Using CalcuList To MapReduce Json Documents

(DISCUSSION PAPER)

Domenico Saccà and Angelo Furfaro

DIMES Dept, Università della Calabria, 87036 Rende, Italy
sacca@unical.it, a.furfaro@unical.it

Abstract. CalcuList (*Calcu*lator with *List* manipulation), is an educational language for teaching functional programming extended with some imperative and side-effect features, which are enabled under explicit request by the programmer. As the language natively supports json objects, it may be effectively used to implement generic MapReduce, which is a popular model in distributed computing that underpins many NoSQL systems. As a list a jsons can be thought of as a dataset of a document NoSQL datastore, it turns out that CalcuList can be used as a tool for teaching advanced query algorithms for document datastores such as MongoDB and CouchDB.

Keywords: MapReduce, Document Databases, Query Languages, Functional Language, Imperative Features

1 Introduction

Pure functional programming designates a functional programming paradigm that treats all computation as the evaluation of mathematical functions. This paradigm forbids changing-state and mutable data so that functions will only depend on their arguments, regardless of any global or local state (i.e., it has no *side effects*).

An important member of the purely functional languages is Haskell [5, 4], which provides relevant features including polymorphic typing, static type checking, lazy evaluation and higher-order functions. Other functional languages are less pure (*impure*), as they contain imperative features. For instance, most of the languages of the Lisp Family [14] were designed to be multi-paradigm, mainly because the use of side-effects, in some cases, simplifies the developed code or even results in a better computational efficiency.

CalcuList (*Calcu*lator with *List* manipulation) is an *educational* programming language for teaching the functional paradigm, suitably extended with

The full version of this paper is published in [11] and can be downloaded as an open access publication from https://dl.acm.org/citation.cfm?doid=3216122.3216164

some imperative features involving side effects [10]. Thus, CalcuList essentially belongs to the family of "impure" functional languages but it may be use as a *pure* functional language, in that side effects are required to be explicitly enabled.

CalcuList syntax is rather similar to that of Python [13] and natively supports processing of strings, lists and json objects. Json (*JavaScript Object Notation*) [3] is language-independent data interchange format, which has been originally introduced as a subset of the JavaScript scripting language and is now widespread in many applications, e.g. web-services, and support for it has been added to the standard libraries of many programming languages.

The language is strongly typed, but type checking is mainly dynamic as most of the type checking is done at run-time. An interactive computation session with the user is established by means of a REPL (*Read-Evaluate-Print-Loop*) shell. During a session, a user may define a number of global variables and functions and run suitable computations on them as queries, that are computed and displayed on the fly. CalcuList expressions and functions are first compiled and then executed each time a query is issued.

In this paper we shall illustrate powerful features of CalcuList to support *MapReduce*, which is a programming model for processing and generating large data sets, inspired by the functions map and reduce commonly used in functional programming and introduced by Google to support parallel computations on large data sets spread over clusters of computers. Nowadays MapReduce is considered a popular "Data-Oriented" model in distributed computing that processes data simultaneous data sources in two primary steps: *Map* and *Reduce*. The first step maps the input data into intermediate data sets and the second step aggregates the intermediate results in a consolidated result.

MapReduce is mainly used to implement applications for a *document store* (also called document database or document-oriented database), which is a subset of a type of NoSQL database systems that have been released and widely adopted in many domains poorly served by relational databases [7]. A document store is used for storing, retrieving, and managing semi-structured data, mainly organized as jsons. In this paper we show how to use CalcuList to implement MapReduce programs on jsons and we argue that suitable interfaces could make CalcuList a powerful query environment for popular document stores such as CouchDB [12] and MongoDB [1].

The remainder of this paper is organized as follows. Section 2 introduces basic notions about the CalcuList functional core, illustrates higher-order functions and their usage to define MapReduce queries, and describes the imperative aspects of the language and their usage for some situations where they may simplify the definition of MapReduce code or even result in a better computational efficiency. Section 3 focus on the manipulation of json structures in CalcuList and on the usage of higher-order functions to implement MapReduce queries on a document datastore represented bya list of jsons. Finally, Section 4 draws the conclusions and discusses future work.

2 An Overview of CalcuList

The main interface to CalcuList, like other languages (e.g. Python, Scala), is a REPL environment where the user can define functions and issue valid expressions (queries in CalcuList), which in turn are parsed, evaluated and printed before the control is given back to the user.

The basic types for CalcuList are six: (1) *double*, (2) *int* , (3) *char*, (4) *bool* (with values *true* or *false*), (5) *null* (that has a unique value named `null` as well) and (6) *type* (whose values are all the other types: *double*, *int*, etc.).

Three compound types are supported: string, list and json. A string in CalcuList is an immutable sequence (possibly empty) of characters – two strings can be concatenated by the overloaded operator `+` returning a new string and slice operators à la Python are available to extract a substring. Jsons will be described in Section 3. Next we provide some insights on lists.

