and distributed execution.

Road-map

We are mostly concerned with MapReduce's abstraction for computations; we are much less concerned with the technicalities of the MapReduce implementation. In Section 2, we recall the informal MapReduce programming model, and we capture the key abstraction as a higher-order function defined by plain function composition. In Section 3 — Section 5, we study the typing of the MapReduce abstraction. In Section 6, we take a closer took at the recursion or iteration schemes that are at the heart of the MapReduce abstraction. In Section 7, we complete the strongly typed, executable specification of the MapReduce abstraction. In Section 8, we refine the key abstraction for MapReduce computations such that the key aspects of their parallel and distributed execution are modeled. In Section 9, we conclude the paper.

We record several 'lessons learned' on the way.

No profound knowledge of Haskell is required. All idioms are explained in footnotes.

2 The power of '.' and '⊥'

We quote the MapReduce programming model [6]:

> *"The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: map and reduce.*
>
> *Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the reduce function.*
>
> *The reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per reduce invocation. The intermediate values are supplied to the user's reduce function via an iterator. This allows us to handle lists of values that are too large to fit in memory."*

We also quote an example complete with pseudo-code [6]:

> *"Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:*

```
map(String key, String value):
 // key: document name
 // value: document contents
 for each word w in value:
  EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
 // key: a word
 // values: a list of counts
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

> *The map function emits each word plus an associated count of occurrences (just '1' in this simple example). The reduce function sums together all counts emitted for a particular word."*

We are not clearly told how the computation is composed together, but we can infer the necessary details from the quoted text, the example and extra hints. Most importantly, both map[1] and reduce are described to process key/value pairs one by one *independently*. So we assume that a MapReduce computation essentially *iterates* over the key/value pairs to apply map and reduce to each pair. This circumstance clearly hints at opportunities for parallelization.

In functional programming terminology, we may say that map and reduce need to be *mapped* (or perhaps folded) over the key/value pairs. We take for granted that

[1] We use sanserif style for MapReduce's map and reduce so that we do not confuse these arguments of a MapReduce computation with Lisp's higher-order combinators map and reduce. MapReduce terminology does not comply with functional programming standards.

a MapReduce computation can be decomposed and represented as a reusable Haskell function `mapReduce` as follows:[2]

```
mapReduce map reduce
   = reducePerKey reduce    -- 3. Apply reduce to each group
   . groupByKey             -- 2. Group intermediates per key
   . mapPerKey map          -- 1. Apply map to each key/value pair
```

We note that the basic skeleton of MapReduce computations is stunningly simple. We assume that `mapPerKey`, `groupByKey` and `reducePerKey` are components of the MapReduce library. We will discover the types and definitions of these components eventually. For now, we keep the functions undefined in our emerging specification:[3]

```
mapPerKey    = ⊥   -- to be discovered
groupByKey   = ⊥   -- to be discovered
reducePerKey = ⊥   -- to be discovered
```

This sort of undefinedness idiom allows us to (partially) check our specification at all times. That is, at least, we can check types. Of course, we have not yet specified *any* types, but Haskell takes care of type checking with full type inference anyway. Some readers may argue that a functional programming language with type inference offers only little convenience over modern mainstream programming languages. For instance, the undefinedness idiom has the following correspondence in C#:

```
public MyType MyMethod()
{
  throw new InvalidOperationException("undefined");
}
```

Here is a challenge: What is a reasonable Java or C# representation of the trivial Haskell function `mapReduce` given above? We are confident to claim that *there is none*. One would need at least the additional type information that follows only in the next section. Without such types, there is no proper way to transcribe our incomplete specification to type-checkable Java or C# code.

Lesson learned 1

This lesson is quite obvious but worth reiterating: The informal description of designs benefits from a machine-checked representation of the key abstractions, be it in the form of higher-order functions, subject to an identification of major building blocks, parameters and forms of composition. The added values are: clarity and partial correctness (due to type checks or other means of validation). ◇

Lesson learned 2

By taking advantage of type inference and the undefinedness idiom, one can perform partial checks on emerging or incomplete designs; at least type checking. This fully enables top-down design. It also enables validation of design prose for an implementation whose actual design is unattainable or non-presentable. ◇

[2] We are using Haskell 98 [13] throughout the paper. We recall details where necessary. One thing to note is that function composition "`.`" is defined as follows: $(g\,.\,f)\;x = g\;(f\;x)$. We also note that functions have names that begin in lower case, such as in `mapReduce`. Finally, line comments are prefixed by "--".

[3] The textual representation for Haskell's '⊥' (say, bottom) is 'undefined'.

3 Type discovery from prose

The type of a MapReduce computation needs to be properly discovered by us. Only the types of map and reduce are available, which we quote [6]:

> *"Conceptually the map and reduce functions [...] have associated types:*
>
> ```
> map (k1,v1) -> list(k2,v2)
> reduce (k2,list(v2)) -> list(v2)
> ```
>
> *I.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values. ".*

We also recall that "*the computation takes a set of input key/value pairs, and produces a set of output key/value pairs*" [6]. We will later discuss the tension between 'list' (in the types) and 'set' (in the wording). For now, we just continue to use 'list', as suggested by the above types. So let us turn prose into a type:[4]

```
computation :: [(k1,v1)] -> [(k2,v2)]
```

We feel confident about this discovery. If the quoted sentence has any suggestive meaning, then it is the one stated by the type. Let us convert the quoted types for map and reduce to Haskell notation, so that we operate in a single type system.

```
map    :: k1 -> v1 -> [(k2,v2)]
reduce :: k2 -> [v2] -> [v2]
```

Consequently, we get the following completed type for `mapReduce`:

```
-- To be amended!
mapReduce :: (k1 -> v1 -> [(k2,v2)]) -- The map function
          -> (k2 -> [v2] -> [v2])    -- The reduce function
          -> [(k1,v1)] -- A set of input key/value pairs
          -> [(k2,v2)] -- A set of output key/value pairs
```

Haskell sanity-checks this type for us. However, now that we can *see* the type, as opposed to reading about it, we can spot an issue. The type of `mapReduce` seems to suggest that a computation may produce *multiple* pairs with the same output key. Admittedly, the type expression `[(k2,v2)]` does not rule out such data anyhow, but this is not the point. The trouble is that reduce returns *a list* of output values per output key, whereas the ultimate result of `mapReduce` seems to lose the grouping per key. Even though the above type seems to follow from the given explanations, we contend that the following type should be preferred:

[4] The list type constructor is denoted by `[]` in Haskell. Lower case ingredients of type expressions in function signatures, such as `k1` and `v1` in the signature for `computation`, are type variables that facilitate polymorphic types; these type variables are implicitly universally quantified.

