Formalization of a Big Graph API in Coq

Jolan Philippe and Wadoud Bousdira
Univ. Orléans, INSA, CVL, LIFO, EA 4022, France
Email: firstname.lastname@univ-orleans.fr

Frédéric Loulergue
School of Informatics Computing and Cyber Systemes
Nothern Arizona University, USA
Email: frederic.loulergue@nau.edu

POSTER EXTENDED ABSTRACT

I. APIS FOR BIG GRAPH COMPUTATIONS

We now live surrounded by sensors, we create information continuously and we leave constantly computer traces of our activities. The processing and analysis of this huge volume data, so called Big Data, offer innumerable and still largely unexplored: health (epidemiology, genomics …), complex energy networks, intelligent cities, forecasting and management of environmental risks, etc. Big Data has, and will increasingly, a very significant impact at the societal economic and commercial levels. Many interesting Big Data problems can be modeled as problems on graphs/networks.

In the recent years, several API for big data processing have been proposed such as Pregel [1], Giraph, GraphX, GraphLab, etc. Even if these API are quite high-level, developing Big Data analyses remains error-prone. We are interested in providing tools for the development of correct Big Data applications.

II. PROGRAM VERIFICATION

By correct programs, we mean formally verified programs, and more specifically the correctness of programs with respect to a specification. There are two main approaches to program correctness in this context. In a *posteri verification* the specification and the program are written independently and the proof of correctness is done usually using a program logic related to Hoare logicand is more or less automated. Tools that support this kind of verification include for example Frama-C [2] for sequential C programs. In *correctness-by-construction* the specification is written first, and then step-by-step this specification is transformed into an efficient executable program. Program calculation, in particular of functional programs [3] are methods to develop correct-by-construction programs. Proof assistants are well suited to support program calculation of functional programs [4]. To our knowledge none of these approaches has been used yet to reason about Big Graph programs. The goal of the proposed work is to formalize a *functional* Big Graph API using the Coq [5] proof assistant. This formalization could then be used either for a posteriori verification or for correctness-by-construction using the SyDPaCC framework [6].

In practice, Coq can be seen as a functional programming language but with a rich type system that allows to express mathematical properties. Coq is based on the Curry-Howard correspondence: a program and a proof are of the same nature. This imposes some restrictions on the programs that can be written in Coq. In particular all functions should be terminating because a function can also be a proog and it is meaningless to have a non-terminating proff. One essential feature of Coq for this work is that is possible to extract functional programs in Haskell, Ocaml or Scheme from Coq developments.

III. FORMALISATION OF A BIG GRAPH API IN COQ

The need of a graph datatype that implements a sequential calculation for parallel computation led us to study the new domain-specific language Fregel [7] presented by Emoto et al. It is a functional approach to vertex-centric graph processing, which means that all vertices can compute in parallel. It promises good performance on large graph problem solving. That’s why, inspired by the Haskell implementation of Fregel, we propose one in Coq [5]. Basically such an API consists of: a distributed data structure for which there exists a sequential model, and a set of algorithmic skeletons [8] on this structure, i.e. functions implemented in parallel that have a semantics possibly modeled as sequential functions.

The main component of the Fregel API is a *Graph* datatype, defined as a list of vertex. A vertex is a record type that contains an identifier, a value and a list of incoming edges. This is a feature of the vertex-centric model: the edges are stored as incoming edges of the vertices, and not as outgoing edges. We hope to have better performances if the vertices are going to get the information than if they receive it as a message. In our case, only one vertex computes, the one which get the information. In a classical case, the first vertex works, by sending a message, and the second works too by getting the information.

Graphs are also composed of edges : the *Edge* type has two fields, the weight of the edge, and the identifier of the source vertex. An edge can only exist in relation to a target vertex because of the absence of target vertex in its definition. The following Coq code realizes these types implementation:

978-1-5386-3250-5/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCS.2017.140

Record Edge B := edge {weight : B ; incoming : Z}.
Record Vertex A B := vertex {vid : Z; val : A;
is : list (Edge B)}.
Defintion Graph A B := list (Vertex A B).

A and B are type variables : these types are polymorphic. You can notice that we used the type Z to define the vertex identifiers. It's a Coq implementation for the arithmetic integer, coded in a binary way.

Four skeletons are proposed by Fregel. Each one has its purpose, but they use same tools. For the ones which use an iteration on the graph value, a condition of termination must be established. The type *Termination* is defined inductively, with three kinds of termination for any structure to be treated. The first one is a maximum number of iterations, the second one is a condition of stop (can be assimilated as a *while* loop) and the last one is the fixed state of the graph. For the *Fixp* condition, the equality of two graphs must be defined, and shown to be decidable.