The *list* is a powerful compound data type used by CalcuList for constructing dynamic data structures and implementing recursive algorithms. A list `L` consists of a number (possibly zero) of comma-separated elements between square brackets. As in Python, the elements can be of any type, including list and json, and heterogeneous. They are numbered with an index starting from zero: the first element (called the *head* of the list) is `L[0]` (also denoted simply by `L[.]`), the second element is `L[1]` and so on.

A list can be extended by adding additional comma-separated elements on top, followed by the append operator (the bar "`|`") - the syntax has been inspired by Prolog [2]. For instance, given a list `L`, `M=[x,y|L]` extends `L` by adding two new elements, `x` and `y`, on top of `L`. Another operator on a list `L` is `L[>]`, which returns the (possibly empty) *tail* of `L`, i.e., the list starting from the element `L[1]`. Slice operators on a list `L` (say with n elements) are deep in the sense that they clone the elements. In particular, `L[:]` clones the whole list `L` whereas `L[i:]`, $\texttt{L[i}_1\texttt{:i}_2\texttt{]}$ and `L[:i]` clone suitable sublists.

The syntax for an expression is like in Java. An expression e is either *simple* or *conditional* with format e_1 `?` e_2 `:` e_3, where e_1 is a logical (i.e., with type *bool*) expression and e_2 and e_3 are expressions (possibly conditional in their turn). The meaning of a conditional expression is: its value is equal to the value of e_2 if e_1 evaluates to *true* or, otherwise, to the value of e_3.

The language allows the user to define global variables. A *global variable* V is defined as "$V = e$", where e is an expression: the type of V is inferred from the type of e and its value is kept until a later re-definition, which may change also the type. A *query* is an expression e prefixed by the `^` symbol and prints the value of the expression at the current state.

A function is defined by giving it a name, followed by: its comma-separated parameter names (included in parentheses), the colon symbol "`:`" and the (typically conditional) expression that computes the value of the function. Parameters as well as the return value of a function are defined without specifying types for them, so that static type checking is performed only for constant operands. A function is compiled while it is defined and, as for Python, the type checking is done at run time, after its call, when all the types become available.

No side effects are allowed in the basic definitions of functions, i.e., functional operations do not modify current global variables and always create new data objects. Differently from Python, global variables are not accessible inside a function so that a possible change of state does not affect the function behavior.

```
>>member(x,L): L !=[] && (x==L[.] || member(x,L[>]));
>>sumL(L) : L==[]? 0: L[.]+sumL(L[>]);
>>range(x1,x2): x1>x2? []: [x1|range(x1+1,x2)];
>>R16= range(1,16); /* global variable */
>>^sumL(R16); /* compute the sum of the elements in R16*/
136
```

Fig. 1. *Functions on Lists*

Figure 1 presents a work session with CalcuList that consists of a number of basic functions for handling lists. The first one is the classical `member` function that checks wether an element belongs to a list. The `sumL` function sums the elements of a list and the `range` function generates the lists of the integers in a given range. Next, a list R16 with elements in the range from 1 to 16 is defined and the query to compute the sum of all such elements is issued.

2.1. Higher-Order Functions and MapReduce Definitions

CalcuList supports higher-order functions since a function parameter can also be a function and a function may return a function. A function parameter `f` is written as `f/`n, where n is the arity of the function `f`. In addition, adding `/`n after a function head `g(...)` prescribes that the function `g` must return a function with arity n.

Some examples of higher-order functions implementing MapReduce computations are presented in Figure 2. The function `map` receives a list `L` and two functions as parameters, both with arity 1: `f` checks whether an element has a certain property and, in case the test succeeds, `m` maps the element into another value. The function `map` scans all elements of `L`, for each of them it performs the possible mapping and eventually returns the list of the results. The first map query filters all numbers divisible by 2 or by 3 (see function `d2or3`) in the range from 1 to 10 and replaces each of these values with their cube (see the lambda function) – let M denote the result of this query. Note that lambda functions are written in Python like syntax.