```
mapReduce :: (k1 -> v1 -> [(k2,v2)]) -- The map function
          -> (k2 -> [v2] -> [v2])    -- The reduce function
          -> [(k1,v1)]               -- A set of input key/value pairs
          -> [(k2,[v2])]             -- A set of output key/value-list pairs
```

Lesson learned 3

Types greatly help avoiding incorrect prose in communicating designs. ◇

4 Reflection on types and designs

We believe that readability (including succinctness) of types is essential for making them useful in reflection on designs. By further reflecting on `mapReduce`'s type, one easily arrives at a point where one asks: *Why do we need to require a list type for output values?* (For instance, the example for word-occurrence counting is phrased such that a *single* count is returned per word.) Also: *Why is the type of output values identical with the type of intermediate values?* We will indeed advise a generalization to resolve both issues.

Let us begin by remembering that the programming model was described such that "*typically just zero or one output value is produced per reduce invocation*" [6]. So the typical case would be covered by the following type:[5]

```
-- A special case
mapReduce :: (k1 -> v1 -> [(k2,v2)])  -- The map function
          -> (k2 -> [v2] -> Maybe v2) -- The reduce function
          -> [(k1,v1)]                -- A set of input key/value pairs
          -> [(k2,v2)]                -- A set of output key/optional value pairs
```

Here we assume that the use of `Maybe` allows the reduce function to express that *zero or one* output value is produced from the given intermediate values. Further, we assume that a key with the value `Nothing` as the result of reduction should not contribute to the final result of `mapReduce`. (Hence, we omit `Maybe` in the result type of `mapReduce`.) Instead of generalizing `Maybe v2` to `[v2]`, we introduce a new type variable `v3`:

[5] Haskell's `Maybe` type constructor models optional values; the presence of a value v is denoted by `Just` v, whereas the absence is denoted by `Nothing`.

```
-- The proposed generalization
mapReduce :: (k1 -> v1 -> [(k2,v2)])    -- The map function
          -> (k2 -> [v2] -> Maybe v3)   -- The reduce function
          -> [(k1,v1)]                  -- A set of input key/value pairs
          -> [(k2,v3)]                  -- A set of output key/value pairs
```

We can instantiate `v3` as follows:

- `v3` $\mapsto$ `v2` — We obtain the aforementioned (important) special case.
- `v3` $\mapsto$ `[v2]` — We obtain the original typing proposal — almost.

In the latter case, one is obliged to return `Nothing` for the proper eradication of a group as opposed to an ad-hoc test for the empty list. We prefer a consistent criterion for group eradication, which does not depend on the instantiation of `v3`. Some readers may argue that this type scheme goes beyond 'normal reduction' but this is the case anyhow, even for reduce's original type; we return to this issue in Section 6.

Let us reconsider the issue of 'lists vs. sets of key/value pairs'. We want to modify `mapReduce`'s type one more time to gain in precision of typing. It is clear that saying 'lists of key/value pairs' does not strictly imply uniqueness of keys for these pairs. The informal programming model used the word "set of key/value pairs" though [6]. However, this does not really resolve the issue. A strict reading of the term 'set' in this context merely states *irrelevance of order among the pairs in the set* and the property that the same key/value pair, e.g., $\langle$*key88*, *val42*$\rangle$, cannot occur several times. We rather need an association map or a dictionary or a finite map, to mention a few names of the concept. By using a dictionary type as opposed to a lists-of-pairs type, we make explicit where we commit to the data invariant "there is one key/value pair for each key at most". We revise the type of `mapReduce` one more time, where we leverage Haskell's library ADT for dictionaries, Data.Map:[6]

```
import qualified Data.Map          -- Library for dictionaries
type Dict k v = Data.Map.Map k v  -- Alias for dictionary type

mapReduce :: (k1 -> v1 -> [(k2,v2)])    -- The map function
          -> (k2 -> [v2] -> Maybe v3)   -- The reduce function
          -> Dict k1 v1    -- A key to input-value mapping
          -> Dict k2 v3    -- A key to output-value mapping
```

We only use a few operations on dictionaries. In the following list, we label the operations with total vs. partial with regard to the aforementioned data invariant:

- `toList` —- expose dictionary as list of pairs. (total)
- `fromList` —- construct dictionary from list of pairs. (partial)
- `empty` — construct the empty dictionary. (total)
- `insert` — insert key/value pair into dictionary. (partial)

[6] We introduce an alias `Dict` for `Data.Map.Map` because we are facing already all sorts of 'maps' in this paper. The Data.Map library is available online: http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Map.html.

- `insertWith` — insert with aggregating value domain. (total)
- `mapWithKey` — list map that preserves keys. (total)
- `filterWithKey` — filter dictionary according to predicate. (total)

(In an efficient implementation, `toList` and `fromList` would be refined to no-ops.)

Lesson learned 4

Systematic typing is helpful in reviewing the precision and the generality of designs. Clearly, for types to serve this purpose effectively, we are in need of a type language that is powerful and that allows for readable as well as succinct type expressions. ◇

As an aside, we do not argue in favor of generalizations that attempt to anticipate powerful new use cases. (This is perhaps a common critique of excessively advanced functional programming.) We rather argue in favor of generalizations that help with *simplification and normalization of designs*.

5 Type discovery vs. type inference

We are now in the position to discover the types of the library functions `mapPerKey`, `groupByKey` and `reducePerKey`. Despite Haskell's capability to perform type inference, there is no magic; we must discover these types actively. If we looked at the inferred types, we see that very polymorphic types are chosen due to the undefined right-hand sides:[7]

```
*Main> :t mapPerKey
mapPerKey :: a
*Main> :t groupByKey
groupByKey :: a
*Main> :t reducePerKey
reducePerKey :: a
```

We can actively discover the intended types by *starting from the back of the function composition* for `mapReduce` knowing that we can propagate the input type of the function. For convenience, we repeat the definition of `mapReduce`:

```
mapReduce map reduce
   = reducePerKey reduce
   . groupByKey
   . mapPerKey map
```

The type of `mapPerKey`'s first argument is clearly the type of map, and the type of its second argument is clearly the type of `mapReduce`'s input. So we have so much of `mapPerKey`'s type:

[7] In a Haskell interpreter session, we can let Haskell infer the type of an expression *exp* by entering "`:t` *exp*" at the prompt.