Inductive Termination T := | Iter (n:Z)
| Fixp (eqG: T→T→bool) | Until (cond : T → bool).

The first skeleton, the function *gmap*, will just apply a function to each element on a *Graph*. It looks like a classical map function on lists (we just give its signature):

gmap: (f:Vertex A B → R) (g:Graph A B) : Graph R B

The second one, *gzip* is generalization to two graphs. The two last ones, *giter* and *fregel*, are iterative functions. For each kind of termination, an iterate corresponding function must be defined with a stop control governed by a maximum number of iterations : *max_op*. Now we are sure that the recursive functions stop in all cases. To use this king of function, an initialization has to be made before. You can apply a function to transform each default value with a coherent value for the problem (Boolean for vertex-cover problem for example). On the original Haskell functions, the iterate functions use an infinite list of graph results, and they take the first one which satisfied the termination condition. We cannot do that in Coq programming, we have to specify a decreasing argument on all recursive functions. If none argument is decreasing naturally (i.e. the recursive calls are done on *syntactically* smaller terms), you have to prove there is one measure (returning a natural number) for which the recursive calls are done on smaller arguments for this measure.

The first iterative function, *giter*, uses directly the graph and iterates a function on it. The other one uses a save of values to iterate and only at the end, it creates a new graph with final values. The easier is the first one, because of its simplicity to implement. You apply your step function on all elements (thanks to the *gmap* function, defined earlier). If the termination condition is satisfied, or if the maximum number of iterations is reached, you stop the compute and you return the graph with new values. The ids of vertices are the same so you can match easily the old values on the old graph with the new ones on the result.

giter (init: Vertex A B → R) (iter : Graph R B → Graph R B)
(term: Termination (Graph R B)) (g: Graph A B)
(max_op : Z) : Graph R B

The fregel operation is the most important. It looks like *giter* but it uses aspects of dynamic programming. The values are not directly stored in the vertices, but in two lists of computation results: one to get the current values, and the other to keep the previous values. Here again, an initialization must be done. You have to begin with an association between the vertex values, and their corresponding value on the resolution of the problem. By the way, on this first step, the two lists of computation results are equivalent, defined with the values on the problem. As the *giter* function, you iterate your step function on the graph using the data stored on the both. At each iteration, the list of current values takes the results of the step function. The condition of termination is the same as on *giter*.

fregel (init: Vertex A B → R)
(step: Vertex A B→(Vertex A B → R)→(Vertex A B → R)→R)
(term: Termination (Graph R B)) (g: Graph A B)
(max_op : Z) : Graph R B

As an example of a Fregel application, you can take the problem of the reachable vertices from a source. The initialization function will associate false to each vertex, except to the source vertex which has a true value. The iterate function will associate tests if an incoming vertex by an edge is reachable by current values, or by previous values. If the incoming vertex is reachable, the destination is too. The best termination control is the fix state of the graph by the iteration function, but you can use a predicate to fix a maximum number of reachable vertices. *Define* abbreviations and acronyms the first time they are used in the text, even after they have been defined in the abstract.

[1] G. Malewicz, M. H. Austern, A.J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel : a system for large-scale graph processing" in *SIGMOD.* ACM, 2010, pp. 135-146

[2] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C : A software analysis perpective," *Formal Asp. Comput.*, vol, 27, no, 3, pp. 573-609, 2015.

[3] R. Bird and O. de Moor, *Algebra of* Programming. Prentice Hall, 1996.

[4] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi, "Program Calculation in Coq," in Algebraic Methodology And SoftwareTechnology (AMAST), ser. LNCS 6486. Springer, 2010, pp. 163–179.

[5] The Coq Development http://coq.inria.fr. Team, "The Coq Proof Assistant,"

[6] F. Loulergue, W. Bousdira, and J. Tesson, "Calculating Parallel Programs in Coq using List Homomorphisms," Int J Parallel Prog, vol. 45, pp. 300–319, 2017

[7] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, and H. Iwasaki, "Think like a vertex, behave like a function! a functional DSL for vertex- centric big graph processing," in ACM SIGPLAN International Conference on Functional Programming (ICFP). ACM, 2016, pp. 200–213.

[8] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, 1989, available at http://homepages.inf.ed.ac.uk/mic/Pubs.