The subsequent function `reduce` receives a list `L`, a function parameter `f` (mapping two values into another value) and an initial value that is returned at final stage of the recursion, when `L` reduces to an empty list. Then `reduce` applies `f` to two actual parameters: the list head and the result of `reduce` for the list tail. Therefore, the list is eventually reduced to a single value by recursively applying `f` from right to left. The first reduce query computes the sum of all elements in M (the list returned by `map`), whereas the second one computes the product of the squares (see second lambda function parameter) of all integers in the range from 1 to 10 that are divisible both by 2 and by 3.

```
>>map(L,f/1,m/1) : L==[]?[]: f(L[.])?
   [m(L[.])|map(L[>],f,m)]:  map(L[>],f,m);
>>d2or3(x) : x%2==0 || x%3==0;
>>^map(R16,d2or3,lambda x: x*x*x);
[ 8, 27, 64, 216, 512, 729, 1000, 1728, 2744, 3375, 4096 ]
>>reduce(L,f/2,in): L==[]? in: f(L[.],reduce(L[>],f,in));
>>sum(x,y):x+y;
>>^reduce(map(range(1,10),d2or3, lambda x: x*x*x),sum,0);
14499
>>^reduce(map(range(1,10),lambda x:x%2==0&&x%3==0,
   lambda x:x*x),lambda x,y: x*y,1);
5184
```

Fig. 2. *High-Order Functions and MapReduce Definitions*

2.2. Imperative Features of CalcuList

CalcuList has a *pure* functional programming core but it also allows the programmer to define a function with side effects by adding `*` next to its name (*star function*). In this way, the usage of global variables inside a function is enabled (provided that they are explicitly listed in its definition body) and they may have assigned new values during the function execution, as in an imperative programming language. Actually also parameter values can be modified during the execution of a star function.

Global variables and parameters can be updated inside a function by means of the so-called *global setting commands* (GSC). A *GSC* is an assignment of an expression value to a labeled global variable or to a function parameter and represents an imperative statement with side effects. GSCs can be inserted both before and after the expression defining the star function. In case of a conditional expression, GSCs may be also inserted before and after each sub-expression. The usage of GSCs has been inspired by the semantic rules of an Attribute Grammar [6].

An example of star function is shown in Figure 3. The function is `reduceCount` that takes a list L of possibly duplicated elements and returns a list LC of pairs $[e, n]$, where e is an element in L and n is the number of occurrences of e in L. The subsequent query returns the list of characters and their numbers of occurrences in the string `"Hello_World"`. Note that `"Hello_World"@list` transforms the string in a list of characters.

3 Definitions of Jsons and Manipulation of Json Lists

CalcuList natively supports json objects (referred to simply as *json*), which are at runtime represented as (possibly empty) sequences of fields separated by comma and enclosed into curly braces. A field is a pair (key, value) separated by a colon: *key* is a string and *value* can be of any type, including json.

```
>>rc1*(_,_) : null;
>>rc2*(L,M,C) : M==[]? rc1(L[>],[[L[.],1]|C]):
  L[.]==M[.][0]?{!M[0][1]+=1!} rc1(L[>],C): rc2(L,M[>],C);
>>rc1*(L,M) : L==[]? M: rc2(L,M,M);
>>reduceCount*(L) : rc1(L,[]);
>>^reduceCount("Hello_World"@list);
[ [ 'H', 1 ], [ 'e', 1 ], [ 'l', 3 ], [ 'o', 2 ],
>>^reduceCount(map(R16,lambda x: true,
  lambda x: x%2==0? "even": "odd"));
[ [ "even", 8 ], [ "odd", 8 ] ]
```

Fig. 3. *Star Functions*

```
>>emps = [ { "name": "e1", "age": 30 },
 {"name":"e2","age":32,"proj":["p1","p2"],"bonus":10},
 {"name":"e3", "age":28, "proj":["p1","p3"] } ];
>>^emps[2]["proj"];
[ "p1", "p3" ]
>>^emps[0]["proj"] %*;
null
```

Fig. 4. *Json Definition and Manipulation in CalcuList*

In Figure 4 we define a variable `emps` as a list of three jsons, each representing an employee. For instance, `emps[1]` is the employee with name `"e2"`, aged 32 and working for the projects `"p1"` and `"p2"`. Given a json J and a key k, $J[k]$ denotes the value of the field of J with key equal to k – if there is no such a field then the value *null* is returned as it happens for `^emps[0]["projects"]` (the string `%*` next to the query is a display option to force the writing of the null value). Given a value v, $J[k] = v$ modifies the value for the field if J includes the key k or otherwise, J is extended with a new field with key k and value v.

Figure 5 presents a high-order function to filter the jsons that satisfy some conditions in one of the fields: `jsFilter`, whose filtering condition parameter is expressed by a function `filterC`, that receives the current json and a field, say with key K and value V, to be used for the evaluation. Next, the figure shows two implementations for `filterC`: (1) `selKV`, which checks whether the json has a field equal to (K, V), and (2) `selKinV`, which checks whether the value V is included in the list of elements that represents the value for the the field K in the json. Two queries on the variable `emps` defined in Figure 4 are issued: the first one filter all employees with age 28 and the second one the employees that work for at least the project `"p1"`.