```
mapPerKey :: (k1 -> v1 -> [(k2,v2)])   -- The map function
          -> Dict k1 v1                -- A key to input-value mapping
          -> ? ? ?                     --  What's the result and its type?
```

The application of `mapPerKey` is supposed to return a list of intermediate key/value pairs. We assure ourselves that we need a list indeed; we cannot hope for a dictionary because any number of pairs with the same intermediate key may be produced by the various applications of map. (Recall the introductory example; think of the same word occurring several times.) So we can complete the type:

```
mapPerKey :: (k1 -> v1 -> [(k2,v2)])   -- The map function
          -> Dict k1 v1                -- A key to input-value mapping
          -> [(k2,v2)]                 -- The intermediate key/value pairs
```

We move slowly from the back to the front of `mapReduce`; thus, we obtain:[8]

```
groupByKey  :: [(k2,v2)]      -- The intermediate key/value pairs
            -> Dict k2 [v2]   -- The grouped intermediate values

reducePerKey :: (k2 -> [v2] -> Maybe v3)   -- The reduce function
             -> Dict k2 [v2] -- The grouped intermediate values
             -> Dict k2 v3   -- A key to output-value mapping
```

These types are quite telling and we should have few problems to inhabit these types.

Lesson learned 5

There is the established dichotomy for top-down vs. bottom-up design; both directions are often alternated in practice. A second (crosscutting) dichotomy concerns definitions vs. types. That is, one may first define the type of an abstraction, and later provide the definition, or vice versa. On the one hand, an interesting (non-trivial) type is likely to suggest useful ways of inhabiting the type. On the other hand, uninteresting (perhaps complicated) types are sometimes more easily inferred from interesting (perhaps less complicated) definitions. ◇

In Appendix A, we illustrate that the typed MapReduce abstraction (outlined so far) can be represented in a mainstream OO programming language. We recall that the untyped version from the previous section was not yet amenable to such a representation.

6 Getting the recursion schemes right

The MapReduce "*abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages*" [6]. Functional programming does indeed stand out when software designs can benefit from the employment of schemes for recursion or iteration over data. These schemes allow powerful forms of decomposition.

[8]The reader well versed in Haskell may have noticed that the actual types of all the helpers are more polymorphic than suggested by our explanations. That is, type variables are bound *per function signature*. Hence, our use of the same type variables only provides an illusion; see Appendix A for a remedy. The Haskell 98 extension for lexically scoped type variables [14] can also be put to work; cf. Section 8.

In particular, they allow for the separation of the generic scheme from the problem-specific ingredients. Also, the schemes typically suggests parallelism, if the problem-specific ingredients are free of side effects and meet some algebraic properties. We will now discover the precise correspondence between the map and reduce functions of the MapReduce programming model vs. existing idioms in functional programming.

6.1 Lisp's map and reduce

Due to the above-mentioned reference to Lisp, we should first recall the map and reduce combinators of Lisp — even though Lisp is untyped, and therefore not really suited for our purposes. We include the following explanations of the map and reduce combinators as they are described in "*Common Lisp, the Language*" [17]:[9]

`map result-type function sequence &rest more-sequences`

> *"The* `function` *must take as many arguments as there are sequences provided; at least one sequence must be provided. The result of map is a sequence such that element j is the result of applying function to element j of each of the argument sequences. The result sequence is as long as the shortest of the input sequences."*

We note that this combinator actually provides two concepts that can be easily separated: mapping over a single list while applying a given function to each element vs. zipping n lists as a list of n-tuples and perhaps uncurrying the function (so that the function takes a tuple as opposed to several curried arguments).

`reduce function sequence &key :from-end :start :end :initial-value`

> *"The reduce function combines all the elements of a sequence using a binary operation; for example, using + one can add up all the elements.*
>
> *The specified subsequence of the sequence is combined or "reduced" using the* `function`*, which must accept two arguments. The reduction is left-associative, unless the* `:from-end` `argument` *is true (it defaults to nil), in which case it is right-associative. If an* `:initial-value` *argument is given, it is logically placed before the subsequence (after it if* `:from-end` *is true) and included in the reduction operation.*
>
> *If the specified subsequence contains exactly one element and the keyword argument* `:initial-value` *is not given, then that element is returned and the* `function` *is not called. If the specified subsequence is empty and an* `:initial-value` *is given, then the* `:initial-value` *is returned and the* `function` *is not called.*
>
> *If the specified subsequence is empty and no* `:initial-value` *is given, then the* `function` *is called with zero arguments, and reduce returns whatever the function does. (This is the only case where the* `function` *is called with other than two arguments.)"*

[9] The relevant quotes are available on-line: http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node143.html.

We note that the fundamental notion of reduction is more streamlined [2]: a list of values is reduced to a single value of the same type by repeated applications of a binary operation; the empty list is reduced to a default value. The argument and result types of the binary operation are normally required to coincide with the element type, and the operation is normally required to be associative, with the 'default value' as unit. It is not too uncommon though to consider algebraically weaker and more flexibly typed forms of reduction.

6.2 Haskell's map and reduce

Haskell's `map` combinator processes a single list.
As an example of using `map`, in the following session we double all numbers in a list:

```
*Prelude> map ((*) 2) [1,2,3]
[2,4,6]
```

In Haskell, reduction is expressed in terms of the `foldl` combinator (or its strict companion `foldl'`), which defines a left-associative fold over a list. For instance, we can aggregate the sum of all integers in a list, using `foldl`:

```
*Prelude> foldl (+) 0 [1,2,3]
6
```

Haskell provides `map` and `foldl` in the Prelude.
The functions are easily defined by pattern matching on lists:[10]

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f b []     = b
foldl f b (a:as) = foldl f (f b a) as
```

We can restrict `foldl` to reduction by type specialization:

```
reduce :: (a -> a -> a) -> a -> [a] -> a
reduce = foldl
```

For the record, we mention that the combinators `map` and `foldl` are actually both instances of a more general and fundamental recursion scheme `foldr` for right-associative folds [12, 11]; the functions that are expressible in terms of `foldr` are also known as list catamorphisms or bananas.

[10] '[]' is the empty list constructor and ':' is the infix constructor for constructing a non-empty list from a head and a tail.

6.3 MapReduce's map and reduce

Equipped with this knowledge of map and reduce, we are ready to ask some questions:

- How do MapReduce's map and reduce correspond to folklore map and reduce?
- In particular:
  1. Does map perform a folklore map?
  2. Does map serve as the argument of a folklore map?
  3. Does reduce perform a folklore reduce?
  4. Does reduce serve as the argument of a folklore reduce?

Let us reconsider the sample code for the problem of counting occurrences of words:

```
map(String key, String value):
 // key: document name
 // value: document contents
 for each word w in value:
  EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
 // key: a word
 // values: a list of counts
 int result = 0;
 for each v in values:
  result += ParseInt(v);
 Emit(AsString(result));
```

It is clear that both functions are applied on a per-key basis and each application produces a result that is independent of any previous application. Indeed, we will later define `mapPerKey` and `reducePerKey` as variations on list map.

Fact: MapReduce's map and reduce are arguments of list maps.

Let us now consider the inner workings of MapReduce's map and reduce. The situation for reduce is straightforward only at first sight. The above-listed code is the perfect example of an imperative aggregation, which corresponds to a reduction, indeed. This seems to be the case for most MapReduce examples that are listed in [6].

Fact: MapReduce's reduce typically performs reduction.

In contrast with functional programming, the MapReduce programmer is not encouraged to identify the *ingredients of reduction* (i.e., an associative operation with its unit). We also must note that the original proposal for the type of reduce does not comply with the to-be-expected type of reduction, which should be a function of type `[v2] -> v2` as opposed to `[v2] -> [v2]`. It turns out that MapReduce's use of the term reduction is intentionally lax. Here is an example that shows the occasional deviation from 'normal reduction' [6]:

> *"Inverted index: The map function parses each document, and emits a sequence of ⟨word,document ID⟩ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a ⟨word, list(document ID)⟩ pair."*

(As an aside, we note that the sample prose commits to a mental type error by saying that "reduce [...] emits a [...] pair". This is a recurring issue. Again, validation of such prose by means of a machine-checkable representation would be easily feasible.) In

this example, reduce is supposed to perform sorting as opposed to reduction. In other examples, reduce may be supposed to filter. This observation about the behavioral flexibility of reduce further motivates our earlier generalization of `mapReduce`'s type. For instance, the generalization allows us to define a reduce function in terms of `foldl` — without restricting the type of `foldl` to the reduction scheme. In Section 8, we will revisit reduce in the context of discussing parallelization and distribution.

Fact: MapReduce's reduce occasionally deviates from reduction.

Let us now turn to map. In the example, map *splits up the input string into words*, and it iterates over these words to emit *one* intermediate value per word. A normal map *consumes* list shape and preserves it. By contrast, we face an operation that *introduces (or produces) list shape*.

Fact: MapReduce's map cannot be viewed as a normal list map.

Admittedly, the 'for-each' loop in the above code performs a map-like operation. However, it is crucial to notice that the mapping from a given word w to a pair $\langle w, \texttt{"1"} \rangle$ is a trivial helper step. The central part of MapReduce's map lies in the act of list introduction. In the example, this central part is concealed in the following code:

`word` w `in value`

A mapping from a value to a list of values (words) is not a map.
We may often define such a mapping in terms of the `foldl`/`foldr` combinators.
We may sometimes employ the notion of unfold — as discussed below.
In all remaining cases, we define the mapping as a free-wheeling recursive function.

6.4 The under-appreciated unfold

The recursion scheme for list producers is normally provided as the so-called `unfold` combinator for lists; the functions that are expressible in terms of unfold are also called anamorphisms or (concave) lenses [12, 7, 1]. The unfold combinator for lists is defined in Haskell as follows:[11]

```
unfold :: (u -> Either () (x,u)) -> u -> [x]
unfold f u = case f u of
               Left ()      -> []
               Right (x,u') -> x : unfold f u'
```

That is, the argument of `unfold` looks at the input at hand, and either decides to stop unfolding, or it returns the head of a non-empty list complete with the remaining input that can be recursively unfold to compute the tail. As an illustration, we define the `words` function in terms of unfold:[12]

[11] We use the `Either` type constructor for sums. A term of type `Either a b` is either of the form `Left x` (with `x` of type `a`) or of the form `Right y` (with `y` of type `b`). We use a `case` expression to discriminate on values of the `Either` type.

[12] The Haskell Prelude defines `words` in some style, but we reconstruct it here anyway in terms of `unfold`. We use some library functions: `dropWhile` keeps dropping the head of a list until the given predicate is no longer true. Hence, `dropWhile isSpace` removes all leading spaces. `span` splits up a given list into two segments as controlled by the given argument. The first segment contains the prefix for which the predicate was true, the second segment contains the remainder of the input.

```
words :: String -> [String]
words = unfold (split . dropWhile isSpace)
 where
  split u = case span (not . isSpace) u of
              ([],[]) ->  Left ()
              (w,u')  ->  Right (w,u')
```

Here is an illustrative Haskell session:

```
*Main> words "the under-appreciated unfold"
["the","under-appreciated","unfold"]
```

The alert reader may argue whether or not `words` is intrinsically an instance of the unfold notion, for one can define it in terms of `foldl`, too. This may be a reasonable choice indeed, but we reiterate the point that we cannot expect to encode the `words` function in terms of list map. Here is the `foldl`-based encoding:

```
words :: String -> [String]
words = reverse . snd . foldl transition (False,[])
 where
  transition (state, words) char =
   if state
     then if isSpace char
       then (not state, words)
       else (state, (head words ++ [char]) : tail words)
     else if isSpace char
       then (state, words)
       else (not state, [char]:words)
```

This specification essentially uses the fold operation to scan the input on a per-character basis, while it maintains a pair consisting of a state (in the sense of a deterministic, finite automaton) and the list of words encountered so far. There are two states: `False` — not currently scanning a word, `True` — the opposite.