Figure 5 also presents a classical map function that scans all employees to produce the list of all projects for which they work – note that, by adding the option `[:]` to `E[.]["proj"]`, the project lists are cloned to avoid their

```
>>jsFilter(LJ,filtC/3,K,V):LJ==[]?[]:filtC(LJ[.],K,V)?
    [LJ[.]|jsFilter(LJ[>],filtC,K,V)]:
    jsFilter(LJ[>],filtC,K,V);
>>selVeqK(J,K,V): J[K]!=null && J[K]==V;
>>^jsFilter(emps,selVeqK,"age",28);
[ {"name":"e3",  "age":28, "proj":["p1","p3"] } ]
>>selVinK(J,K,V) : J[K]!=null && member(V,J[K]);
>>^jsFilter(emps,selVinK,"proj","p1") ;
[{"name":"e2","age":32,"proj":["p1","p2"],"bonus":10},
 {"name":"e3","age":28,"proj":["p1","p3"] } ]
>>mapProj(E) : E==[]? []: E[.]["proj"] != null?
    E[.]["proj"][:]+mapProj(E[>]): mapProj(E[>]);
>>^reduceCount(mapProj(emps));
[ [ "p1", 2 ], [ "p2", 1 ], [ "p3", 1 ] ]
```

Fig. 5. *Higher Order Functions on Jsons*

coalescence. The query calling the function `reduceCount`, defined in Figure 3, reduces the list by storing each project once as a pair: project name and number of its occurrences (i.e., the total number of employees working for it).

Figure 6 includes a star function that updates a list of jsons. As a list of jsons can be though of as a collection in a documentary database, star functions have the crucial role of updating the database. As an example we have written the function `giveBonus` that assigns a bonus of a given amount to all employees in a list – the amount is incremented if an employee already has a bonus. Obviously the function has side effects as it updates the elements in the parameter `emps`. We present a query that assigns a bonus of 100 Euro to all employees in the global variable `emps` (defined in Figure 4) who work in project `p1`.

```
>>aBonus*(E,v):E==[]?true:E[.]["bonus"]==null?
    {! E[0]["bonus"]=v !} giveBonus(E[>],v):
    {! E[0]["bonus"]+=v !} giveBonus(E[>],v);
>>^aBonus(jsFilter(emps,selVinK,"proj","p1"),100);
true
>>^emps;
[{"name": "e1", "age": 30},
 {"name":"e2", "age":32, "proj":["p1","p2"], "bonus":110},
 {"name":"e3", "age":28, "proj":["p1","p3"], "bonus":100}
]
```

Fig. 6. *Functions with Side Effects on Jsons*

4 Conclusion

In this paper we have presented a new educational functional programming language extended with imperative programming features: CalcuList, whose imperative features are enabled under explicit request by the programmer. CalcuList expressions and functions are first compiled and then executed each time a query is issued. The CalcuList programming environment has been implemented as a small-sized Java project in Eclipse 4.4.1 with 6 packages and 20 classes all together. The Jar File for using Calculist may be dowloaded from the link in [8]. The size of this file is rather small: 134 kb. The draft of a tutorial on CalcuList may be dowloaded from the link in [9].

References

1. K. Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 2013.
2. W. F. Clocksin and C. S. Mellish. *Programming in Prolog (2Nd Ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
3. ECMA Int. The JSON Data Interchange Format. Standard ECMA-404, Ecma International, 2013.
4. P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.
5. S. Marlow. Haskell 2010 Language Report.
6. J. Paakki. Attribute Grammar Paradigms – a High-level Methodology in Language Implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.
7. V. Reniers, D. Van Landuyt, A. Rafique, and W. Joosen. On the State of NoSQL Benchmarks. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, pages 107–112, New York, NY, USA, 2017. ACM.
8. D. Saccà. The CalcuList Environment, Release 4.1.0. https://github.com/saccad/CalcuList/blob/master/CalcuList_R_4_1_0.jar, 2017.
9. D. Saccà. The CalcuList Release 4.1 Tutorial, Version 1. https://github.com/saccad/CalcuList/blob/master/CalcuList_Tutorial_Release_4_v3_5.pdf, 2017.
10. D. Saccà and A. Furfaro. CalcuList: a Functional Language Extended with Imperative Features. *CoRR*, abs/1802.06651, 2018.
11. D. Saccà and A. Furfaro. Using calculist to mapreduce jsons. In B. C. Desai, S. Flesca, E. Zumpano, E. Masciari, and L. Caroprese, editors, *Proceedings of the 22nd International Database Engineering & Applications Symposium, IDEAS 2018, Villa San Giovanni, Italy, June 18-20, 2018*, pages 74–83. ACM, 2018.
12. C. Team. *CouchDB 2.0 Reference Manual*. Samurai Media Limited, United Kingdom, 2015.
13. G. van Rossum and J. Fred L. Drake. *The Python Language Reference Manual (version 3.2)*. Network Theory Limited, 2011.
14. P. Winston and B. Horn. *LISP. Second edition*. Addison Wesley Pub.,Reading, MA, Jan 1986.