7 Completion of the executable specification

We recall the key abstraction from Section 2:

```
mapReduce map reduce
   = reducePerKey reduce    -- 3. Apply reduce to each group
   . groupByKey             -- 2. Group intermediates per key
   . mapPerKey map          -- 1. Apply map to each key/value pair
```

All what is missing are these helper abstractions:

- `mapPerKey`
- `groupByKey`
- `reducePerKey`

The helper `mapPerKey` is really just a little more than the normal list map. That is, we first need to turn the dictionary into a plain list of pairs ('a no-op' — conceptually); then we map `mapPerKey`'s argument over this list; finally we concatenate the many intermediate result lists to one single list:[13]

```
mapPerKey f = concat            -- 3. Concatenate per-key lists
            . map (uncurry f)   -- 2. Map f over list of pairs
            . Data.Map.toList   -- 1. Turn dictionary into list
```

The helper `groupByKey` folds over all intermediate key/value pairs and builds a new dictionary keyed by the intermediate key type. Each entry can hold a list of intermediate values. Since the same key may be encountered several times, an aggregating insert operation is used to potentially extend the existing dictionary entry for a key by appending (cf. ++) the new value:

```
groupByKey = foldl insert Data.Map.empty
 where
  insert m (k2,v2) = Data.Map.insertWith (++) k2 [v2] m
```

Mapping MapReduce's reduce over the groups is essentially a key-aware list map. Some extra effort is necessary to eliminate groups that are reduced "away". (Remember the use of `Maybe` in the revised type of `reduce`.)

```
reducePerKey f = Data.Map.mapWithKey h      -- 3. Eliminate Maybe
               . Data.Map.filterWithKey g   -- 2. Filter Justs
               . Data.Map.mapWithKey f      -- 1. Apply reduce per key
 where
  g k2 Nothing  = False
  g k2 (Just _) = True
  h k2 (Just x) = x
```

Here, the operation `Data.Map.mapWithKey` is essentially the normal list map, and `Data.Map.filterWithKey` is essentially a predicated-controlled filter on a normal list. (The Haskell Prelude offers a general `filter` operation, indeed.) The definition looks more difficult than it should. Firstly, a single fold over the key/value pairs would be sufficient in principle, but `Data.Map` does not happen to provide an operation that combines list map and filter. Secondly, any instance of reduce that cannot possibly eradicate groups, would require only a single list map.

Now that we have worked out the helpers of `mapReduce`, the earlier type of `groupByKey` turns out to be too polymorphic. The use of `Data.Map.insertWith` implies that an explicit type of `groupByKey` needs to establish that intermediate/output keys can be compared. The Haskell type checker (here: GHC's type checker) readily tells us what the problem is and what to do:

> *No instance for* `(Ord k2)`
> *arising from use of* '`insert`' *at <file, line number, character position>.*
> *Probable fix: add* `(Ord k2)` *to the type signature(s) for* '`groupByKey`'.

[13] The argument of `mapPerKey` takes its two arguments one-by-one. Hence, we need to `uncurry` it, so that it can take a pair instead.

So we constrain the signature of `groupByKey`.[14]
(We also need to constrain the type of `mapReduce` like that, which is omitted here.)

```
groupByKey :: Ord k2 => [(k2,v2)] -> Dict k2 [v2]
```

Our Haskell specification is executable.
Here is the MapReduce computation for counting occurrences of words in documents:[15]

```
wordOccurrenceCount = mapReduce myMap myReduce
 where
  myMap    = const (map (flip (,) 1) . words)
  myReduce = const (Just . sum)  -- ... or use length!
```

Here is a main function and its demonstration:[16]

```
main = print
     $ wordOccurrenceCount
     $ Data.Map.insert "doc2" "appreciate the unfold"
     $ Data.Map.insert "doc1" "fold the fold"
     $ Data.Map.empty

*Main> main
{"appreciate":=1,"fold":=2,"the":=2,"unfold":=1}
```

8 Parallelism and distribution

We will now describe and specify conceptual issues of the parallelization and distribution of MapReduce computations. The MapReduce programmer may largely abstract from the parallel and distributed execution of MapReduce computations — even though not completely: the programming model comprises some extra arguments for controlling parallelization and distribution.

8.1 Opportunities for parallelism

The skeleton for MapReduce computations readily exhibits the following opportunities for parallelism, which we label for subsequent reference:

- list map: Conceptually, the use of a normal list map for processing input values and intermediate value groups implies that the corresponding steps `mapPerKey` and `reducePerKey` are amenable to a form of data parallelism [3, 16]. That is, in principle, a list map can be executed totally in parallel with regard to the elements of the processed list.

[14] The standard Haskell type class `Ord` comprises comparison functions. A polymorphic function signature is constrained by listing class constraints *cs* on the type variables before the actual function type t such as in '*cs* => *t*'.

[15] The uses of `const` express that the user functions of the computation do not observe the keys. The first argument of `mapReduce` splits up a document into words and then pairs each word with the constant '1'. Here, '(,)' is the pair constructor while `flip` inverts the parameter order of '(,)'.

[16] '$' denotes infix right-associative function application. (Recall normal function application, as in '$f\ x\ y$', is left-associative.) '$' saves us from extensive parenthesization.

- reduction: We assume here that reduce happens to define a proper reduction. Each application of reduce can be massively parallelized by computing subreductions in a tree-like structure while applying the associative operation at the nodes [2, 5, 15, 4, 8, 9]. If the binary operation is also commutative, then the order of combining intermediate results is negligible, thereby reducing synchronization constraints.

- homomorphism: The `mapPerKey` step, as a whole, is a list homomorphism [2] and thereby amenable to parallelism. It is a list homomorphism because it composes a list map and the application of `concat`, which is a reduction based on append (++) as associative operation.

- sort: The `groupByKey` step, which precedes the `reducePerKey` step, boils down to sorting on the intermediate key, for which various efficient parallel formulations exist, again including some that use list homomorphisms.

Of course, these simple insights do not suggest an evidently efficient parallelization and distribution on a specific architecture such as a cluster of commodity machines. The chief challenge is network performance: distributed processing of huge data sets requires specific insights as to manage the scarce resource *network bandwidth*.

8.2 The basic distribution strategy

Figure 1 depicts the overall strategy adopted by Google's MapReduce implementation [6]. Basically, input data is split up into pieces and intermediate data is partitioned (by key) so that these different pieces and partitions can be processed by different machines with local store. Here are the details:

- The input data is split up into M pieces to be processed by M map tasks, which are eventually assigned to worker machines. The number M is implied by a programmer-specified limit for the size of a piece. Hence, the `mapPerKey` step is *explicitly* parallelized [16] due to the explicit control on size or number of pieces. This parallelization is justified by the item list map above.

- There is a single master per MapReduce computation (not shown in the figure), which controls distribution such as the assignment of worker machines to tasks and the propagation of local filenames for remote download. The master also manages fault tolerance by pinging working machines, by re-assigning tasks for crashed workers, and by speculatively assigning new workers to compete with the slowest map tasks.

- Each worker for a map task downloads the relevant piece of input data for local processing. The results are locally stored, too. In fact, the results are readily stored in R partitions, where R is the number of programmer-specified reduce tasks. Hence, the `reducePerKey` step is *explicitly* parallelized [16] due to the explicit control on partitioning the key domain. This parallelization is justified by the item list map above. The local grouping essentially leverages the parallelism admitted for sorting (grouping); cf. the item sort above.

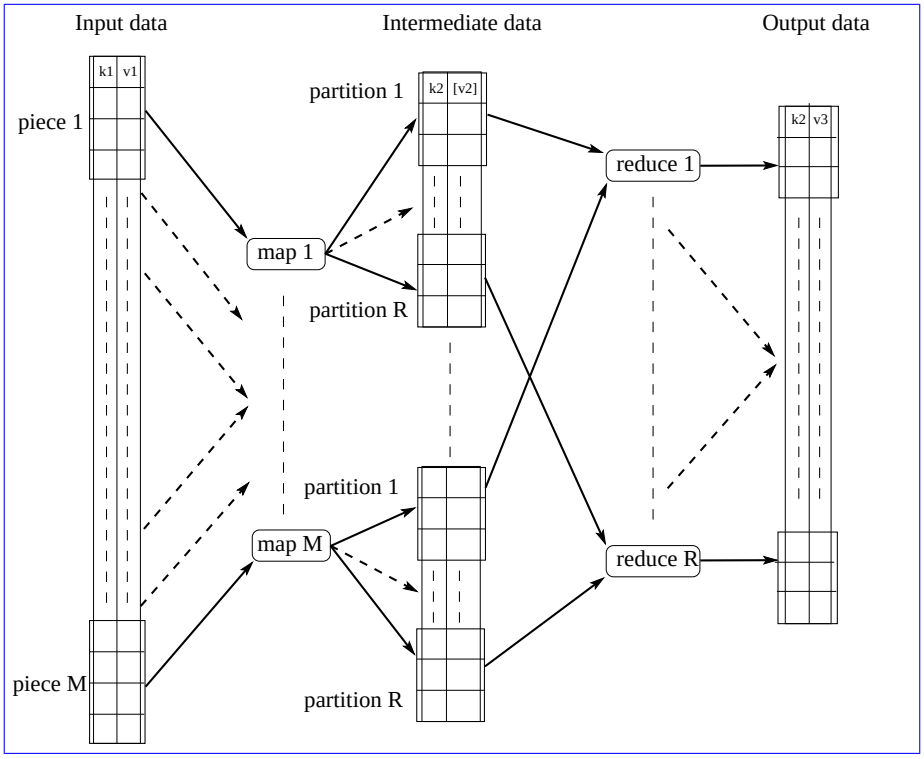

Figure 1: Splitting input data and partitioning intermediate keys

- Once all map tasks have been completed, the workers for the reduce tasks may download and merge the partitions of the intermediate results from the workers for the map tasks so that reduce can be applied to each partition. (We note that each partition is scattered over the workers for the map tasks.) The merging step is the completion of parallel sorting (grouping); cf. the item sort above.
- Finally, the results of the reduce tasks can be essentially concatenated.

8.3 Distributed reduction

There is one important refinement of the aforementioned basic distribution strategy. To reduce the volume of intermediate data to be transmitted from map tasks to reduce tasks, there is a new optional argument, combiner, which is a function "*that does partial merging of this data before it is sent over the network. [...] Typically the same code is used to implement both the combiner and reduce functions*" [6]. This refinement aims to leverage the reduction opportunity for parallelism that we listed above. We need to constrain reduce and combiner so that the distribution is correct, i.e., the end result of the computation does not depend on the number of map or reduce tasks. A sufficient condition for correctness is the following: reduce and combiner

indeed implement the same function, and they perform proper reduction based on an associative operation.

As an example, we consider counting word occurrences again. There are many words with a high frequency, e.g., 'the'. These words would result in many intermediate key/value pairs such as ⟨'the',1⟩. Transmitting all such intermediate data from a map task to a reduce task would be a considerable waste of network bandwidth. The map task may already combine all such pairs for each word in a single pair.

The distribution-enabled type of `mapReduce` looks as follows:[17]

```
mapReduce :: forall k1 k2 v1 v2 v3 v4. (Ord k1, Ord k2)
          => (v1 -> Int)  -- Size of input values
          -> Int          -- Split size for map tasks
          -> Int          -- Number of partitions
          -> (k2 -> Int)  -- Partitioning for keys
          -> (k1 -> v1 -> [(k2,v2)])     -- The map function
          -> (k2 -> [v2] -> Maybe v3)    -- The combiner function
          -> (k2 -> [v3] -> Maybe v4)    -- The reduce function
          -> Dict k1 v1                  -- Input data
          -> Dict k2 v4                  -- Output data

mapReduce size split parts keycode map combiner reduce
  = ... -- to be continued
```

The argument `size` defines a size function for input values; sizes are represented as `Int` values. The argument `split` defines a limit for the size of pieces of input data. (Remember: there are as many map tasks as pieces.) The argument `parts` defines the number of partitions for intermediate data (which equals the number of reduce tasks R). The argument `keycode` defines the partitioning function on the intermediate key domain; it is supposed to map keys to the range 1, ..., `parts`. The argument `combiner` defines the combiner function for reducing the data volume per map task. We give it the same general type as reduce (modulo type variables); the result type of combiner is the element type reduced by reduce. The intended optionality of combiner is modeled by admitting its definition as the identity function under the type specialization `v3` $\mapsto$ `[v2]` (and modulo the trivial embedding into `Maybe`). It is not difficult to observe that the result of a MapReduce computation does not depend on M and R, if one of the following conditions holds:

1. reduce is an arbitrary function; combiner is the identity function.
2. reduce is an arbitrary function; combiner performs a list map.
3. reduce and combiner perform the same proper reduction.
4. Like (3.) but combiner is pre-composed with a list map.
5. (3.) and (4.) but reduce is post-composed with an arbitrary function.

This enumeration does not come with any claim of completeness.

[17] We explicitly quantify the function signature (cf. ∀, i.e., 'forall'). Thereby, we can take advantage of a convenient Haskell 98 extension for lexically scoped type variables [14]. That is, we bind the type variables of `mapReduce`'s signature for further reference in signatures of local helper functions.

8.4 Executable specification

We owe to specify the new function `mapReduce`. The plan is not to go here as far as to specify a proper distributed application with the details of storage in local file systems and communication between the various tasks. Instead, we want to be explicit about splitting and partitioning. At the top-level, we may decompose MapReduce computations as follows:[18]

```
mapReduce size split parts keycode map combiner reduce =
     concatReducts        -- 9. Concatenate results
   . Prelude.map (
       reducePerKey       -- 8. Apply reduce to each partition
     . groupByKey )       -- 7. Group intermediates per key
   . mergeParts           -- 6. Merge scattered partitions
   . Prelude.map (
       Prelude.map (
          combinePerKey   -- 5. Apply combiner locally
        . groupByKey )    -- 4. Group local intermediate data
     . partition          -- 3. Partition local intermediate data
     . mapPerKey )        -- 2. Apply map locally to each piece
   . splitInput           -- 1. Split up input data into pieces
 where
 -- To be continued.
```

The factored applications of list map immediately express opportunities for parallelism. The two bold ones are indeed those list maps that are exploited for parallelization by Google's implementation as described earlier. The functions `mapPerKey`, `groupByKey` and `reducePerKey` are defined just as in the earlier version, even though they are applied more locally now. The function `combinePerKey` is just a clone of `reducePerKey`, except that it applies combiner for reduction as opposed to reduce. We need to go into details of the local functions for splitting input data (1.), partitioning intermediate data (3.) and merging partitions again eventually (6.).

The function for splitting the input essentially folds over the key/value pairs such that a new piece is started whenever the size limit has been reached for the current piece. Given a dictionary from input keys to input values, `splitInput` returns a list of such dictionaries.

```
splitInput :: Dict k1 v1 -> [Dict k1 v1]
splitInput =
   Prelude.map Data.Map.fromList       -- 4. Turn list of pairs into dictionary
 . Prelude.fst                         -- 3. Project away size of last piece
 . Prelude.foldl splitHelper ([[]],0)  -- 2. Splitting as a list fold
 . Data.Map.toList                     -- 1. Access dictionary as list of pairs
 where

  splitHelper :: ([[(k1,v1)]],Int)   -- Pieces so far with size of head
              -> (k1,v1)             -- The key/value pair to be considered
              -> ([[(k1,v1)]],Int)   -- New set of pieces and size of head
```

[18] We systematically qualify all library functions for the reader's convenience.

```
  splitHelper (ps,s) x@(k1,v1) =
    if size v1 + s < split || Prelude.null (Prelude.head ps)
      then (((x:Prelude.head ps):Prelude.tail ps), size v1 + s)
      else ([x]:ps,size v1)
```

The function for partitioning intermediate data refers to `parts` (number of reduce tasks R) and `keycode` (the partitioning function for the intermediate key domain). For each partitioning code (i.e., for each value in 1 .. `parts`), the relevant intermediate key/value pairs are extracted and placed in a separate list:

```
partition :: [(k2,v2)] -> [[(k2,v2)]]
partition all = Prelude.map partitionHelper [1..parts]
 where
  partitionHelper :: Int -> [(k2,v2)]
  partitionHelper p = Prelude.filter
                        ( (==) p
                        . keycode
                        . Prelude.fst) all
```

Prior to the normal application of `reducePerKey`, we need to unite the scattered contributions to each partition as they are hold by the workers for the various map tasks. To this end, we need to transpose the map-task-biased grouping to the reduce-task-biased grouping. (The term transposition refers to the same concept in matrix manipulation.) That is:

```
  mergeParts :: [[Dict k2 v3]] -> [[(k2,v3)]]
  mergeParts =
   Prelude.map ( Prelude.concat          -- 3. Unite partition
       . Prelude.map Data.Map.toList)    -- 2. Access dictionaries
   . Data.List.transpose                 -- 1. Transpose grouping
```

We refer to the paper's web site for the full specification. One may actually argue whether or not the amount of explicit configuration is necessary (cf. `size`, `split`, `parts`, `keycode`). For instance, it is not easy to see why a MapReduce programmer would want to define a problem-specific size limit; it seems that the computing power of the given architecture and on overall fairness constraint for simultaneously executed MapReduce computations are more meaningful controls. The paper's web site also comprises a specification that illustrates MapReduce computations without any configuration burden. The following assumptions underly this advanced approach: (i) The MapReduce computing cluster defines the size limit for pieces of input data to be processed by workers on map tasks; (ii) the same limit (or another general limit) also applies to the size of partitions for the workers on reduce tasks; (iii) the number of partitions is determined once the map tasks have completed their tasks by summing up the size of intermediate values; (iv) the intermediate key domain implements a partitioning function that is parameterized by the number of partitions. Among these assumptions, the only debatable one seems to be (iii) because it restricts the interleaved operation of map and reduce tasks.

9 Conclusion

The original formulation of the MapReduce programming model seems to stretch some of the established terminology of functional programming. Nevertheless, the actual assembly of concepts and their evident usefulness in practice is an impressive testament to the power of functional programming primitives for list processing. It is also quite surprising to realize (for the author of the present paper anyway) that the relatively restricted model of MapReduce fits so many different problems as encountered in Google's problem domain. This insight sheds some new light on distributed programming as a whole. We believe that our analysis of the MapReduce programming model helps with a deeper understanding of the ramifications of Google's results.

We have used the strongly typed functional language Haskell for the discovery of the rigorous description of the programming model. Thereby, we have substantiated, once again: functional programming is an excellent specification tool in the context of exploring designs that can benefit from functional composition, higher-order combinators and type inference. (This insight has been described more appropriately by Hughes, Thompson, and surely others [10, 18].) We have shared all the lessons learned in the MapReduce exercise so that others may receive new incentives for the deployment of functional programming as a design tool for their future projects, be in the context of distributed computing, data processing, or elsewhere.

References

[1] L. Augusteijn. Sorting morphisms. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Program ming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, Sept. 1998.

[2] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] W.-N. Chin, J. Darlington, and Y. Guo. Parallelizing conditional recurrences. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing, Vol. I, EURO-PAR'96 (Lyon, France, August 26-29, 1996)*, volume 1123 of *LNCS*, pages 579–586. Springer-Verlag, 1996.

[5] M. Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Technical Report CSR-25-93, Department of Computer Science, University of Edinburgh, 1993.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04, 6th Symposium on Operating Systems Design and Implementation, Sponsored by USENIX, in cooperation with ACM SIGOPS*, pages 137–150, 2004.

[7] J. Gibbons and G. Jones. The under-appreciated unfold. In *ICFP '98: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, New York, NY, USA, 1998. ACM Press.

[8] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing, Vol. I, EURO-PAR'96 (Lyon, France, August 26-29, 1996)*, volume 1124 of *LNCS*, pages 401–408. Springer-Verlag, 1996.

[9] Z. Hu, H. Iwasaki, and M. Takeichi. Construction of list homomorphisms by tupling and fusion. In W. Penczek and A. Szałas, editors, *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science, MFCS'96 (Cracow, Poland, September 2-6, 1996)*, volume 1113 of *LNCS*, pages 407–418. Springer-Verlag, 1996.

[10] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

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

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Proceedings of 5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, Berlin, 1991.

[13] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

[14] S. Peyton Jones and M. Shields. Lexically scoped type variables, Mar. 2004. Available at `http://research.microsoft.com/Users/simonpj/papers/scoped-tyvars/`.

[15] D. B. Skillicorn. *Foundations of Parallel Programming*. Number 6 in Cambridge Series in Parallel Computation. Cambridge University Press, 1994.

[16] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[17] G. L. Steele, Jr. *Common Lisp: the language*. Digital Press, second edition, 1990.

[18] S. Thompson. *The Craft of Functional Programming*. Addison Wesley, 1996. 2nd edition in 1999.

A MapReduce computations in C#

We will now represent the MapReduce types and the overall functional decomposition for MapReduce computations in C#. This is an exercise in comparing the specification capabilities of a mainstream OO language and a pure, higher-order functional with type inference. To prepare the C# encoding, we provide a more OO-enabled Haskell encoding of sections 4 and 5. That is, we group user functions (map and reduce) as well as library functions (`mapPerKey` etc.) in designated record types that are explicitly parametric in the corresponding key and value types.

The following code is self-contained. For brevity, we do not re-include the implementations of the helpers that we gave in Section 7; we use the undefinedness idiom, again. (As a side note, the record-based encoding properly restricts the polymorphism of all involved functions, and it is therefore more precise than the original Haskell encoding.)

```
import Prelude hiding (map)        -- Name clash with normal list map
import qualified Data.Map          -- Library for dictionaries
type Dict k v = Data.Map.Map k v  -- Alias for dictionary type

data MapReduceUser k1 k2 v1 v2 v3 = MapReduceUser {
 map    :: k1 -> v1 -> [(k2,v2)],
 reduce :: k2 -> [v2] -> v3
}

data MapReduceLibrary k1 k2 v1 v2 v3 = MapReduceLibrary {
  mapReduce :: MapReduceUser k1 k2 v1 v2 v3 -> Dict k1 v1 -> Dict k2 v3
, mapPerKey  :: (k1 -> v1 -> [(k2,v2)]) -> Dict k1 v1 -> [(k2,v2)]
, groupByKey :: [(k2,v2)] -> Dict k2 [v2]
, reducePerKey :: (k2 -> [v2] -> v3) -> Dict k2 [v2] -> Dict k2 v3
}

mapReduceLibrary = MapReduceLibrary {
  mapReduce    = \user -> reducePerKey mapReduceLibrary (reduce user)
                        . groupByKey mapReduceLibrary
                        . mapPerKey mapReduceLibrary (map user)
, mapPerKey    = ⊥ -- omitted for brevity; see Section 7
, groupByKey   = ⊥ -- omitted for brevity; see Section 7
, reducePerKey = ⊥ -- omitted for brevity; see Section 7
}
```

In transcribing the Haskell specification to C#, we aim to be as 'structure-preserving' as possible. In particular, the Haskell record type `MapReduceUser` is transcribed to an C# interface, while the library functionality is gathered in a static (and sealed) class with only static methods (i.e., 'functions'). We are able to reuse an existing collection type for dictionaries. We provide a (trivial) `Pair` class (since the .NET 2.0 library does not have one readily). We are forced to introduce delegate types so that we are able to express the higher-order types of `MapPerKey` and `ReducePerKey`. Again, we put the undefinedness idiom to work so that we omit the routine implementations of the helpers. We leave it to the reader's judgment to assess the merits of the Haskell and C# encodings.

```
using System;
using System.Collections.Generic;
```

```
namespace MapReduceLibrary
{

    public class Pair<a, b>
    {
        public a fst;
        public b snd;
    }

    public interface MapReduceUser<k1, k2, v1, v2, v3>
    {
        IEnumerable<Pair<k2, v2>> Map(k1 key, v1 value);
        v3 Reduce(k2 key, IEnumerable<v2> iterator);
    }

    public sealed class MapReduceLibrary<k1, k2, v1, v2, v3>
    {

        public static Dictionary<k2, v3> MapReduce(
                      MapReduceUser<k1, k2, v1, v2, v3> user
                    , Dictionary<k1, v2> input)
        {
            return ReducePerKey(user.Reduce,
                        GroupByKey(
                                MapPerKey(user.Map, input)));
        }

        delegate IEnumerable<Pair<k2, v2>> MapType(k1 key, v1 value);
        delegate v3 ReduceType(k2 key, IEnumerable<v2> iterator);

        static List<Pair<k2, v2>> MapPerKey(
               MapType map
             , Dictionary<k1, v2> input)
        {
            throw new InvalidOperationException("undefined");
        }

        static Dictionary<k2, List<v2>> GroupByKey(
               List<Pair<k2, v2>> intermediate)
        {
            throw new InvalidOperationException("undefined");
        }

        static Dictionary<k2, v3> ReducePerKey(
               ReduceType reduce
             , Dictionary<k2,List<v2>> grouped)
        {
            throw new InvalidOperationException("undefined");
        }

    }

}
```